Cohesion, Coupling and Abstraction Level: Criteria for Capability Identification

Ramya Ravichandar and James D. Arthur
Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA
Email: {ramyar, arthur}@vt.edu

***Abstract*— Large-scale software-based systems warrant lengthy development cycles during which there is a constant evolution of user needs and technology specifications. It is imperative that in order to function satisfactorily the system components accommodate change. However, the traditional process of development advises baselining requirements, as opposed to architecting the system such that it supports evolution. We propose the construction of systems based on Capabilities to accommodate functional changes in a non-intrusive manner. Capabilities are functional abstractions designed to exhibit properties of stability — high cohesion, low coupling, and balanced abstraction levels — which promote an underlying change-tolerant framework. To measure these characteristics we explore two algorithms — *Synthesis* and *Decomposition* — based on polar approaches to problem solving. The synthesis algorithm measures stability properties of detailed rudimentary elements to determine which aggregates are Capabilities. In contrast, the decomposition algorithm identifies Capabilities by evaluating higher-level abstractions that represent various functionalities of the system to be developed. Upon empirical analysis (of a library system) we determine that neither approach is sufficient in isolation. Therefore, we formulate a computationally viable algorithm by reconciling specific aspects of measurement from the synthesis and decomposition approaches. In particular, it uses the cohesion and coupling measures as defined by the decomposition algorithm and the abstraction level as determined by the synthesis algorithm. We construct specific metrics to compute cohesion and coupling. In addition, we objectively define and illustrate the characterization of a *balanced* abstraction level. An experiment with a Course Evaluation system confirms the efficacy of Capabilities in increasing its change-tolerance.**

***Index Terms*— Capabilities Engineering, Change-tolerance, requirements engineering, software process, metrics, cohesion, coupling**

I. Introduction

Large-scale software-based systems need to be change-tolerant in order to permit system evolution. Various external factors such as market demands, customer needs' volatility, and others influence the expected system functionality over lengthy development cycles. We require the underlying framework of the system to be stable, that is, to accommodate these changes with minimal impact. To design such an architecture, we seek to define "entities" that exhibit specific properties which impart stability to the overall system. Specifically, we consider the three characteristics - high cohesion, low coupling and balanced abstraction levels. We term entities with these properties as *Capabilities*. The property of high cohesion helps localize the impact of change to within a Capability. Also, the ripple effect of change is less likely to propagate beyond the affected Capability because of its reduced coupling with neighboring Capabilities. An optimum level of abstraction assists in the understanding of the functionality in terms of its most relevant details [1]. Thus, we conjecture that if complex emergent systems are architected with Capabilities then they have an increased change-tolerance and can accommodate changes with minimal impact.

We explore two algorithms for measuring these characteristics and determining if an aggregate is a potential Capability. Note that we focus on needs analysis, a phase prior to requirements specification, because Capabilities are formulated from user needs.

- *Synthesis:* Based on a bottom-up approach to problem solving, this algorithm evaluates the system in terms of its most detailed elements. It coalesces low-level elements to form aggregates, and then iteratively computes the measures to determine if they are potential Capabilities.
- *Decomposition:* This is an algorithm based on the top-down approach. The system is visualized in terms of its highest level mission, which is then systematically decomposed into abstractions that are more detailed. Each of these abstractions are measured individually to determine Capabilities.

An empirical analysis of the two algorithms reveals that neither approach is sufficient by itself to determine the best set of Capabilities [2]. Therefore, we construct a composite algorithm to establish an equilibrium between the two polar approaches. This algorithm is based on a complementary approach that incorporates elements of cohesion and coupling from the decomposition strategy, and models abstraction from the synthesis perspective. The remainder of the paper is organized as follows: Section II discusses related work and outlines the overall process of engineering Capabilities. In Section III and Section IV, we compare and contrast the three measures that determine a Capability — cohesion, coupling, and abstraction level — from the synthesis and decomposition

This paper is based on "Reconciling Synthesis and Decomposition: A Composite Approach to Capability Identification," by R. Ravichandar, J. D. Arthur and R. P. Broadwater, which appeared in the Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), Tucson, USA, March 2007.

approaches, respectively. In Section V, we describe our composite algorithm that combines the two approaches, and in Section VI we discuss evaluation results of a real-world application (Course Evaluation System). Our conclusions are presented in Section VII.

II. Background and Related Work

A primary manifestation of evolution is in the form of requirements volatility [3], which is known to increase the defect density and affect project performance resulting in schedule and cost overruns [4] [5]. Traditional Requirements Engineering (RE) strives to manage volatility by baselining requirements. However, the dynamics of user needs and technology advancements during the extended development periods for complex emergent systems discourage fixed requirements. More recently, techniques such as the Performance based specifications [6] and Capability Based Acquisition (CBA) [7] are being utilized to mitigate change in large-scale systems.

We propose the construction of a stable system based on Capabilities. Capabilities incorporate evolutionary-friendly characteristics such as high cohesion, minimal coupling, and pragmatic levels of functional abstraction. Figure 1 illustrates the two major phases of the CE process. Phase I identifies sets of Capabilities based on

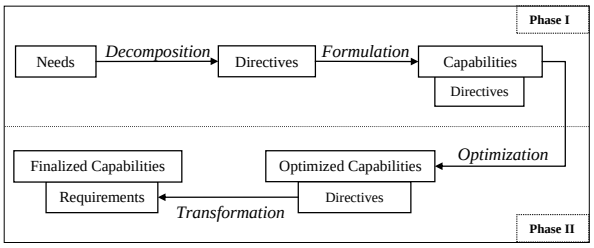

Figure 1. *Capabilities Engineering Process*

the values of cohesion, coupling and abstraction levels. Techniques of modularization suggest that high cohesion and low coupling are typical of stable units [8] [9]. Stability implies resistance to change; in the context of CE, we interpret stability as a property that accommodates change with minimal ripple effect. Ripple effect is the phenomenon of propagation of change from the affected source to its dependent constituents. Specifically, dependency links between aggregates behave as change propagation paths. The higher the number of links, the greater is the likelihood of ripple effect. Because coupling is a measure of interdependence between units [10] we choose low coupling as one indicator of stability for an aggregate. In contrast, cohesion — the other characteristic of a stable structure — depicts the "togetherness" of elements within an aggregate. A unit is said to be highly cohesive if each of its elements is directed towards achieving a single objective. As a general observation, as the cohesion of a unit increases, the coupling between the units decreases. However, this correlation is only approximate, and thereby, cannot be used to estimate the values of cohesion and coupling [9]. Therefore, we develop specific metrics to compute these values for potential Capabilities.

Phase II, a part of our ongoing research, further optimizes these initial sets of Capabilities to accommodate schedule constraints and technology advancements. In this paper, we focus on identifying Capabilities as outlined by Phase I.

In the following sections, we discuss the synthesis and the decomposition algorithms for computing the measures. We then explain the necessity for a composite algorithm that includes elements of cohesion, coupling, and abstraction from both of these approaches.

III. Synthesis

The objective of the synthesis algorithm is to formulate Capabilities — functional abstractions with high cohesion and low coupling — from user needs that are obtained during the process of elicitation [11]. Needs are affiliated with the problem domain and requirements are associated with the solution domain. Capabilities are computed after an analysis of user needs but prior to requirements specification. We envision that by doing so, Capabilities can bridge the chasm between the problem and the solution space, also described as the complexity gap [12]. It is recognized that this gap is responsible for information loss, misconstrued needs, and other detrimental effects that plague system development [13] [14].

The synthesis algorithm is based on a bottom-up approach, and hence, envisions a system in terms of its details. In particular, we consider system details that are defined at low levels of abstraction and are stated from a user's perspective. We term these details as *directives*. More specifically, a directive is a system specification that is described using the terminology in the problem domain. In contrast, a requirement is a system specification stated in the technical language of the solution domain. However, both a directive and a requirement share the commonality of being defined at a low level of abstraction.

Directives are a natural derivative of user needs. We use the directives as input to the synthesis algorithm for formulating Capabilities because they serve three main purposes. Firstly, directives strive to alleviate loss of domain knowledge, which has been identified as an important problem in RE [13]. They do so by describing system functionality in terms of the problem domain. This assists in capturing domain information. Secondly, directives are utilized to compute the cohesion and coupling values of potential Capabilities. Recall that optimal sets of Capabilities are to be determined from different functional abstractions. Capabilities are essentially system functionalities, and hence, are composed of one or more directives. Therefore, the cohesion and coupling measures of Capabilities are determined using directives. Lastly, directives facilitate the mapping to system requirements. Note that Capabilities only provide a high-level architecture based on system functionalities, and therefore,

requirement specifications are still necessary to direct system development. Directives are easily mapped to requirements because both entities are defined at similar levels of abstraction.

A. The Synthesis Algorithm

The synthesis algorithm aims to identify abstractions with maximum cohesion and minimum coupling, *i.e.* Capabilities. In particular, it strives to maximize functional cohesion, the most desirable cohesion among all other types of cohesion [15]. This objective of the synthesis algorithm is illustrated in Figure 2. If every element of a

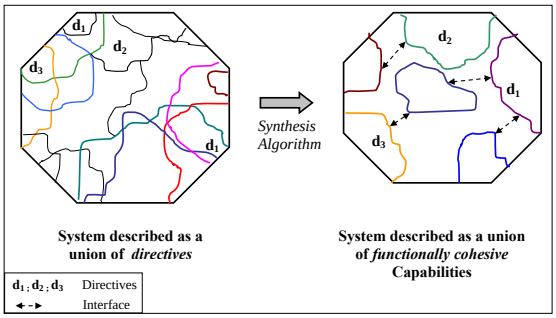

Figure 2. *Objective of Synthesis Algorithm*

unit is essential to the performance of a single function, then that unit is said to exhibit high functional cohesion [9]. Therefore, the first step of the algorithm enumerates functions that possess high functional cohesion. More specifically, we examine the significance of each directive in accomplishing various system functions. We use these significance values to compute the cohesion of a function in terms of all its participating directives. However, it is possible that the same function can be described at multiple levels of abstraction. We represent the functions using Venn diagrams to visually understand and resolve the abstraction level dilemma. The algorithm is explained in detail below:

Let $d_1, d_2, \ldots, d_n$, $n \in \mathbb{N}$, denoting directives derived from user needs, be the input to the synthesis algorithm. For each d_i perform the following steps to determine the Capabilities of a system:

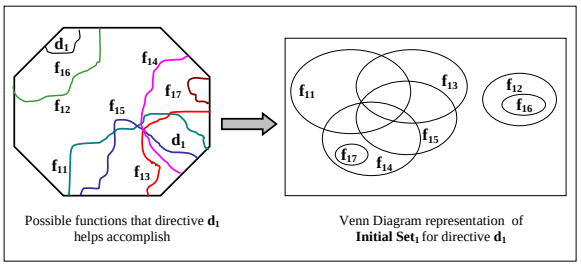

Figure 3. *Example Initial Set for directive* d_1

Identify all possible functions to which directive d_i *contributes:* The relevance of a directive in accomplishing a function is estimated using the impact categories shown in Table I. This classification is intended to assess the impact of risks on a project [16]. The failure to implement a directive is also a risk, and therefore, we use this classification to determine the significance of a directive in implementing a system functionality. We assign relevance values based on the perceived significance of each impact category; these values are normalized to the [0,1] scale. Formally, we enumerate the list of functions f_{im} that d_i

TABLE I.
Relevance Values

IMPACT	DESCRIPTION	RELEVANCE
Catastrophic	Task failure	1.00
Critical	Task success questionable	0.70
Marginal	Reduction in performance	0.30
Negligible	Non-operational impact	0.10

is associated with, as $InitialSet_i = \{f_{i1}, f_{i2}, \ldots, f_{im}\}$. For example, let d_1 help achieve functions $f_{1j}, j = 1, \ldots, 7$. A Venn diagram representation indicating the different abstraction levels of the functions of $InitialSet_1$ is shown in Figure 3.

Identify functionalities that subsume lower level functionalities: Expected system functionalities deduced from user needs can be stated at different levels of abstraction. Consequently, certain functions constituting $InitialSet_i$ may be inclusive of other functions in the same $InitialSet_i$. For example, in Figure 3, f_{12} is inclusive of f_{16}. We avoid considering functional abstractions that are partially or completely redundant as potential Capabilities by constructing $Subset_i \subseteq InitialSet_i$ where $Subset_i = \{f_{ix} | f_{ix} \supseteq f_{iy}, \forall f_{iy} \in InitialSet_i; 1 \leq x, y \leq m\}$. Note that the functions in $Subset_i$ are not encompassed by any other function in $InitialSet_i$. This implies that $Subset_i$ consists of functions defined at the *highest* level of abstraction among all other functions in $InitialSet_i$. Thus, as shown in Figure 4 for d_1, $Subset_1 = \{f_{1j}\}, j = 1, \ldots, 5$.

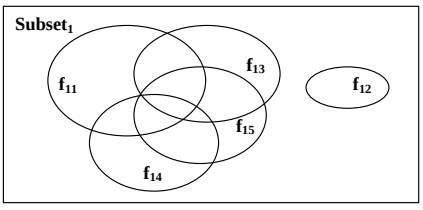

Figure 4. *Example Subset for directive* d_1

Coalesce functionalities that exhibit intersections: Although the aggregates in $Subset_i$ are not subset to any other aggregate, they can share common functionalities, which is an indicator of coupling. Recall that a Capability is a self-contained functional abstraction that is minimally coupled with other Capabilities. We strive to minimize the coupling between abstractions by reducing their dependencies. Specifically, in the synthesis algorithm we use the abstraction level as an instructive factor in constructing minimally coupled aggregates. The technique of coalescing allows us to contain the dependencies within

the boundaries of a higher abstraction. In particular, we identify aggregates that exhibit overlapping functionalities and aggregate them to form more decoupled abstractions. Hence, we create *aggregate subsets* $AG_{ij}(1 \leq i \leq n; 1 \leq j \leq m)$ from $Subset_i$ to contain aggregates with commonalities. Specifically,

$$AG_{ij} = \{f_{ix}, f_{iy} | f_{ix} \cap f_{iy} \neq \emptyset; 1 \leq x, y \leq m\}$$

such that $Subset_i = \underset{x}{\cup} f_{ix}, \forall f_{ix} \in AG_{ij}$;

We then abstract the entities of AG_{ij} to form higher level aggregates such that $AG_{ij} = \{F_{ij}\}$ where $F_{ij} = \{f_{ix} \cup f_{iy} \cup \ldots \cup f_{iz}\}; 1 \leq x, y, \ldots, z \leq m$. F_{ij} encompasses all aggregates in AG_{ij}. We term F_{ij} as **core functions**. Hence, we utilize core functions to derive and represent the functionality of system aggregates at a higher level of abstraction. For example, for directive d_1, in Figure 5, $AG_{11} = \{F_{11}\}$ where $F_{11} = \{\underset{j}{\cup} f_{1j}\}, j = 1, 3, 4, 5$ and $AG_{12} = \{F_{12}\}$ where $F_{12} = \{f_{12}\}$.

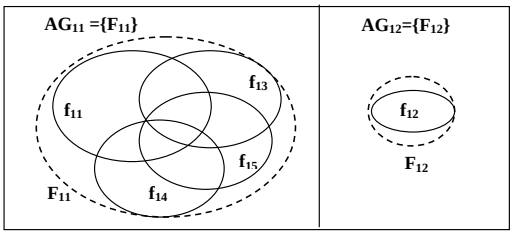

Figure 5. *Example Aggregate Subsets for directive* d_1

Define Core Function Sets: Let the core functions, F_{ij}, of all the aggregated subsets AG_{ij} related to directive d_i constitute the i^{th} **Core Function Set**, CFS_i, such that $CFS_i = \{F_{i1}, F_{i2}, \ldots, F_{ij}\}; 1 \leq i \leq n; 1 \leq j \leq m$. Hence, CFS_i comprises core functions that are functional abstractions initially defined at a more detailed level. These functional abstractions are potential Capabilities. Thus, as shown in Figure 6, $CFS_1 = \{F_{11}, F_{12}\}$.

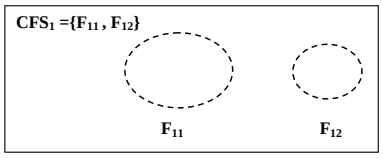

Figure 6. *Example CFS for directive* d_1

Thus, in this manner, the synthesis algorithm defines *a Core Function Set (CFS) for each directive in the system*. Specifically, each directive d_i has an associated CFS_i. The elements of a CFS are core functions, which are aggregates derived from a systematic process of synthesizing directives.

B. Analysis of Synthesis Approach

Recall that Capabilities are functional abstractions that exhibit high cohesion and low coupling. Therefore, we now measure the cohesion and coupling values and examine the abstraction level of each core function in order to determine the set of Capabilities.

Cohesion: For each directive the synthesis algorithm generates a CFS comprising core functions. The cohesion of a core function is computed as an average of the relevance values of each participating directive in achieving that function.

This implies that the list of directives associated with each core function in every CFS be enumerated; this necessitates substantial time and effort. Also, note that the core functions associated with different directives may be defined at various abstraction levels. Consequently, core functions may be subsets of one another resulting in redundant computations of relevance values. Furthermore, in our empirical analysis we observe that although the calculation of the average cohesion value is direct, the process of eliciting relevance values for each core function is highly cumbersome and notably subjective. These factors suggest that we need to explore alternate approaches for determining the cohesion of potential Capabilities.

Coupling: Units are said to be coupled if changes in a source unit affect one or more dependent entities. The only information available for computing the coupling between the elements of CFSs in the synthesis algorithm is the set of common directives shared by the core functions.

Experimental results show that determining coupling values based merely on this number is unrepresentative of the actual implementation. Furthermore, the synthesis approach fails to provide information about the strength of dependency between functions. Hence, we conclude that the synthesis algorithm is ill-equipped to facilitate the computation of coupling between potential Capabilities.

Abstraction Level: We know that each directive has an associated CFS whose elements are core functions. Empirical analysis reveals that at the abstraction level computed by the synthesis algorithm the core functions of a particular CFS do not share commonalities with other functions. Any further reduction in the abstraction level results in common intersections between aggregates.

The synthesis algorithm indicates that the abstraction level of a core function is determined by examining its links with other core functions. Therefore, one needs to consider the abstraction level, and the links between aggregates when formulating Capabilities.

In summary, the synthesis algorithm attempts to identify Capabilities from the detailed directives of complex emergent systems. Given the large magnitude of these systems, considerable effort is required to establish the CFSs. We note that, although the synthesis algorithm does provide insights regarding an ideal abstraction level of Capability, it is difficult to automate the computation of cohesion and coupling measures. Therefore, it seems impractical that the synthesis algorithm be utilized exclusively for identifying Capabilities. This mandates that we design a more objective algorithm that is far less dependent on user input. Hence, we examine an alternative solution — a decomposition algorithm based on the top-down approach — in the following section.

IV. DECOMPOSITION

The decomposition algorithm utilizes a graph-based representation of user needs, *viz.* a Function Decomposition (FD) graph, to formulate Capabilities. An FD graph represents functional abstractions of the system obtained by the systematic decomposition of user needs. A need at the highest level of abstraction is the mission of the system and is represented by the root. We use the top-down philosophy to decompose the mission into functions at various levels of abstraction. We claim that a decomposition of needs is equivalent to a decomposition of functions because a need essentially represents some functionality of the system. Formally, we define an FD graph $G = (V, E)$ as an acyclic directed graph where V is the vertex set and E is the edge set. V represents the system functionality: leaves represent directives, the root symbolizes the mission, and internal nodes indicate system functions at various abstraction levels. Similarly, the edge set E comprises edges that depict decomposition, intersection or refinement relationships among nodes. These edges are illustrated in Figure 7. An edge between a parent and its child nodes represents functional decomposition and implies that the functionality of the child is a proper subset of the parents functionality. Only internal (non-leaf) nodes with an outdegree of at least two can have valid decomposition edges with their children. The refinement edge is used when there is a need to express a node's functionality with more clarity, say, by furnishing additional details. A node with an outdegree of one symbolizes this type of relationship with its child node. To indicate the commonalities between functions defined at the same level of abstraction the intersection edge is used. Hence, a child node with an indegree greater than one represents a functionality common to all its parent nodes. The FD graph utilizes these definitions to provide a structured top-down representation of system functionality, and thereby, facilitates the decomposition algorithm to formulate Capabilities in terms of their cohesion, coupling, and abstraction values. We discuss the mechanics of the algorithm next.

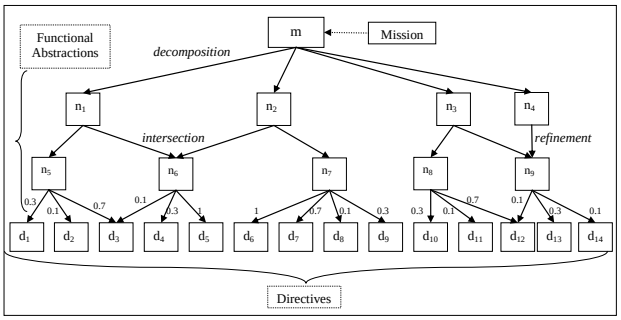

Figure 7. *Example FD Graph* $G = (V, E)$

A. The Decomposition Algorithm

The input to the decomposition algorithm is an FD graph, $G = (V, E)$ that represents the functionality of the system to be developed.

The cohesion and coupling values for each slice is computed using the measures described next. We also discuss the average abstraction level of nodes that possess high cohesion and low coupling values.

Cohesion: As in the synthesis algorithm, the cohesion of a node is computed as an average of the relevance values of the participating directive. The relevance values are assigned based on the values listed in Table I. However, we make a distinction between the parent and ancestor nodes of a directive. In order to reduce the need for user input, we elicit the relevance value of a directive only with respect to its parent node, whose cohesion is the arithmetic mean of the relevance values of its directives. Figure 7 illustrates relevance values of directives to their parents. The cohesion of an ancestor is computed as a weighted average of the size (number of associated directives) and cohesion of its non-leaf children. Specifically, the cohesion measure of an internal node n with $t > 1$ non-leaf children is:

$$Ch(n) = \frac{\sum_{i=1}^{t}(size(v_i).Ch(v_i))}{\sum_{i=1}^{t} size(v_i)}$$

such that $(n, v_i) \in E$, and where

$$size(n) = \begin{cases} \sum_{i=1}^{t} size(v_i) & (n, v_i) \in E; outdegree(v_i) > 0; \\ 1 & outdegree(n) = 0 \end{cases}$$

Coupling: To measure coupling we need information about dependencies between system functionalities. By the virtue of its construction, the structure of the FD graph represents the relations between different aggregates. In particular, we compute coupling between two nodes in terms of their directives. Two directives are said to be coupled if a change in one affects the other. We compute this effect as the probability that such a change occurs and propagates along the shortest path ($dist$) between them. Note that the coupling measure is asymmetric. Generalizing, the coupling measure between any two internal nodes $p, q \in V$, where $outdegree(p) > 1, outdegree(q) > 1$ and $D_p \cap D_q = \{\phi\}$ is:

$$Cp(p, q) = \frac{\sum_{d_i \in D_p} \sum_{d_j \in D_q} Cp(d_i, d_j)}{|D_p|.|D_q|}$$

where $Cp(d_i, d_j) = \dfrac{P(d_j)}{dist(d_i, d_j)}$ and $P(d_j) = \dfrac{1}{|D_q|}$.

$P(d_j)$ is the probability that directive d_j changes among all other directives associated with the node q.

Abstraction Level: The experimental results of the decomposition algorithm indicate that nodes which exhibit maximum cohesion and decreased coupling are also at higher abstraction levels. We know that abstraction level is related to size - the higher the level of a node, the greater the number of its associated directives. Thus, the decomposition algorithm identifies potential Capabilities

as sets of nodes that exhibit high cohesion and low coupling but are also of increased sizes. The latter, however, is undesirable from an implementation standpoint.

B. Analysis of Decomposition Approach

The decomposition algorithm provides an approach to automate the cohesion and coupling measures. Preliminary experimental results indicate that values computed using these metrics are indicative of desirable software engineering characteristics. In particular, we observe that on an average, potential Capabilities that have high cohesion values also exhibit low coupling with other nodes. However, the decomposition approach fails to provide nodes at an abstraction level that are optimal with respect to size. Therefore, we now explore a reconciliation between the synthesis and decomposition algorithms to determine Capabilities that are optimal with respect to the abstraction levels and the computations of cohesion and coupling.

V. Reconciliation

Our objective is to determine Capabilities based on the three measures of cohesion, coupling, and abstraction level. We construct a composite algorithm which is a derivative of both the synthesis and decomposition approaches.

Sections III and IV describe the synthesis and decomposition algorithms to formulate Capabilities. In particular, we observe that the computation of coupling and cohesion values using the *decomposition approach* can be easily automated. This is because the coupling measure is a function of distance of change propagation and probability of change, and therefore, is completely objective. Likewise, the cohesion measure, although less objective, is conveniently computed for all functional abstractions. In contrast, the excessive subjectivity of the synthesis approach presents little scope for automating the formulation of Capabilities in complex emergent systems. However, unlike the decomposition algorithm, the *synthesis approach* provides insights about the optimum abstraction level of a Capability. Hence, we construct a composite algorithm to formulate Capabilities such that it incorporates elements of cohesion and coupling from the decomposition algorithm and that of the abstraction level from the synthesis algorithm.

A. Composite Algorithm

Step 1: *Construct an FD graph*
As a first step, we need to establish the input to this algorithm. In the synthesis approach one considers all possible directives to determine functional abstractions. Although, an iterative process, the analysis of such a detailed representation challenges the limited processing capacity of the human mind [17]. On the other hand, the decomposition approach begins with the system mission (see Figure 7) that is easily comprehensible. Furthermore, it follows an intuitive process of decomposing it into its constituent functionalities.

Step 2: *Determine all slices*
Next, to determine the collection of nodes that are Capabilities we need to compute the cohesion, coupling, and abstraction level values of each node, in an FD graph. However, for a large system the number of such nodes will span several hundreds, thus, exponentially increasing the computational complexity. To reduce the complexity we examine only those combinations of nodes, which form a valid set of Capabilities. For example, we remove from consideration combinations that consist of nodes with overlapping directives (parent-child cases) because one is a subset of the other. We term valid combinations of nodes *slices*, and are defined by the following constraints:

1) *Complete Coverage of Directives:* A Capability is associated with a set of directives, which is eventually mapped to system requirement specifications (see Figure 1). We ensure that each directive is accounted for by some Capability, by enforcing the constraint of complete coverage given by $\bigcup_{i=1}^{m} D_i = L$, where
   - D_i is the set of leaves associated with the i^{th} node of slice S
   - $L = \{u \in V | outdegree(u) = 0\}$ denotes the set of all leaves of G
   - $m = |S|$
2) *Unique Membership for Directives:* To avoid implementing redundant functionality, we ensure the unique membership of directives by the constraint $\bigcap_{i=1}^{m} D_i = \{\phi\}$.
3) *System Mission/ Directive is not a Capability:* The root is the high level mission of the system, and a directive describes low level details. Neither can be considered a Capability as they are at extreme levels of abstraction. Hence, $\forall u \in S, indegree(u) \neq 0; outdegree(u) \neq 0$.

Step 3: *Compute Cohesion of each slice*
The cohesion value of a slice is the average of the cohesion of its constituent nodes. We use the cohesion measure defined in the decomposition approach (Section IV-A) to compute the cohesion of individual nodes. For example, in Figure 9 the cohesion of n_{50} is 0.75, n_{63} is 0.6125.

Step 4: *Compute Coupling of each slice*
Similarly, the coupling value of a slice is the average of the coupling of its constituent nodes. Specifically, this measure is a pair computation. For example, in Figure 8, if $S_1 = \{n_3, n_{55}, n_{41}\}$ then $Cp(S) = Avg(Cp(n_3, n_{55}), Cp(n_3, n_{41}), Cp(n_{55}, n_{41}))$. Again, we adopt the coupling metric defined in the decomposition approach (Section IV-A).

Step 4: *Select slices that exhibit high cohesion or low coupling*
Note that the permutations of S_1 have different coupling values, but the same cohesion value. This is because the

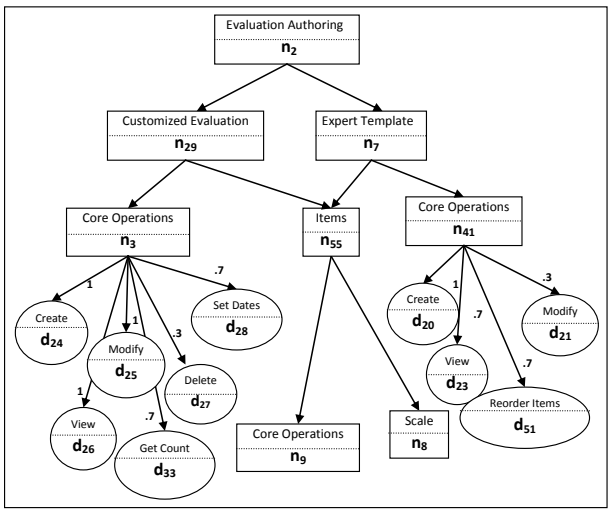

Figure 8. Course Evaluation System: *Common Functionality*

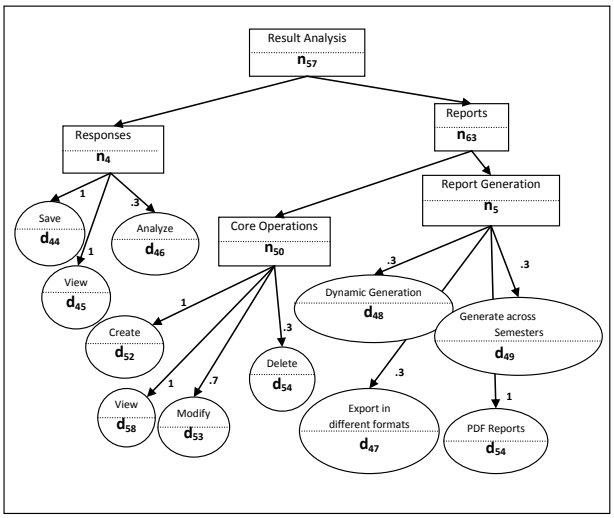

Figure 9. Course Evaluation System: *No Common Functionality*

coupling measure is asymmetric in nature; $Cp(n_3, n_{55}) \neq Cp(n_{55}, n_3)$. Thus, we rank the set of all slices and their permutations, in accordance to their cohesion and coupling values. We choose those slices that rank among the top 10, either in the high cohesion *or* the low coupling category. We then examine this set, selecting slices which have cohesion values above the overall cohesion average, and whose coupling values are below the overall coupling average of the top 10 slices.

Step 5: *Select the slice with balanced abstraction levels as the desired set of Capabilities*

We observe that as the abstraction level becomes lower, the node sizes decrease but the coupling values increase (size is the number of directives associated with a Capability). We strive to identify nodes of reduced sizes as Capabilities, in line with the principles of modularization. There are two possible scenarios when attempting to lower the abstraction level of a node, as illustrated by Figures 8 and 9. These graphs are subsets of the FD graph of a Course Evaluation system. Upon computing the cohesion and coupling values of the slices in this graph, we find that the slice $\{n_6, n_2, n_{57}\}$ exhibits the highest cohesion and lowest coupling value. However, does this slice have a *balanced* abstraction level? For this, we examine the effect of lowering the abstraction level of n_2 and n_{57}. (Note that n_6, is a node with only directives as children, and thus, we refrain from lowering its level)

Common Functionality: In Figure 8, assume that the size of n_2 is too large, and hence, we attempt to reduce its abstraction level to its children *viz.* `Customized Evaluation` (n_{29}) and `Expert Template` (n_7), which are of a relatively smaller size. However, we observe that these nodes share a common functionality in `Items` (n_{55}). This implies that one of the links, (n_{29}, n_{55}) or (n_7, n_{55}), needs to be broken in order to implement `Items` as a part of a parent Capability. Let (n_{29}, n_{55}) be broken, and `Items` be implemented as a part of `Expert Template`. Consequently, Capabilities n_{29} and n_7 are content coupled [18] because n_{29} may attempt to manipulate the `Items` part ingrained in n_7. Thus, lowering the abstraction level of `Evaluation Authoring` results in Capabilities of decreased sizes but increased coupling.

No Common Functionality: Now we consider the reduction of n_{57} to smaller-sized nodes, *i.e* the reduction of a single aggregate to smaller sized nodes that have no commonalities. Figure 9, illustrates that `Results Analysis` (n_{57}) reduces to `Responses` (n_4) and `Reports` (n_{63}). We observe that although there is a *marginal* increase in coupling, nodes n_4 and n_{63} are of smaller sizes when compared to n_{57}. Thus, we choose n_4 and n_{63} over their parent n_{57}. We are willing to accommodate this negligible increase in coupling for the convenience of increased modularity.

From the analysis above, we determine that slices containing nodes $\{n_4, n_{63}, n_2\}$ exhibit a balanced abstraction level. Because these nodes are also in the set of slices resulting from **Step 4**, they also exhibit high cohesion and low coupling. The slice with the highest cohesion and lowest coupling is desired set of Capabilities for the Course Evaluation System.

VI. VALIDATION

We empirically tested the hypothesis that a system design based on Capabilities is more change-tolerant than a design generated from the traditional RE approach. Specifically, we examine the impact of changing needs on the RE and CE-based designs of a Course Evaluation system. The original high-level design of this system is based on an RE approach, and is termed as RE-based design. The CE-based design is constructed based on Capabilities of the system. To determine the optimal

Capability set, we construct an FD graph. Certain parts of this graph are illustrated in Figures 8 and 9. Then, the algorithm described in Section V is executed on the entire FD graph. This results in a total of 1495 slices, from which the set of nodes exhibiting high cohesion, low coupling, and a balanced abstraction level is selected as the desired Capabilities of the Course Evaluation System. Finally, a CE-based design is constructed based on the chosen Capability set.

The RE and CE-based designs are now subject to different changes in needs. In particular, we examine the impact of six different needs' changes on the Course Evaluation system. An example of a need change is "*Need information about the handicapped accessible facilities for courses taught in Room XXX*". We propagate each change on the CE and RE-based designs and record the number of affected classes. We perform the Wilcoxon Signed Rank test, the non-parametric alternative to the paired t-test, which results in a P-value of 0.018. The P-value indicates the probability that the population medians of the number of affected classes in the RE and CE-based designs are different because of chance. The very small P-value compels us to reject the null hypothesis that the change-tolerance of the system is indifferent to either the CE or the RE approach. Thus, the alternate hypothesis that the number of impacted classes in the CE-based design is significantly lesser than that of the RE-based design is true. This result is in agreement with our research claim that the change-tolerance of a system improves with the use of a design based on Capabilities.

VII. Conclusion

The CE approach strives to construct complex emergent systems with the property of change-tolerance. For this, we use a composite algorithm to compute maximally cohesive, minimally coupled, and balanced functional abstractions as Capabilities. The cohesion and coupling measures of these basic building blocks are computed as in the decomposition algorithm and the abstraction level as defined by the synthesis algorithm. Note that the former algorithm is based on a top-down approach and the latter, on a bottom-up approach. Thus, the composite algorithm is a blend of the two polar approaches. Furthermore, empirical evaluation results from the experiment on Course Evaluation system are agreement with our research claim that the change-tolerance of a system improves with the use of a design based on Capabilities.

References

[1] D. L. Parnas, P. Clements, and D. Weiss, "The modular structure of complex systems," in *Proc. of 7th International Conference on Software Engineering*, Mar. 1984, pp. 408–417.

[2] R. Ravichandar, J. D. Arthur, and R. P. Broadwater, "Reconciling synthesis and decomposition: A composite approach to capability identification," in *14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, Tucson, Arizona, USA, 2007, pp. 287–296.

[3] S. D. P. Harker, K. D. Eason, and J. E. Dobson, "The change and evolution of requirements as a challenge to the practice of software engineering," in *Proceedings of IEEE International Symposium on Requirements Engineering*, Jan. 1993, pp. 266–272.

[4] Y. Malaiya and J. Denton, "Requirements volatility and defect density," in *10th International Symposium on Software Reliability Engineering*, Los Alamitos, California, Nov. 1999, pp. 285–294.

[5] D. Zowghi and N. Nurmuliani, "A study of the impact of requirements volatility on software project performance," in *9th Asia-Pacific Software Engineering Conference*, Gold Coast, Australia, Dec. 2002, p. 3.

[6] G. A. Tull, "Guide for the preparation and use of performance specifications," Department Of Defense," Headquarters U.S. Army Material Command, 1999.

[7] M. Montroll and E. McDermott. (2003) Capability-based acquisition in the missile defense agency: Independent research project. Storming Media. [Online]. Available: http://www.stormingmedia.us/82/8242/A824224.html

[8] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, pp. 1053–1058, 1972.

[9] E. Yourdon and L. Constantine, *Structured Design*. Englewood Cliffs, N.J: Prentice-Hall, 1979.

[10] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[11] J. A. Goguen and C. Linde, "Techniques for requirements elicitation," in *Proc. Int. Symp. Req. Engineering*, Los Alamitos, California, 1993, pp. 152–164.

[12] L. B. S. Racoon, "The complexity gap," *ACM SIGSOFT Software Engineering Notes*, vol. 20, pp. 37–44, 1995.

[13] P. Zave and M. Jackson, "Four dark corners of requirements engineering," *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 1–30, 1996.

[14] S. Lauesen and O. Vinter, "Preventing requirements defects: An experiment in process improvement," *Requirements Engineering*, vol. 6, pp. 37–50, 2001.

[15] J. M. Bieman and L. M. Ott, "Measuring functional cohesion," *IEEE Transactions in Software Engineering*, vol. 20, pp. 644–657, 1994.

[16] B. W. Boehm, *Software Risk Management*. New York: IEEE Computer Society Press, 1989.

[17] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *The Psychological Review*, vol. 63, pp. 81–97, 1956.

[18] M. Page-Jones, *The Practical Guide to Structured Systems Design*. New York, NY: YOURDON Press, 1980.

Ramya Ravichandar is currently a Ph.D. candidate at Virginia Polytechnic and State University, VA, USA. Her research interests include Large-Scale System Analysis, Requirements Engineering, Change-tolerant Systems, Software Measurement, and Impact Analysis. She is a member of the IEEE Computer Society.

James D. Arthur received his M.S. and Ph.D. degrees in computer science from Purdue University, and an MS degree in mathematics from the University of North Carolina at Greensboro. He is currently an Associate Professor of Computer Science at Virginia Polytechnic and State University, VA, USA His current research interests include Requirements Engineering, Verification and Validation, Network-Centric Architectures, Agile Practices and Software Quality Assessment. He is a member of the IEEE Computer Society and the ACM.