Constructing Change-Tolerant Systems Using Capability-Based Design

Dr. James D. Arthur and Ramya Ravichandar
Virginia Tech

Large-scale, complex emergent systems demand extended development life cycles. Unfortunately, the inescapable introduction of change over that period of time often has a detrimental impact on quality, and tends to increase associated development costs. In this article, we describe a capability-based approach to evolving change-tolerant systems; that is, systems whose entities (or capabilities) are highly cohesive, minimally coupled, and exhibit balanced levels of abstraction.

The widespread advancements in technology have encouraged the demand for large-scale problem solving. This has resulted in substantial investments of time, money, and other resources for complex engineering projects such as hybrid communication systems, state-of-the-art defense systems, and technologically advanced aeronautics systems. Unfortunately, the expenditures are belied by the failure of such systems. Plagued by evolving needs, volatile requirements, market vagaries, technology obsolescence, and other factors of change, a large number of projects are prematurely abandoned or are catastrophic failures [1, 2, 3]. The inherent complexity of these systems, compounded by their lengthy development cycles, is further exacerbated by utilizing development methods that are hostile to change. Moreover, this complexity often results in emergent behavior [4] that is unexpected. For example, the introduction of a new functionality in the system can result in unanticipated interactions with other existing components that can be detrimental to the overall system functionality.

More recently, techniques such as the performance-based specifications (PBSs) [5, 6] and capability-based acquisition (CBA) [7] are being utilized to mitigate change in large-scale systems. PBSs are requirements describing the outcome expected of a system from a high-level perspective. The less detailed nature of these specifications provides latitude for incorporating appropriate design techniques and new technologies. Similarly, CBA is expected to accommodate change and produce systems with relevant capability and current technology. It does so by both delaying requirement specifications in the software development cycle and allowing time for a promising technology to mature so that it can be integrated into the software system. However, the PBS and CBA approaches lack a scientific procedure for deriving system specifications from an initial set of user needs. Moreover, they neglect to define the level of abstraction at which a specification or a capability is to be described. Thus, these approaches propose solutions that are not definitive, comprehensive, or mature enough to accommodate change or benefit the development process for complex emergent systems.

In order to function acceptably over time, complex emergent systems must accommodate the effect of dynamic factors—such as varying stakeholder expectations, changing user needs, advancing technology, scheduling constraints, and market demands—during their lengthy development periods. We conjecture that these changes can be achieved with minimum impact if systems are architected using aggregates that are embedded with change-tolerant characteristics. Such aggregates are defined as *capabilities*.

> *"... changes can be achieved with minimum impact if systems are architected using aggregates that are embedded with change-tolerant characteristics."*

Capabilities are functional abstractions that populate the space between needs and requirements. As such, they (a) are more rigorously defined than user needs, (b) retain crucial context information inherent to the problem space, but at the same time (c) avoid solution specification commitments ascribed to requirements. Capabilities are constructed so that they exhibit high cohesion, low coupling, and balanced abstraction levels. 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 [8]. The balanced level of abstraction assists in understanding the embedded functionality in terms of its most relevant details [9]. Additionally, we observe that the abstraction level is related to the size of a capability; the higher the abstraction level, the greater the size of a capability [10]. From a software engineering perspective, abstractions with a smaller size are more desirable for implementation.

Capabilities are generated using a capabilities engineering (CE) process. Specifically, this approach employs a unique algorithm and a set of well-defined metric computations that exploit the principles of decomposition, abstraction, and modularity to identify functional aggregates (i.e., capabilities). Such capabilities embody the desirable software engineering attributes of high cohesion, low coupling and balanced abstraction levels. Change-tolerance is achieved through the embodiment of such attributes. The integration of the CE process with existing development paradigms, and the exploitation of enhanced traceability that accompanies it, are expected to reveal more effective methods for designing, building, and maintaining software for real-world systems. This results in a capability-based system specification that is change-tolerant, permitting a just-in-time specification of requirements, and an incremental development cycle that can span long periods of time.

The CE Process

The problem of changing requirements, especially in developing large complex systems, is well established [11]. Software development processes that are ill-equipped to accommodate change are pri-

marily afflicted with requirements volatility [12]. This phenomenon is known to increase the defect density and affect project performance resulting in schedule and cost overruns [2, 13]. 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 of complex emergent systems discourage fixed requirements.

Our approach, the CE process, builds change-tolerant systems on the basis of optimal sets of capabilities. Figure 1 illustrates the two major phases of the CE process. Phase I identifies sets of capabilities based on the values of cohesion, coupling, and abstraction levels. Phase II, a part of our ongoing research, further optimizes these initial sets of capabilities to accommodate schedule constraints and technology advancements. The CE process is discussed further in the following section.

The capabilities identification algorithm (also described in the following section) employs measures of cohesion, coupling, and abstraction to identify candidate sets of capabilities that necessarily and sufficiently embody the desired system functionality. Once identified, they can be further optimized to suit schedule and/or technology constraints; but because capabilities are formulated from user needs, our efforts required focus on needs analysis, a phase prior to requirements specification. At this point, we consider only the functional aspects of a system.

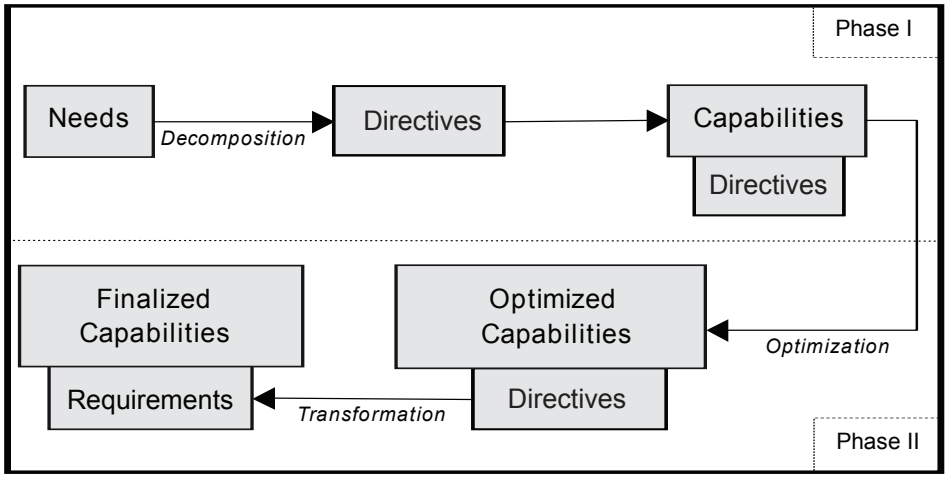


Figure 1: The *CE Process*

Computing Capabilities: The Algorithm

Capabilities are determined mathematically from a function decomposition (FD) graph (see Figure 2). This is an acyclic directed graph, implicitly derived from user needs, and represents system functionality at various levels of abstraction. The highest abstraction level, represented by the root node, connotes the mission of the system; the lowest levels of abstraction (i.e., the leaves), represent *directives*. Directives are low-level characteristics of the system formulated in the language of the problem domain. They differ from requirements in that requirements are formulated using language and terminology inherent to the more technically oriented solution domain. Thus, capabilities are identified after the elicitation of needs but prior to the formalization of technical system requirements. This unique spatial positioning permits the definition of capabilities to be independent of any particular development paradigm. We envision that by doing so, capabilities can bridge the chasm between the problem and the solution space, also described as the *complexity gap* [14]. It is recognized that this gap is responsible for information loss, misconstrued needs, and other detrimental effects that plague system development [15, 16].

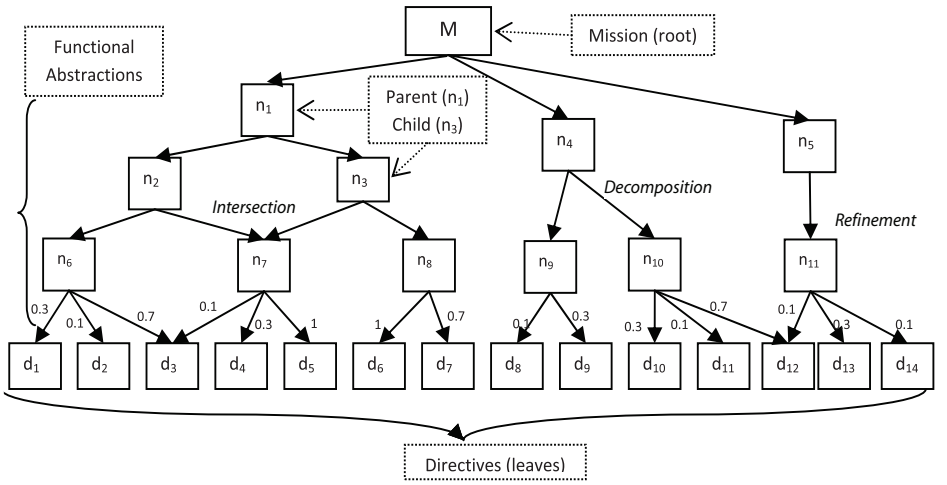
To identify capabilities, we need to examine all possible functional abstractions of a system represented in the FD graph. Intuitively, the algorithm for computing the desired set of capabilities is a five-step process that produces *slices* through the FD graph. We define a *slice* to be any subset of interior nodes of the FD graph such that their respective frontiers uniquely cover all directives. We select the slice containing the set of interior nodes that are maximally cohesive, minimally coupled, and exhibit balanced levels of abstraction. In effect, this slice contains the desired set of capabilities.

The following sub-sections outline the process for identifying the slice containing the desired set of capabilities.

Step 1: Constructing the Functional Decomposition Graph

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's 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 2. An edge between a *parent* and its *child* nodes represents functional decomposition and implies that the

Figure 2: *Example FD Graph G = (V,E)*



Impact of Directive Omission	Description of Impact on Associated Parent Task	Relevance of Directive
Catastrophic	Task failure	1.00
Critical	Task success questionable	0.70
Marginal	Reduction in performance	0.30
Negligible	Non-operational impact	0.10

Table 1: *Relevance Values*

functionality of the child is a proper subset of the parent's functionality. Only internal (non-leaf) nodes with an outdegree of at least 2 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 1 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 1 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, thereby facilitating the formulation of capabilities in terms of their cohesion, coupling, and abstraction values. We discuss those computational mechanics next.

Step 2: Computing Cohesion and Coupling Values

Only interior nodes are considered as capability candidates. For each interior node, its cohesion value is directly proportional to how important its children nodes are to achieving its defined functionality. Coupling, on the other hand, is a pair-wise relationship between two interior nodes and reflects the probability that a change in one node will have an impact on the other. The cohesion value for each node and the coupling value for each set of pair-wise nodes are computed using the FD graph *G,* and these measures are described next.

- **Cohesion.** The cohesion of a node is computed as an average of the relevance values of the participating directives. The relevance values are assigned based on the values listed in Table 1. 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. Figure 2 illustrates relevance values of directives to their parents.

  Assuming that each directive can be associated with a unique parent node (the validity of this assumption is established in [17]), then the cohesion for any node *n* can be computed (as shown in Figure 3).
- **Coupling.** To measure coupling, we need information about dependencies between system functionalities. By 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 between them. Note that the coupling measure is asymmetric.

  For two nodes *p* and *q,* the coupling between them is computed (as shown in Figure 4).

Figure 3: *Equation for Computing Node Cohesion*

(a) if node *n* has *only* directives as its children, then its cohesion is the arithmetic mean of the relevance values of the associated directives, i.e.:

$$Coh(n) = \frac{\sum_{\substack{\text{for each directive } i \\ \text{associated with node } n}} (\text{relevance value for directive } i)}{\text{total \# of directives associated with node } n}$$

(b) for all other nodes:

$$Coh(n) = \frac{\sum_{\substack{\text{for each immediate child } i \\ \text{associated with node } n}} [(\text{\# of directives associated with child } i) * (\text{cohesion of child } i)]}{\sum_{\substack{\text{for each immediate child } i \\ \text{associated with node } n}} (\text{\# of directives associated with child } i)}$$

Figure 4: *Equation for Computing Coupling Between Nodes*

$$Cp_{node}(p,q) = \frac{\sum_{\substack{\text{for each pairwise directive } i \text{ and } j \\ \text{associated with nodes } p \text{ and } q, \text{ respectively}}} (\text{coupling between directive i and directive j})}{\left(\begin{matrix}\text{total number of directives} \\ \text{associated with node } p\end{matrix}\right) * \left(\begin{matrix}\text{total number of directives} \\ \text{associated with node } q\end{matrix}\right)}$$

where the coupling between two directives *i* and *j* is computed as:

$$Cp_{directive}(i,j) = \frac{(\text{probability that directive j will change})}{\left(\begin{matrix}\text{length of shortest path} \\ \text{connecting directive i and directive } j\end{matrix}\right)}$$

and where the probability that directive *j* will change is computed as:

$$PrbChg(j) = \frac{1}{\left(\begin{matrix}\text{total \# of directives associated} \\ \text{with parent node of directive } j\end{matrix}\right)}$$

Step 3: Identifying the Candidate Set of Slices

Recall that slices are sets of nodes that necessarily and sufficiently cover all directives identified in the FD graph. Moreover, no slice contains the mission node (*M*). We compute all possible slices and then rank them. The first ranking (high to low) is based on the average cohesion of each slice's constituent nodes; the second ranking (low to high) is based on the average coupling of each slice's constituent nodes. The top 10 slices (an arbitrary count) common to both sets are then

chosen to form the *pruned* candidate set and represent those slices that have the highest average cohesion and lowest average coupling.

Step 4: Computing Balanced Abstraction Levels

In the next step, we individually examine each of the 10 slices with the objective of iteratively decomposing constituent nodes to achieve a balanced level of implementation abstraction. The decomposition process consists of replacing a parent node with its children nodes. We observe that as nodes are decomposed the abstraction level becomes lower—that is, the node sizes decrease but the coupling values increase (size is the number of directives associated with an interior node). We strive to identify nodes of reduced sizes in line with the principles of modularization, but only if the increase in coupling is acceptable. There are two possible scenarios when attempting to lower the abstraction level of a node: The replacement (children) nodes have lower-level common functionality, or they have no common functionality. Referring again to the FD Graph in Figure 2, suppose that one of the candidate slices is $\{n_1, n_4, n_5\}$.

- **Common Functionality.** Assume that the size of n_1 is too large, and hence, we attempt to reduce its abstraction level to its children, viz. n_2 and n_3, which are of a relatively smaller size. We observe, however, that these nodes share a common functionality represented by n_7. This implies that one of the links, (n_2, n_7) or (n_3, n_7), needs to be broken in order to implement n_7 as a part of a single-parent capability. Let (n_2, n_7) be broken, and n_7 be implemented as a part of n_3. Consequently, capabilities n_2 and n_3 are content-coupled [16] because n_2 may attempt to manipulate the n_7 part embodied in n_3. Thus, lowering the abstraction level of n_1 results in capabilities of decreased sizes, but with increased coupling.
- **No Common Functionality.** Now we consider the reduction of n_4 to smaller-sized nodes, n_9 and n_{10}. Note that the proposed reduction has no commonalities. We observe that there is a marginal increase in coupling, but that nodes n_9 and n_{10} are of smaller sizes when compared to n_4. Thus, we choose n_9 and n_{10} over their parent n_4. We are willing to accommodate this negligible increase in coupling for the convenience of increased modularity, a decision based, in part, on subjective evaluation.

Hence, we iteratively compute the appropriate abstraction level for each node in the set of slices identified in Step 3, and perform the appropriate decomposition substitutions. Because the nodes selected for abstraction balancing are in the set of slices resulting from Step 3, they also exhibit high cohesion and low coupling.

Step 5: Selecting the "Optimal" Set of Capabilities

As the final step, we re-compute the average coupling and cohesion values for each of the 10 slices. *The slice having the best balance between high cohesion and low coupling is selected as the set of capabilities for the system.*

A Validation of the Current Work

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. More specifically, we examined the impact of changing needs on the RE- and CE-based designs of a course evaluation system [17]. The original high-level design of this system is based on an RE approach and is termed RE-based design. The CE-based design was constructed using a capabilities approach for the system. To determine the optimal capability set, we constructed an FD graph and then applied the algorithm described earlier. This resulted in a total of 1,495 slices, from which the slice containing the set of nodes exhibiting the highest average cohesion, lowest average coupling, and a balanced abstraction level was selected as the desired capabilities of the course evaluation system. The CE-based design was constructed based on the chosen capability set.

The RE- and CE-based designs were then subjected to various changes in needs. In particular, we examined the impact of six different needs' changes on the course evaluation system. An example of a need change is, "The users need information about the handicapped-accessible facilities for courses taught in Room X." We propagated each change on the RE- and CE-based designs and recorded the number of affected classes. We performed 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 RE or the CE approach. Thus, the alternate hypothesis that the number of impacted classes in the CE-based design is significantly less 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.

Summary

The current and proposed work addresses several issues associated with the design, evolution, and emergent behavior of large-scale, real-world software systems. As stated in this article, CE provides a first-level architectural decomposition of the software system. Modularity and reasoned aggregation are cornerstones for identifying change-tolerant functional units. The underlying algorithm, employing metric-based computations, extends needs analysis to produce sets of capabilities enumerating multiple composition choices and, at the same time, indicates the advantages/disadvantages of selecting one set over the other. The use of capabilities also permits the delayed commitment of needs to requirements which, in turn, support the integration of new technology throughout the (extended) software development effort. Moreover, because capabilities are designed to be loosely coupled, they facilitate emergent behavior through the addition/deletion of functionality as new operational conditions and constraints evolve. Finally, we expect capabilities to support earlier architectural analysis, leading to the design of systems that better accommodate non-functional requirements like performance, security, and reliability.◆

References

1. Deal, D.W. "Beyond the Widget: Columbia Accident Lessons Affirmed." Air Space Power XVIII 2 (2004): 31-50.
2. Glass, R.L. Software Runaways: Monumental Software Disasters. Upper Saddle River, NJ: Prentice Hall, 1998.
3. Jazequel, J.M., and B. Meyer. "Design By Contract: The Lessons of Ariane." Computer 30 (1997): 129-130.
4. Heylighen, F. Self-Organization, Emergence, and the Architecture of Complexity. Proc. of the First European Conference on System Science. Paris, France, 1989: 23-32.
5. Tull, G.A. Guide for the Preparation and Use of Performance Specifications. Department of Defense (DoD) AMC Pamphlet. Alexandria, VA., 1999: 715-17.
6. DoD. Guidebook for Performance-Based Services Acquisition. DoD

Acquisition, Technology, and Logistics. 2002 <www.acq.osd.mil/dpap/Docs/pbsaguide010201.pdf>.

7. Montroll, M., and E. McDermott. "Capability-Based Acquisition in the Missile Defense Agency." Storming Media (2003) <www.stormingmedia.us/82/8242/A824224.html>.
8. Haney, F.M. Module Connection Analysis – A Tool for Scheduling Software Debugging Activities. Proc. of AFIPS Joint Computer Conference. Boston, MA., 1972: 173-179.
9. Parnas, D.L., P. Clements, and D. Weiss. The Modular Structure of Complex Systems. Proc. of the Seventh International Conference on Software Engineering. Orlando FL., 1984: 408-417.
10. Ravichandar, R., J.D. Arthur, and R.P. Broadwater. Reconciling Synthesis and Decomposition: A Composite Approach to Capability Identification. Proc. of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems. Tucson, AZ., 2007: 287-296.
11. Curtis, B., H. Krasner, and N. Iscoe. "A Field Study of the Software Design Process for Large Systems." Communications of the ACM 31 (1998): 1268-1287.
12. Harker, S.D.P., K.D. Eason, and J.E. Dobson. The Change and Evolution of Requirements as a Challenge to the Practice of Software Engineering. Proc. of the IEEE International Symposium on Requirements Engineering. San Diego, CA., 1993: 266-272.
13. Zowghi, D., and N. Nurmuliani. A Study of the Impact of Requirements Volatility on Software Project Performance. Proc. of the Ninth Asia-Pacific Software Engineering Conference. Gold Coast, Australia, Dec. 2002: 3.
14. Racoon, L.B.S. "The Complexity Gap." ACM SIGSOFT Software Engineering Notes. 1995.
15. Lauesen, S., and O. Vinter. "Preventing Requirements Defects: An Experiment in Process Improvement." Requirements Engineering 6 (2001): 37-50.
16. Zave, P., and M. Jackson. "Four Dark Corners of Requirements Engineering." ACM Transactions on Software Engineering and Methodology 6 (1996): 1-30.
17. Ravichandar, R., J.D. Arthur, S.A. Bohner, and D.P. Tegarden. "Introducing Change-Tolerance Through Capabilities-Based Design: An Empirical Analysis." Journal of Software Maintenance and Evolution (20.2) 2008: 135-170.

About the Authors

James D. Arthur, Ph.D., is an associate professor of computer science at Virginia Tech. His research interests include software engineering and requirements engineering as well as methods and methodologies supporting software quality assessment. Arthur has served as an advisor to the U.S. Navy Commonality Working Group, as chair of the Education Panel for National Software Council Workshop, and was guest editor for the *Annals of Software Engineering*. He has master's and doctorate degrees in computer science from Purdue University, and a master's degree in mathematics from the University of North Carolina at Greensboro.

Virginia Tech
Department of Computer Science
2050 Torgersen Hall
Blacksburg, VA 24060
Phone: (540) 231-7538
E-mail: arthur@vt.edu

Ramya Ravichandar is currently a doctoral candidate at Virginia Tech. Her research interests include large-scale system analysis, requirements engineering, change-tolerant systems, software measurement, and impact analysis. Ravichandar is a member of the Institute of Electrical and Electronics Engineers Computer Society.

Virginia Tech
Department of Computer Science
2050 Torgersen Hall
Blacksburg, VA 24060
Phone: (540) 231-7542
E-mail: ramyar@vt.edu