

Automatic Transformation of UML Models into Analytical Decision Models

Sunny Huynh, Yuanfang Cai
Drexel University
Philadelphia, PA 19104
{sh84, yfcai}@cs.drexel.edu

Wuwei Shen
West Michigan University
Kalamazoo, MI 49008
wuwei.shen@wmich.edu

ABSTRACT

Emerging software dependency models, such as *design structure matrices* (DSMs), have been used in software design to capture and assess software modular structure and modularization activities. However, thinking and modeling design in terms of decisions and enumerating their dependencies are not straightforward. On the other hand, the *Unified Modeling Language* (UML) is a well-known and widely-used modeling technique. UML diagrams embody important design decisions and their relations. This paper presents an approach to automatically transform a UML class diagram into a logical design model, from which a DSM model can be automatically derived and the system can be automatically decomposed into modules (independent task assignments). Our approach is to formalize the dependency relations of UML class diagrams, automatically translate these relations into *augmented constraint networks* (ACN), decompose the ACN model into smaller sub-models, and derive DSM models. We use a small example to illustrate our approach, and evaluate the approach using a UML class diagram reverse engineered from Apache Ant. We show that our approach enables automatic translation of a UML model into analytical decision models and support automatic decomposition of a large system into independent decision *modules*.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design; D.2.11 [Software Engineering]: Software Architecture

General Terms

Design, Modeling

1. INTRODUCTION

Emerging analytical decision models, such as *design structure matrices* (DSMs) [32, 18, 9], have shown potential utility in software design. DSM models have been used to explicitly express design decisions and design modular struc-

tures [26, 31], to precisely capture information hiding criteria [34, 35], to explain software evolution phenomena and modularization activities [23, 26, 12], and to assess the value of modular structure [34]. As the counterpart of DSM modeling at design level, our prior work on *augmented constraint network* (ACN) explicitly models design decisions and their relations as logical constraints, provides precise semantics of *pair-wise dependency*, and enables automatic derivation of DSM models from logical models, as well as other analysis techniques, such as change impact analysis [13, 10]. We have also shown that by comparing the DSM model derived from an ACN design model and the DSM generated from the source code of the same project, we were able to identify implicit dependencies and improper implementations that are caused by modularity deviation [21].

Despite their utilities, constructing design-level analytical decision models is not straightforward. Both DSM and ACN modeling require the designer to think in terms of design decisions. We have shown that manually marking the dependencies of a DSM is ambiguous and error-prone [13, 10], while constructing an ACN model requires the ability to model design decisions and their relations using logical expressions. These difficulties limit the application of these models in practice.

By contrast, the *Unified Modeling Language* (UML) [29, 30] is a well-understood and widely-used modeling technique. UML modeling is not designed to generally model design decisions, nor to reveal design modular structures. For example, UML class diagrams use boxes and lines to model classes and their relations. However, important design decisions and their relations are embodied by these models. For example, a class can be viewed as a design dimension, and the relations among classes, such as inheritance and association, determine whether and how a change in one class will influence other classes.

In this paper, we present a framework to automatically derive analytical decision models, ACNs and DSMs, from widely-used UML class diagrams. Our purpose is to reap the best from both sides, enabling software designers, who are familiar with UML modeling, to take advantages of modularity analyses enabled by ACN and DSM modeling. The size of a UML class diagram can easily scale to the extent that comprehension of the overall system becomes difficult [17]. Our prior work presented an approach to automatically decompose a large ACN model into a set of smaller sub-ACN modules, each representing a set of decisions needed to realize a particular feature [14]. Our technique thus enables automatic module decomposition from large UML models.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

Design Structure Matrix.

We recently have shown that *design structure matrix* (DSM) modeling has the potential to address some of these problems by explicitly modeling design decisions and expressing software modular structure [34, 35, 26, 31, 23]. A DSM is a square matrix in which rows and columns are labeled with design dimensions in which decisions have to be made. We view each class as having two design dimensions, an interface (ending with `_interface`) and an implementation (ending with `_impl`). Figure 2 shows a DSM that models the maze game using the abstract factory pattern, in which the 12 classes shown in Figure 1 is modeled using a matrix with 24 rows and columns. A marked cell in a DSM models that the decision on the row depends on the decision on the column. For example, the cell in Line 4, Column 1 indicates that the implementation of the class `Room`, `Room_impl`, depends on the interface of `MapSite`, `MapSite_interface`.

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

Figure 2: Maze Game DSM

Augmented Constraint Network.

Our recent work has shown that manually-constructed DSMs can be ambiguous and error-prone [10, 13]. To address this problem, we have developed a design representation called the *augmented constraint network* (ACN) to precisely represent the relations among design decisions using logical constraints. An ACN model consists of a constraint network and two augmenting data structures to express software design activities that cannot be readily captured by logical expressions.

(1) Constraint network: a constraint network consists of a set of variables and constraints.

Each variable models a design dimension in which a decision is needed and has a *domain* in which each *value* represents a possible choice. For example, we can model the choice of an sorting algorithm as *alg* : {*bubble*, *quick*, *other*}. Take the class `Room` in the maze game as an example, we model its interface dimension as `Room_interface` : {*orig*, *other*}. In this paper, we use *orig* to generally model an original decision, and use *other* to model a possible future change.

The constraints among these decisions are represented as logical expressions. Figure 3 shows the constraints for the design of a bombed maze game. For example, Line 11 in Figure 3 models that the implementation of `BombedMazeFactory` assumes that the interface of `MazeFactory` is as originally

```

1: MazeFactory_impl = orig =>
    MazeFactory_interface = orig;
2: BombedMazeFactory_impl = orig =>
    BombedMazeFactory_interface = orig;
3: Room_interface = orig => MapSite_interface = orig;
4: Wall_interface = orig => MapSite_interface = orig;
5: Door_interface = orig => MapSite_interface = orig;
6: MazeFactory_impl = orig => Maze_interface = orig;
7: MazeFactory_impl = orig => Room_interface = orig;
8: MazeFactory_impl = orig => Door_interface = orig;
9: MazeFactory_impl = orig => Wall_interface = orig;
10: BombedMazeFactory_interface = orig =>
    MazeFactory_interface = orig;
11: BombedMazeFactory_impl = orig =>
    MazeFactory_interface = orig;
12: BombedMazeFactory_impl = orig =>
    MazeFactory_impl = orig;
13: BombedMazeFactory_impl = orig =>
    RoomWithABomb_interface = orig;
14: BombedMazeFactory_impl = orig =>
    BombedWall_interface = orig;
15: BombedWall_interface = orig =>
    Wall_interface = orig;
16: RoomWithABomb_interface = orig =>
    Room_interface = orig;

```

Figure 3: Bomb Maze Game Constraints

agreed. In the next section, we explain in detail how the relations among classes are translated into logical expressions.

(2) Augmenting data structures. Our ACN framework uses a *dominance relation* to model the fact that some design decisions dominate other decisions, for example, interfaces dominate implementations, and uses a *cluster set* to model different ways a system can be modularized.

The dominance relation is a binary relation. For example, (`BombedMazeFactory_impl`, `MazeFactory_interface`) is a member of the maze game dominance relation, modeling that the decision on the interface of `MazeFactory` dominates the decision on how the `BombedMazeFactory` should be implemented. That is, changes in `BombedMazeFactory`'s implementation cannot force the change of the interface of `MazeFactory`. Dominance relation formalized Baldwin and Clark's design rule concept [9], and we have shown that the nontrivial dominance relation plays a key role to decompose a system into modules [14].

Each member of the *cluster set*, a *cluster*, is a tree structure modeling how the variables can be aggregated together as modules. For example, in the maze game, we can aggregate all the interfaces together as a *design rule* module, or aggregate all the classes implementing the same function as a module. We have shown that a DSM can be automatically derived from an ACN [13, 10]. In our prototype tool, Simon, the user can choose a cluster according to which the rows and columns of the derived DSM will be ordered.

However, both DSM and ACN modeling require designers to think in terms of design decisions and ACN modeling requires designers to express design in terms of logical expressions, which can be difficult for software designers who are not trained for this kind of modeling. On the other hand, prevailing models, such as UML, contain significant design decisions and their relations. Accordingly, we first formally define and extract the logical relations and dominance relation from UML diagrams, and then automatically generate an ACN model from a UML model. Base on the ACN model,

we can decompose a system into independent modules, each representing the set of decisions that are needed for a component. We can also automatically generate DSM models with precise semantics.

2.2 From UML to Decision Models

We take the maze game UML as shown in Figure 1 to illustrate how to transform a UML class diagram to an ACN, to decompose a big ACN into a number of smaller sub-ACNs, and to derive DSM models.

From UML to Logical Decision Models.

We consider classes, interfaces, and their relations in a UML class diagram for transforming in our approach. We model classes and interfaces using variables. For example, we view the class `Room` in the maze game example as having two dimensions, its interface and implementation. We thus model them using two variables, `Room_interface : {orig, other}` and `Room_impl : {orig, other}`. Our framework automatically translate the 12 classes in Figure 1 into 24 variables.

We model the relations among classes and interfaces as logical expressions. For example, class `BombedMazeFactory` inherits from class `MazeFactory`. It means that the change in both the interface and implementation of `MazeFactory` will influence the implementation of `BombedMazeFactory`. In other words, the implementation of `BombedMazeFactory` makes assumptions about the interface and implementation of `MazeFactory`. Line 11 and 12 in Figure 3 model these relations. Although a change in the implementation of a super class will implicitly propagate to a subclass, and the code of the subclass may not need to be changed explicitly, the designer should be aware of this implicit change. Line 12 in Figure 3 makes this relation explicit.

To automatically generate the dominance relation, we dictate that interfaces dominate implementations, and that the implementation decision of a super class dominates the implementation decision of a subclass. The reason is that the super class may have other subclasses. Forcing a super class to change due to the change of a subclass may cause unwanted side effects to other subclasses. In Section 3 we introduce the formal transformation from UML class diagram elements to ACN components. Since obtaining the cluster set is trivial, in this paper, we only consider the constraint network and dominance relation part of an ACN.

The ACN model automatically generated from the maze game UML class diagram (as shown in Figure 1) consists of 24 variables, 50 lines of logical expressions, and its dominance relation has 49 pairs. In our framework, the user does not need to write the ACN model manually, nor to understand the model as a whole. Instead, as we will show next, the whole ACN can be decomposed into 7 sub-ACNs, each representing all the decisions needed for one component. We can also generate DSM models from each of the sub-ACN, and compose it into full DSM. In other words, the ACN model can be totally transparent to the user.

ACN Model Decomposition.

In our previous work [14], we introduced an algorithm to automatically decompose a big ACN model into smaller sub-ACNs, to address the scalability issue caused by the difficulty of constraint solving. The basic idea is to model the constraint network part of the ACN using a directed graph.

In this graph, each node represents a design variable. Two variables are connected if they appear in the same constraint expression. We then cut the edges of the directed graph using the non-trivial dominance relation of the ACN: if `A` cannot influence `B`, then the edge from `A` to `B` will be removed from the graph.

After that, we compute the condensation graph of the resulting graph [37, 8], and put all the variables along the chains ending with the same minimal elements into a sub-ACN, with the relevant subset of constraints, dominance relation and cluster set. As a result, we decompose the big ACN into a set of smaller sub-ACNs that can be solved individually. We also presented the algorithm of integrating analysis results of sub-ACNs into a full solution in our prior work [14].

More interestingly, we found 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’s functionality. In other words, each sub-ACN has the potential to be an independent responsibility assignment module. The maze game example is thus decomposed into 7 sub-ACNs, representing the 7 tasks as shown in Table 1. Figure 3 shows all the constraints in a decomposed sub-ACN modeling the task of implementing the `BombedMazeFactory` (module 1).

Table 1: Decomposed MazeGame

No.	Module	# Variables
1	BombedMazeFactory_impl	11
2	BombedWall_impl	6
3	DoorNeedingSpell_impl	7
4	EnchantedMazeFactory_impl	11
5	EnchantedRoom_impl	6
6	Maze_impl	4
7	RoomWithABomb_impl	6

DSM Derivation.

Our prior work presents an approach of automatically generating a DSM from an ACN model [10, 13]. The basic idea is to first formally define the semantics of *pair-wise dependency* based on constraint networks: if x depends on y , then there must be a change in y that makes the constraint network inconsistent, and at least one of the minimally disruptive repairs involves a change in x . According to this definition, we have created an algorithm to automatically compute all the pairs that depend on each other in an ACN. A DSM can thus be automatically derived by using these pairs to populate the matrix and using a selected cluster to order the columns and rows. Figure 4 shows the DSM derived from the bombed maze game sub-ACN, and Figure 5 shows the DSM derived from the enchanted maze game sub-ACN. Our prior work also presented the algorithm of composing the whole DSM from sub-DSMs. Figure 2 is the full DSM composed from the 7 sub-DSMs.

From the full DSM shown in Figure 2, we can see that the maze game design can be viewed as a layered system. The sub-DSMs in Figure 4 and Figure 5 are part of the full DSM. The gray areas in Figure 4 and Figure 5 cover the variables involved in both bombed and enchanted sub-DSMs. We observe that each decomposed DSM models a *subsystem module*, in which some of the variables may be shared with other

	1	2	3	4	5	6	7	8	9	10	11
1 MapSite_interface	.										
2 Room_interface	x	.									
3 Wall_interface	x		.								
4 Door_interface	x			.							
5 Maze_interface					.						
6 MazeFactory_impl	x	x	x	x	x	.	x				
7 MazeFactory_interface								.			
8 RoomWithABomb_interface	x	x							.		
9 BombedWall_interface	x		x							.	
10 BombedMazeFactory_impl	x	x	x	x	x	x	x	x	x	.	x
11 BombedMazeFactory_interface								x			

Figure 4: A sub-DSM: Bomb Maze Game

	1	2	3	4	5	6	7	8	9	10	11
1 MapSite_interface	.										
2 Room_interface	x	.									
3 Wall_interface	x		.								
4 Door_interface	x			.							
5 Maze_interface					.						
6 MazeFactory_impl	x	x	x	x	x	.	x				
7 MazeFactory_interface								.			
8 EnchantedRoom_interface	x	x							.		
9 DoorNeedingSpell_interface	x			x						.	
10 EnchantedMazeFactory_impl	x	x	x	x	x	x	x	x	x	.	x
11 EnchantedMazeFactory_interface								x			

Figure 5: A sub-DSM: Enchanted Maze Game

subsystem modules. The subsystem modules are not readily shown in a full DSM in which each module is traditionally viewed as a block along the diagonal, and these blocks either do not overlap with each other, or can only overlap with their neighbors. As a result, it is not straightforward to show how design decisions are shared among multiple subsystems.

3. MODEL TRANSFORMATION

In this section, we present the formalization of UML class diagrams using augmented constraint networks. For the sake of space, we only present the formalization of UML classes, interfaces, and their major relations. These UML elements covered in this section are all supported by IBM Rational Rose [3] and used in Apache Ant [7], the real system we experimented with.

3.1 UML Classes and Interfaces

From the design decision perspective, each UML class consists of two design dimensions: the interface dimension, and the implementation dimension. As a result, we model a class, *A*, using two variables: *A_interface* and *A_impl*. Each dimension can vary, so we model the domain of each variable as having at least two values (*orig*, *other*), where *orig* models the current decision and *other* models an unelaborated new choice for the decision.

The ACN translated from class *A* is shown as below. Besides the two variables, we use a logical expression to model that the implementation of *A* makes assumptions about its interface. We also assume that the interface of a class dominates its implementation, which is translated into a dominance relation. As a result, we translate the class *A* into the following ACN:

```
Constraint Network:
  A_interface : (orig, other)
  A_impl : (orig, other)
```

```
A_impl = orig => A_interface = orig
Dominance Relation:
(A_impl, A_interface)
```

Since a UML interface, *I*, has no implementation, we model it using only one variable, the interface variable:

```
I_interface : (orig, other)
```

3.2 UML Relations

Given that all the classes and interfaces are translated into variables, we now translate their relations into logical expressions and dominance relation of an ACN.

Dependency, Association, Aggregation, and Composition.

We consider these four relations together because from the decision-making perspective, the assumption relation among the elements of these relations are the same, and we map them into an ACN in the same way. The four diagrams in Figure 6 depict these relations in order.

A *dependency* exists between two elements, *A* and *B*, if a change to the definition of one could result in a change to the other. This is indicated by a dashed arrow pointing from the dependent to the independent element.

An *association* represents a structural relationship between two classes; if there exists an association between classes *A* and *B* then objects of class *A* can be related to class *B*, such as “department offers courses”. This is indicated by a line with each end connected to a class box.

Aggregation is a variant of the “has a” or association relationship. In UML, it is graphically represented as a clear diamond shape on the containing class end of the line that connects the contained class to the containing class. Aggregation can occur when a class is a collection or container of other classes, but where the contained classes do not have a strong life cycle dependency on the container—essentially, if the container is destroyed, its contents are not. For example, “a professor has classes” is an aggregation relation.

Composition is a stronger variant of the “has a” association relationship, such as “a cars has four wheels”. It is represented by a filled diamond shape on the containing class end of line that connect contained class to the containing class. Composition has a strong life cycle dependency between instances of the container class and instances of the contained class(es): If the container is destroyed, every instance that it contains is destroyed as well. A composition is an aggregation that must satisfy the following two constraints: (1) an object may be part of only one composite at a time, and (2) the composite object has sole responsibility for the management of all its parts. Obviously, composition relationship is a strong form of aggregation relationship.

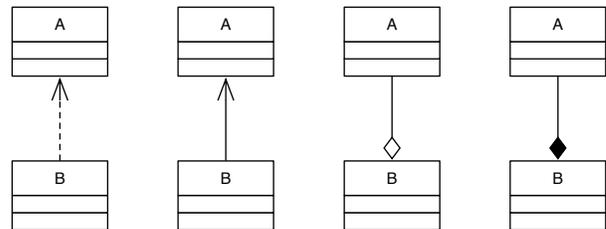


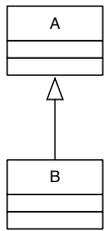
Figure 6: Dependency, Association, Aggregation, and Composition

Viewed in terms of design decisions and assumptions, the four relations shown in Figure 6 have the same assumption relations. A dependent class or a containing class, B, makes assumptions about the interface of the independent class or contained class, A. In other words, the change in the interface of A will influence B’s implementation, but not vice versa. Their logical expression and dominance relation are as follows:

Constraint Network:
 $B_impl = orig \Rightarrow A_interface = orig$
 Dominance Relation:
 $(B_impl, A_interface)$

Generalization.

The diagram in Table 2 shows a UML class diagram for a *generalization* relation. The generalization relation consists of two elements, one general and one specific. Each instance of the specific element is indirectly an instance of the general element, therefore the specific element inherits the features of the general element [29]. The specific element can also provide more features than the general element.

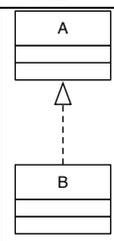
Table 2: Generalization	
UML	Augmented Constraint Network
	Constraint Network: $B_impl = orig \Rightarrow A_interface = orig$ $B_interface = orig \Rightarrow A_interface = orig$ $B_impl = orig \Rightarrow A_impl = orig$ Dominance Relation: $(B_impl, A_interface)$ (B_impl, A_impl) $(B_interface, A_interface)$

In the diagram shown in Table 2, A is the general element and B is the specific element. Since B inherits the features of A, this means the design decision of A’s interface dominates and influences B, both its interface and its implementation, which is modeled using the first two logical expressions shown in the table.

Moreover, the implementation decision of A also influences the implementation of B. Although the changes in A will be propagated to B implicitly, and the user may not need to change B’s code, she/he should be aware of this change to avoid unwanted side effects. The third logical expression makes this relation explicit. The A_impl also dominates B_impl because a change in B should not force changes in A, which may be inherited by other classes. The dominance relation is thus constructed accordingly.

Interface Realization.

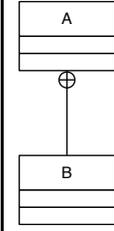
The diagram in Table 3 shows the *interface realization* relation in UML class diagrams. In the realization relation, one element (the supplier), A, provides a specification, and another element (the client), B, provides an implementation matching said specification [29]. Similar to the *Generalization* relation, we model the relations among the decisions of A and B using an ACN shown in Table 3, which is different from the Generalization relation in that the parts involving A_impl are removed, because the interface A does not have an implementation.

Table 3: Realization	
UML	Augmented Constraint Network
	Constraint Network: $B_impl = orig \Rightarrow A_interface = orig$ $B_interface = orig \Rightarrow A_interface = orig$ Dominance Relation: $(B_impl, A_interface)$ $(B_interface, A_interface)$

Owned Element Association.

The diagram in Table 4 shows a owned element association, representing that the class B (inner class) is nested within the class A (outer class), that is, B is declared in the scope of the outer class A. An outer class and its inner class are connected by an anchor line, with an anchor icon which appears as a plus sign inside a circle on the end connected to a outer class.

In this relation, if A’s interface or implementation (e.g. an inner variable used by B) changes, B’s implementation may need to change. This means that the design decisions of both A’s interface and implementation dominate and influence B’s implementation, which is modeled as the ACN shown in the table.

Table 4: Owned Element Association	
UML	Augmented Constraint Network
	Constraint Network: $B_impl = orig \Rightarrow A_interface = orig$ $B_impl = orig \Rightarrow A_impl = orig$ Dominance Relation: $(B_impl, A_interface)$ (B_impl, A_impl)

3.3 Discussion

In this paper, we only present the formalization of the major UML class diagrams that are used in our experiment with Apache Ant. A relation can have variation, however, our framework only concerns the assumption relations among these component, and these variations, such multiplicity in an association relation, do not change the way their ACN model are generated.

There are also higher order associations with more than two ends, and an association can have role names, ownership indicators, multiplicity, and visibility adorned in the end. There are several variations of association, but we only show a uni-directional, navigable association in the second diagram of Figure 6. Objects of class B can access objects of class A but not vice versa.

The *object constraint language* (OCL) [28] also expresses constraints and relations in UML diagrams. However, we do not use OCL because it does not model dominance relation and cluster set and does not provide the functionalities we desire, such as module decomposition and DSM derivation. Although, one of our future works is to incorporate OCL

specified constraints into the ACN we generate from a UML diagram.

4. FRAMEWORK AND REALIZATION

Figure 7 shows an overview of our framework, which consists of four tools: UML-to-ACN Converter—converts a UML class diagram into an ACN; ACN Decomposer—decomposes a big ACN into a set of smaller ACNs; DSM Composer—composes sub-DSMs into a full DSM; and DSM Generator—generates a DSM directly from an ACN. After the DSMs are generated, a number of DSM-based analyses can be conducted, such as net option value analysis and design change impact analysis [36, 13, 10]. Our prototype tool, Simon [13, 14], has the functionality of ACN decomposer, DSM composer and DSM generator, as well as these analyses. We have built another prototype, uml2acn, to translate a UML diagram into an ACN according to the formalization shown in Section 3. In this section, we mainly present how the UML-to-ACN component works.

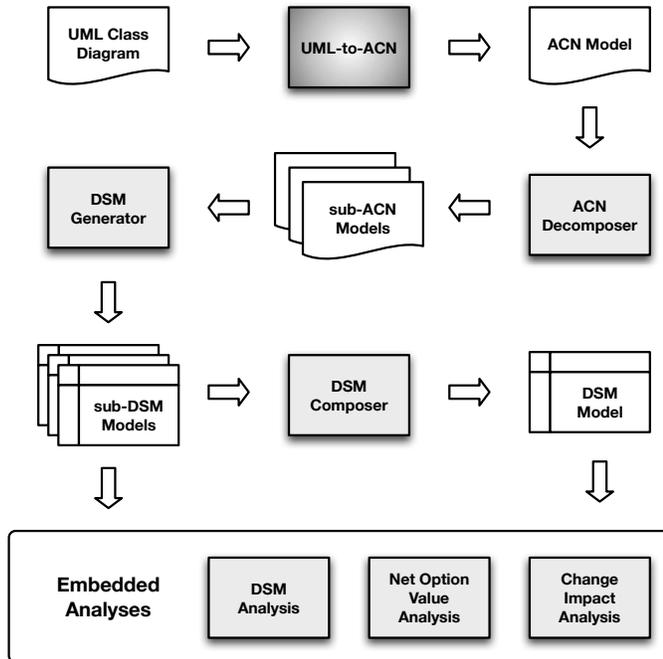


Figure 7: Framework Overview

Our tool currently accepts UML class diagrams in the file formats of IBM Rational Rose [3], Dia [2] (the open-source diagram editor), and XMI [27]. Within the UML-to-ACN component, the UML class diagram is first analyzed for its elements (classes and interfaces) and the relations between these elements. These relations are extracted to a common intermediate format so that new file format inputs can be easily added to our approach without changing the ACN related parts of the algorithm. From the intermediate file, the UML-to-ACN component performs the model transformation, and outputs an ACN project required by Simon.

5. PRELIMINARY EVALUATION

This section presents the experiment we conducted to preliminarily evaluate our approach. The following subsections

introduce the overview of our experiment, the process, and the results.

5.1 Experiment Overview

Using our prototype tools, uml2acn and Simon, our experiment aims to show (1) the framework is applicable to a project with hundreds of classes; (2) the framework can decompose a big UML class diagram into a number of smaller subsystem modules, each faithfully capturing all the tasks needed for a particular task.

To test the first claim, we applied the framework to a non-trivial UML class diagram model reverse-engineered from a real open-source project, Apache Ant [7]. Apache Ant [7] is developed in Java for automating software build processes. The current latest version of Apache Ant, 1.7.0, has a core library (called “ant.jar”) containing approximately 700 classes and 93 KSLOC.

Although our ultimate purpose is to facilitate modularity analysis at design level, we could not find a publicly available UML model with hundreds of classes, the size of a real, middle-sized project. Therefore, since UML diagrams were not available for the project, we reverse engineered the class diagram from the available source code and binary files. We used a combination of Rational Rose [3] and Dependency Finder [1] to extract all the inter-component relations in the system. Rational Rose extracted the interface level dependencies (e.g. class inheritance, method signatures, nested classes, etc.) while Dependency Finder provided the implementation level dependencies (e.g. classes used within method implementation bodies). We had to use both Rational Rose and Dependency Finder because Rational Rose could not detect implementation level dependencies. We could not find any available tools that could distinguish both types of dependencies for us. With a more advanced reverse engineering tool, we would not need to use two tools for deriving the UML class diagram information.

Figure 8 shows a small portion of the UML class diagram we derived. It is obvious that a UML diagram of this size becomes very difficult to understand. People have created methods to derive abstract views of such diagram [15]. We show that our framework will decompose the diagram into much smaller modules so that each can be comprehended independently.

To test the second claim of correctness, we compare the UML-derived DSM (design DSM) with the DSM derived by Lattix [24], directly from the source code (source DSM). In Lattix DSM, the variables are the classes, functions, and variables, and the dependencies are direct, syntactic references, including the usage, inheritance, and invocation relations among these constructs. By default, Lattix uses classes and packages as units, and automatically derives a DSM from a source code repository. The user can configure Lattix to show lower level program units, like variables and functions.

Since the UML model is reverse engineered from the source code, the design DSM and source DSM are consistent with each other. For a function represented by a sub-DSM, a subset of the full design DSM, we find the same function in Lattix, move together all the classes the function depends on to form a block, representing a module with all the classes needed to realize the function. After that, we compare the block, a subset of the source DSM, with the design-level sub-DSM, to see if there are any discrepancies. If there are,

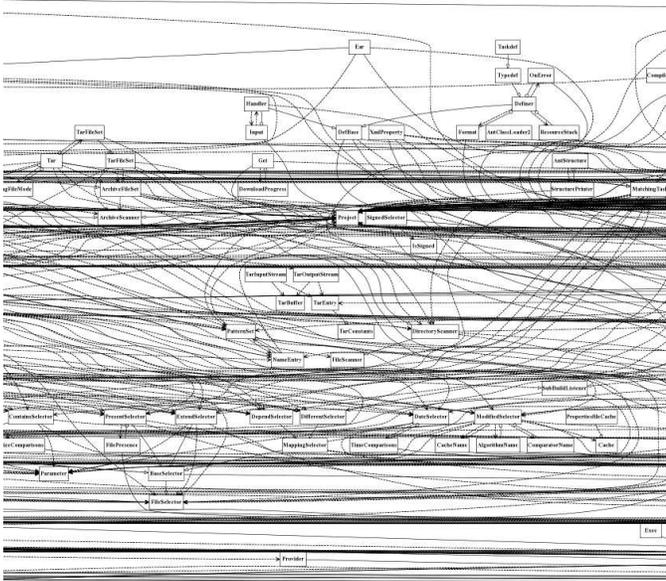


Figure 8: Apache Ant UML Diagram

we further analyze the reason behind: do the dependencies provide additional insights? Which DSM faithfully captures all the dimensions needed for a function?

5.2 Experimental Process

Starting with the UML class diagram of Apache Ant shown in Figure 8 as the input, we transform the model in our framework as shown in Figure 7, going through the following stages:

UML-to-ACN Transformation.

The input UML model contains 626 classes and interfaces with over 3000 inter-component relations. Stored in Rational Rose, this file is 14MB in size. Our intermediate file during the transformation process is 378KB. The resulting ACN contains 1200 variables and about 4400 constraints.

ACN Model Decomposition.

We use Simon [13, 10] to decompose the large ACN model into 435 sub-ACN models. These sub-ACN models have between 2 and 86 variables each. As shown in our previous work [14], each sub-ACN represents a task, whose class implementation dimension is one of the minimal elements in the condensation graph. Accordingly, we use the name of the minimal element as the name of the ACN. For example, a sub-ACN named `org.apache.tools.ant.types.resources_selectors_Name_impl` means that this sub-ACN concerns the implementation of the `Name` class.

DSM Derivation.

We use Simon [13, 10] to derive DSM models for each of the decomposed modules. For any function represented by a module, we identify a block representing the same functionality in the Lattix DSM and compare them.

5.3 Results

We discuss our results from the following two perspectives:

Feasibility.

Our `uml2acn` tool converts this UML model into a ACN model in approximately 15 minutes, and it takes Simon a few seconds to decompose the big ACN into 435 sub-ACNs. Of the 15 minutes for running `uml2acn`, approximately 10 minutes were spent parsing the Rational Rose input file, 2 minutes on generating the intermediate file, and 3 minutes on producing the ACN project required by Simon. Our current implementation of `uml2acn` is still in a prototype state with no performance optimizations, and the performance can be substantially improved. We discuss our future work for optimizing the `uml2acn` tool in Section 6. The experiment shows that it is feasible to apply this framework to the UML diagram modeling this real project.

Accuracy.

To check if a decomposed subsystem module indeed captures all and only the dimensions needed for a function, we compare its sub-DSM with a block representing all the classes the function depends on in the source DSM generated by Lattix. We perform this comparison on about a dozen of the modules of varying sizes. We find that a subsystem module indeed captures all and only the decisions and dependencies needed for a function, and our DSMs derived from the UML model provide additional insights than the corresponding source code DSMs. We did not compose the full DSM of Ant to compare with the full Lattix DSM due to the size of the system. Instead, our divide-and-conquer approach provides a much more efficient way to make detailed comparison for each subsystem

We found two types of discrepancies between design and source sub-DSMs: (1) the design sub-DSMs reveal indirect and implicit dependencies; (2) the design DSMs lack the dependencies irrelevant to the task it models. In other words, a subsystem module decomposed using our approach captures exactly the set of dimensions needed for the task.

We use one of the smaller modules decomposed by our technique for illustration, a module regarding the implementation of the `org.apache.tools.ant.types.resources_selectors.Name` class. To ease the comparison, we copy and paste the design and source sub-DSMs into a spreadsheet and mark the differences. Figure 9 shows the design sub-DSM of this module generated by our approach, and Figure 10 shows the corresponding source sub-DSM extracted from Lattix.

	1	2	3	4	5	6	7
1 SelectorUtils interface	.						
2 ProjectComponent interface	.	.					
3 DataType interface	x	.					
4 Resource interface	x	x	.				
5 ResourceSelector interface					.		
6 Name interface					x	.	
7 Name_impl	x	x	x	x	x	x	.

Figure 9: Our Generated ACN for Name Module

(1) Figure 9 has one more element than Figure 10. This is

	1	2	3	4	5	6
1 SelectorUtils	.			?		
2 ProjectComponent		.				
3 DataType		x	.			
4 Resource			x	.		
5 ResourceSelector				x	.	
6 Name	x			x	x	.

Figure 10: Lattix Generated ACN for Name Module

because our framework models a class as having two dimensions, interface and implementation, while a Lattix DSM doesn't. So the last two elements in Figure 9 correspond to `Name` in Figure 10.

(2) Three dependencies exist in Figure 9, but not Figure 10, which are marked using dark background and white font. The discrepancies are due to the fact that our UML-based DSMs explicitly reveal implicit and indirect dependencies, for example, caused by inheritance:

First, the design sub-DSM has dependencies from `Name_impl` to `DataType_interface` and `ProjectComponent_interface` while this dependencies are not captured by the Lattix sub-DSM. The reason is that the `Resource` class inherits from the `DataType` class, and if the `DataType` class's interface changes then the `Resource` class's interface would also automatically change. Since the implementation of the `Name` class uses the `Resource` class, it is depending on the `Resource` class's interface. Hence if the interface of `DataType` changes, it would change the interface of `Resource` and potentially require a change to the implementation of the `Name` class. Therefore, although there are no explicit dependencies from `Name` to `DataType`, the inheritance hierarchy of `Resource` implies this implicit dependency. `Name_impl` depends on `ProjectComponent_interface` for the same reason. Therefore, our approach makes these implicit dependencies explicit.

Second, the DSM generated by our approach has a dependency from `Resource_interface` to `ProjectComponent_interface` but this dependency is missing from the Lattix generated DSM. Since the `Resource` class derives from `DataType`, which derives from `ProjectComponent`, part of the interface of `Resource` is defined in `ProjectComponent`. If the interface of `ProjectComponent` changes then the interface for `Resource` also automatically changes. Hence, there is an implied dependency from the interface of `Resource` to `ProjectComponent`.

Most of the discrepancies we found in our experiment are due to this kind of indirect and implicit relations among classes, which are not captured by Lattix DSMs. It is important to expose these dependencies to designers explicitly.

(3) There exists a dependency in the Lattix source sub-DSM in Figure 10 that does not exist in our design sub-DSM in Figure 9. The reason is that our approach splits each class into two separate design decisions (interface and implementation) while Lattix does not. The dependency expressed in the Lattix DSM is that the `SelectorUtils` class uses the `Resource` class in its implementation. But this module is concerned with the implementation of the `Name` class so this dependency is irrelevant to the implementer of `Name` in this context. The implementer of the `Name` class does not need to deal with potential changes to the implementation of the `SelectorUtils` class due to changes in the `Resource` class in-

terface. All the implementer of the `Name` class is concerned with regarding the `SelectorUtils` class is its interface. The implementation is encapsulated and can change without affecting the `Name` class.

In addition, this dependency is not missing from our design DSM. In module 349 of our decomposed ACNs, it does indeed show that the implementation of `SelectorUtils` depends on the interface of `Resource` because that module focuses on the implementation task for `SelectorUtils`. A big DSM composed from sub-DSMs will reveal this dependency between `Resource` and `SelectorUtils`.

In summary, our approach indeed aggregates exactly the set of dimensions needed for a function into a sub-ACN/sub-DSM, and has the potential to make explicit both indirect and implicit dependencies embodied in a UML class diagram. Our separation of interface and implementation design dimensions allows us to filter out superfluous dependencies that implementers of specific parts of a system do not need to be concerned with.

6. DISCUSSION

Since our `uml2acn` tool is still a prototype, its performance can still be improved. The current part of our code that takes the majority of the time is parsing of the Rational Rose input file. This process is time intensive because we are using a manually written parser based on our understanding of the Rational Rose file format. For future work, we will investigate available libraries or APIs for reading and parsing the Rational Rose file format to improve this performance. Another part of our tool that, where we can improve performance is the transformation section for generating the ACN. We are currently using the freely available, Saxon XSLT engine [4] for performing the transformation but the commercially available version of this engine claims better performance. This engine is used extensively in our tool, for converting UML inputs to the intermediate format and for converting the intermediate format to ACN project outputs.

In this paper, we only considered UML class diagrams, but not all the UML diagrams, such as sequence diagrams. Automatically transforming other UML diagrams into ACNs is part of our future work.

The UML diagrams only embody part of design decisions of the design space. There could be possible decisions that are not representable by any UML diagrams, such as power consumption or other environmental conditions [34]. As a result, the ACN model derived from UML models can not be complete. On the other hand, an ACN model is expandable. New design decisions and constraints can always be added.

The subsystem modules represent only part of the functions of the whole system, but not all of them. Take the maze game for example. One possibility is to just generate a plain maze game without any fancy features. Our decomposition method will not generate a subsystem for it because none of the decisions involved in this task is without a dependent: even the implementation of a plain room, `Room`, has other classes depending on it.

Our method can identify modules, each representing a task, but not features, because a feature may comprise a number of modules. How to identify features from modules is our future work.

Since the DSMs that Lattix generates miss indirect dependencies, a possible method to consider to improve those

DSMs would be to take the transitive closure. In fact in our previous work [21], we compared transitively closed source code DSMs against ACN generated DSMs, and found that the transitively closed DSMs contained redundant and false dependencies.

Baldwin and Clark define design rules as a stable design decision that dominates other decision [9], which can be viewed as decision variables that are visible to other variables, but cannot be influenced by subordinating variables. This implies a design rule hierarchy, which determines the visibility scope of each variable. We plan to explore the approach to automatically derive this hierarchy.

7. RELATED WORK

DSMs were created by Steward [32], developed by Epfinger [18], and have been applied to other many engineering disciplines [32, 18]. Baldwin and Clark provided insight into the connections between design structure and economic value in design using DSMs [9]. Sullivan et al. [34, 35] and Lopes et al. [25] have shown that DSM modeling is valuable for software design.

Since our approach transforms UML class diagram models into ACNs, all existing ACN-based techniques can be applied. One such technique is our work on using ACNs for testing of software design modularity [12, 11]. Using this technique, we can determine how well modularized the design is for accommodating future changes. Our design testing framework quantitatively measures intuitive design principles such as coupling and cohesion, open to extension but closed to modifications, and complexity using ACNs. Using ACNs and DSMs also allows for applying Baldwin and Clark's net option value analysis [9], as accomplished by Sullivan [36], Cai [14], and Lopes [25].

Another similar model transformation technique is UML2Alloy of Anastasakis et al. [6]. Anastasakis et al. transform UML and OCL into Alloy [22] but the goals of UML2Alloy and our approach are different. Our goal is to transform a UML model into an analytical decision model to enable modularity analysis while the goal of UML2Alloy is to perform property checking on UML models, such as consistency checking.

Finding inconsistencies in a UML model is another area that is being investigated by researchers [38, 33, 16]. Yeung et al. [38] applied formal methods such as CSP [20] and B [5] to formalize UML diagrams. Based on the formalization, Straeten et al. [33] made use of supporting tools to detect errors in the corresponding UML diagrams. However, most of these tools suffer from performance problems. Our divide-and-conquer approach has the potential to decompose a big UML model into smaller pieces and help with the performance issues in the area of consistency checking. Our experiments have not shown any significant performance problems but testing our approach on larger software systems is a future work.

Class diagram abstraction [15] is another related work. Contrary to our translation of the relationships in a UML class diagram into ACNs, the abstraction techniques concentrate on the ability to zoom out of diagrams so that engineers can investigate them where fewer details mask their interrelationships.

8. CONCLUSION

To address the problems that emerging analytical decision models, such as design structure matrix and augmented constraint network, present sharp learning curves for software designers who are not used to decision and logical modeling, and that the prevailing UML models are not designed to explicitly capture software decisions and modular structure, we presented an approach to reap the best both sides: automatically derive analytical decision models from UML class diagrams.

Our approach is to first formally define the dependency relation embodied by each UML class diagrams using logical expressions, and then automatically generate an ACN model from a UML class diagram. Based on the generated ACN model, we then decompose it into a number of smaller sub-ACNs, each capturing all the decisions needed for a component. After that, we generate DSMs from the sub-ACNs.

We have evaluated our approach using a real project's UML class diagram, Apache Ant. The UML class diagram was reverse engineered using Rational Rose from the source code. We compared the UML DSMs generated in our framework with the source code DSMs generated by Lattix. We observe that our framework can correctly decompose a large UML model into subsystem modules, each representing independent task assignment. Our approach reveals how decisions are shared among subsystems and indirect dependencies, which are not easily shown by traditional DSM models. Decomposing a large UML model reverse engineered from source code also has the potential to help the user to understand the system in a divide-and-conquer way.

9. REFERENCES

- [1] Dependency finder. <http://depfind.sourceforge.net/>.
- [2] Dia. <http://live.gnome.org/Dia>.
- [3] IBM Rational Rose. <http://www-306.ibm.com/software/awdtools/developer/rose/>.
- [4] Saxon XSLT processor. <http://saxon.sourceforge.net/>.
- [5] J. B. Abrial. *The B-Book*. Cambridge University Press, 1996.
- [6] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In *10th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 436–450, Sept. 2007.
- [7] Apache Software Foundation. Apache ant project. <http://ant.apache.org/index.html>.
- [8] S. Baase and A. V. Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison Wesley, 3rd edition, Nov. 1999.
- [9] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press, 2000.
- [10] Y. Cai. *Modularity in Design: Formal Modeling and Automated Analysis*. PhD thesis, University of Virginia, Aug. 2006.
- [11] Y. Cai and S. Huynh. An evolution model for software modularity assessment. In *1st ICSE Workshop on Assessment of Contemporary Modularization Techniques*, May 2007.
- [12] Y. Cai, S. Huynh, and T. Xie. A framework and tool supports for testing modularity of software design. In *22nd IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2007.

- [13] Y. Cai and K. Sullivan. Simon: A tool for logical design space modeling and analysis. In *20th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2005.
- [14] Y. Cai and K. Sullivan. Modularity analysis of logical design models. In *21st IEEE/ACM International Conference on Automated Software Engineering*, Sept. 2006.
- [15] A. Egyed. Automated abstraction of class diagrams. In *ACM Transactions on Software Engineering and Methodology*, volume 11, pages 449–491, 2002.
- [16] A. Egyed. Fixing inconsistencies in uml models. In *29th International Conference on Software Engineering*, May 2007.
- [17] A. Egyed, W. Shen, and K. Wang. Maintaining life perspectives during the refinement of uml class structures. In *8th International Conference on Fundamental Approaches to Software Engineering*, Apr. 2005.
- [18] S. D. Eppinger. Model-based approaches to managing concurrent engineering. *Journal of Engineering Design*, 2(4):283–290, 1991.
- [19] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Nov. 1994.
- [20] C. A. R. Hoare. *Communicating Sequential Process*. Prentice Hall, 1985.
- [21] S. Huynh, Y. Cai, Y. Song, and K. Sullivan. Automatic modularity conformance checking. In *Proceedings of the 30th International Conference on Software Engineering*, May 2008.
- [22] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering Methodology*, 11(2):256–290, 2002.
- [23] M. J. LaMantia, Y. Cai, A. D. MacCormack, and J. Rusnak. Analyzing the evolution of large-scale software systems using design structure matrices and design rule theory: Two exploratory cases. In *7th Working IEEE/IFIP International Conference on Software Architectures*, pages 83–92, Feb. 2008.
- [24] Lattix Inc. The lattix approach, 2004.
- [25] C. V. Lopes and S. K. Bajracharya. An analysis of modularity in aspect-oriented design. In *4th International Conference on Aspected-Oriented Software Development*, pages 15–26, Mar. 2005.
- [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*, 52(7), July 2006.
- [27] Object Management Group. Xml metadata interchange.
<http://www.omg.org/technology/documents/formal/xmi.htm>.
- [28] Object Management Group. Object constraint language specification.
<http://www.omg.org/technology/documents/formal/ocl.htm>, May 2006.
- [29] Object Management Group. Unified modeling language: Superstructure.
<http://www.omg.org/technology/documents/formal/uml.htm>, Feb. 2007.
- [30] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
- [31] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2005.
- [32] D. V. Steward. The design structure system: A method for managing the design of complex systems. *IEEE Transactions on Engineering Management*, 28(3):71–84, 1981.
- [33] R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between uml models. In *6th International Conference on the Unified Modelling Language: Modelling Languages and Applications*, LNCS 2863, pages 326–340, 2003.
- [34] K. Sullivan, Y. Cai, B. Hallen, and W. G. Griswold. The structure and value of modularity in design. *ACM SIGSOFT Software Engineering Notes*, 26(5):99–108, Sept. 2001.
- [35] K. Sullivan, W. Griswold, Y. Song, and Y. C. et al. Information hiding interfaces for aspect-oriented design. In *10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 166–175, Sept. 2005.
- [36] K. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 99–108, Sept. 2001.
- [37] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.
- [38] W. L. Yeung. Checking consistency between uml class and state models based on csp and b. In *Journal of Universal Computer Science*, volume 10, pages 1540–1558. Springer, 2004.