

# Adaptation Architectures

**Kevin Sullivan**

University of Virginia Department of Computer Science  
151 Engineer's Way, P.O. Box 400740, Charlottesville VA 22904-4740  
[sullivan@cs.virginia.edu](mailto:sullivan@cs.virginia.edu)

## I. Introduction

The complexity of emerging software-intensive systems, and the complexity and dynamism of the environments in which they are developed and operated, are growing to unmanageable levels. We now even foresee the emergence of *ultra-large-scale* (ULS) systems, which are project to be so complex that developing them will be unmanageable with the current scientific, engineering and managerial paradigm of largely centralized control and substantially top-down development [ULS]. Notwithstanding some important progress away from traditional methods, such complex systems will nevertheless still require significant advances beyond the current state of the art in design theory and methods. As the U.S. President's Council of Advisors on Science and Technology (PCAST) put it, "As software's complexity continues to rise, today's ... problems will become intractable unless fundamental breakthroughs are made in the science and technology of software design and development [PCAST]."

In this paper we focus on the need for fundamental advances in the particularly important discipline of software and software-intensive systems architecture. While recognizing that many important advances have been made, we see advantages in exploring a significant change in perspective to support *design for adaptation* of complex software-intensive *ecosystems*. In this paper we outline one promising direction and summarize evidence developed to date in support of ongoing work in this direction.

The rest of this paper is organized as follows. Section II presents a critique of the current formulation of software architecture: namely that it focuses on artifacts rather than on decisions. Section III provides a quick overview of the alternative formulation that my colleagues and I are working to develop: toward a decision-based ontology for software architecture. Section IV summarizes results of varying maturity from a number of studies that taken together suggest that the new perspective does have the potential to support a significant research and development effort. Section V concludes.

## II. The Current Ontology of Software Architecture

Many deep and useful advances have been made in the area of architecture over the last few decades. That said, we find that our current understanding of architecture is overly constrained by an ontology that does not capture the concerns most essential in the area of adaptive capacity; and, to the extent that it can capture them implicitly, is too narrow in its scope of concerns. We describe an emerging perspective on architecture that focuses not on software components and their behaviors, per se, but on the decisions that determine system configurations and, in particular, on constraints that organize the coupling and decoupling of such decisions into modules. We summarize evidence from a number of studies for the proposition that this new perspective has the potential to significantly advance our scientific understanding of, and eventually to lead to useful tools and methods for, the definition, design, development, deployment, operation and adaptation of the kinds of complex software-intensive systems that are now on the horizon or in development.

There are at least two crucial dimension in which the current paradigm in software architecture should be reconsidered. First, current methods are largely committed to abstract modeling of computational components, compositions, and behaviors. What needs to be adapted over time, however, are the decisions that we make about such things in light of decisions that others make about their needs and the state of the world. For example, designers make decisions about what data structures to use within a system in response to decisions that others outside a system make about the quantity of data to be represented and system performance characteristics.

We need an ontology, a science, an approach to modeling and analysis, that lets us express *decisions* in many such dimensions, options to change them, how decisions are coupled (e.g., how decisions about performance requirements are coupled to decisions on choices of data structures), the consequences of exercising options to change decisions, and other phenomena of this sort. A particularly important issue is modularity in system design, for that is a property that is essential to providing adaptive capacity. We need a theory of modularity that makes explicit the modular structure of a system understood as a partitioning of decisions into subsets within which decisions can be changed independently of other subsets; and we need a theory in which the prior decisions that produce such partitions are explicit.

Second, to the extent that current software-component-focused architectural methods represent decisions or options to change them, they represent decisions and options implicitly: more importantly only within in relatively narrow domain, namely the domain of the software itself. They do not adequately support modeling and analysis of the decisions made *around software and software-intensive systems* (e.g., by customers, by marketplaces, by technology vendors, by malicious attackers) or the ways in which such decisions made in the environment are coupled to (and demand adaptation of) decisions that are made within the system boundary.

### **A. Ontology: Component versus Decision Abstraction**

In this subsection and the one that follows we elaborate on the foregoing analysis. The basic problem we see is that we lack a science of architecture (and supporting technology, e.g., in modeling and analysis tools) adequate to support effective development of complex systems with the kinds of adaptive capacity that will be so important in the future. There are many key areas in which we still lack basic capabilities: how to formulate, express and analyze comprehensive requirements for adaptive capacity across a system; how to make economically rational decisions about, and how much and in what dimensions to invest in (e.g., how to value) adaptive capacity;

how to provide adaptive capacity in the right amounts and dimensions in design, operational, and runtime execution processes; how to exploit adaptive capacity optimally over time. At bottom we lack an adequate science and technology base to render provisioning and exploitation of adaptive capacity into a scientific, engineering, and managerial discipline.

Adaptive capacity is strongly linked to the degrees of freedom in a system left open when the architectural constraints of the system are stabilized and to some extent locked down. It is by adaptation in these free dimensions within the architectural constraints that the adaptation of a complex system to its environment occurs. Yet current architectural theories, concepts, notations, tools and methods do not adequately recognize, explicitly represent, or enable rigorous reasoning about such things as the dynamic forces outside a system under consideration that create needs for adaptation. Current approaches provide little by way of a unified framework for representing mechanisms embedded in organizations and systems for detecting these forces, representing the dimensions in which systems remain adaptable, abstract spaces within which, or trajectories by which adaptation can occur. Without the ability to account for such issues more explicitly and in analyzable form, we believe it will remain unnecessarily hard to design complex systems with appropriate levels and kinds of adaptive capacity.

Current notions and practices of architecture are primarily concerned with the abstract representation of computational components, how components are composed, constraints on components and their compositions, and analysis of behavioral properties of such components and compositions (e.g., schedulability in real-time systems). Architecture is largely understood as providing abstract descriptions of *system morphology* and *execution behavior*.

By contrast, a growing body of evidence points to the conclusion that a rather different conception and ontology are needed to support rigorous engineering of adaptive capacity. A promising alternative ontological basis for such a science and practice is rooted instead in the abstraction of the *decisions* that lead to given system configurations, and related concepts such as the uncertainties surrounding decisions, the options available to change decisions, the agents that are responsible for implementing changes, the tasks involved in making such changes, and the ways in which decisions and thus tasks and agents are coupled (or not). Current representations do not provide adequate techniques for representing such issues in explicit and analyzable forms.

## ***B. Scope: The Coupling of Decisions to the Environment***

Moreover, to the extent that decisions, options to change them, and related issues are represented in current forms of architectural description (albeit implicitly), the scope of such decisions is generally limited primarily to the realm of code in the abstract. What we need, by contrast, is a way explicitly to represent decisions across a much broader range of concerns.

Notations such as UML, for example, have proven to be valuable in modeling classes, objects, interactions, state machines, and other such abstract code and execution concerns. UML and similar notations do also provide means to represent issues, such as requirements expressed in the form of use cases, as well as specifications represented in the object constraint language. Moreover, such languages are often extensible and in principle can be forced to model anything. In practice, however, they are primarily directed as modeling programs. They have proven very valuable, but they are not particularly well suited to modeling adaptation-related issues in areas beyond programming code. Among other things, they provide little if any real support for modeling available options for adaptation over time or how the forces impinging on a system are linked to its adaptive capacity.

We believe that it is now important to extend our conception of architecture: both in the scope of concerns to be modeled, and in the ontology for modeling them. In terms of scope, we need to reconsider what are the *subject systems* for which we need to design architectures, and what in addition to the system itself do we need to model when developing an architecture. We believe that it's now worthwhile to consider larger *ecosystems* of software production and use as the subjects of architectural design for adaptation, and that we should model such ecosystems in the environmental contexts to which they must adapt.

These broader ecosystems include but extend well beyond code artifacts, to include software artifacts – not just code but artifacts spanning the lifecycle (from requirements to code to documentation to tests). These ecosystem also include the design team itself, the operators, and the users. Such ecosystems are also connected to factors in dynamic external environments. It is not just code that needs to adapt; it is the whole ecosystem. Moreover, as designers we have to consider where adaptive capacity is best placed: sometimes it is not in the structure of the code per se, but in some organization: for example the operators or even the users, or in the running autonomic system. We need to develop the means to devise adaptation architectures that span and account for all of these *levels*. This notion in our view is the meaning of the term, *design of all levels*, as it appeared and was intended in the ULS systems report [ULS].

### III. Toward a New Ontology of System Architecture

Designers often have flexibility in where to allocate adaptive capacity and associated adaptation mechanisms, including both sensing and response capabilities. It is for this reason that, in order to achieve adequate adaptive capacity in (and across) systems of the future, we need to design and analyze adaptation architectures for the broader ecosystems. Consider an example to make these ideas more concrete: What sort of architecture is needed to provide the adaptive capacity for a system to respond to a variety of security attacks?

There are in fact many places across the ecosystem to which the responsibility, authority and ability to adapt could be assigned. First, one could place such adaptive capacity within the running system itself. The resulting architecture would include autonomic mechanisms. This would be an appropriate structure if there were a need for a fast adaptive dynamics without costly and relatively slow human intervention by operators or developers.

Second, one could allocate the required adaptive capacity to the human operators of the system. People are often the most adaptive elements of complex software-intensive systems, and they can often adapt in ways that would at best be costly to implement in software. A decision to place adaptive capacity at this level could exploit human problem-solving ability while avoiding the need for complex autonomic software and while sparing expensive software developers the need to modify the software in response to every previously unrecognized attack.

Third, security attacks that exceed the adaptive capacity of the running system and the operational support infrastructure could in fact be allocated to the design team. Indeed, this is where responsibility for adaptation to unanticipated contingencies generally has to be placed today. The response generally would be relatively slow and costly because it would involve the production and deployment of new software into the fielded system, a manual activity requiring costly expertise.

An adaptation architecture might even assign adaptation to some attacks to some kind of countermeasures mechanism or team that would disable the attacker. This approach effectively renders what would otherwise be an *environment* variable (a decision made outside the system as

to whether the system is under attack or not) into a *system* variable: a decision controlled by the system itself.

This example also provides us a perspective on one particular kind of *evolution* of adaptation architectures for complex ecosystems over time. For example, responsibility for adaptation to some kinds of attack might *at first* have to be handled by the developers through changes to the software. As knowledge is gained, all or part of that adaptive capacity might be moved to the operator group. The developers would provide mechanisms that would give the operators the sensors needed to detect and the wherewithal to respond to attacks. When operator-based adaptation is too costly or slow, then adaptive capacity might be moved into the autonomic layers of the running system. We can even envision a day when, perhaps through mechanisms of digital evolution, autonomic mechanisms will be possible that will develop and deploy new code fully automatically, in essence moving adaptive capacity in the form of the ability to produce new code from the software development team directly into the running system.

What we need in general then is a science and engineering technologies and methods that support design for adaptation not just of code artifacts but of software-intensive ecosystems from end to end. What this means is that it is clearly no longer enough to design the code or even the broader set of software artifacts for ease of change [Parnas72], although it remains necessary and essential to do so. We need to *design for change* our larger ecosystems of software production, deployment, and use. We also need to do this in light of explicit models of the environments to which such ecosystems must adapt to remain profitable over time.

Among other things, we must shift our attention from the question, what *design decisions* [Parnas72] are likely to change, to the question what *environment decisions* are likely to change, how will we detect such changes, and what decisions can we change to adapt? Decisions about the environment variables to observe, choices about the kinds and levels of adaptive capacity to provide, and decisions where in the ecosystem to place adaptive capacity really are questions that need to be addressed by the ecosystem architect, not just by a software architect. That said, the software remains at the center of, and largely provides the mechanisms needed by the rest of, the ecosystem. What we thus need to develop is a discipline not just of *software architecture* but of *software-intensive ecosystem architecture* within which is embedded an integral discipline of software architecture, per se.

Again we by no means claim that attention can shift away from software. It is software that will generally provide the mechanisms that *enable* adaptation by other parts of a system, e.g., by providing operators with sensing capabilities and the mechanisms to reconfigure systems as conditions change. The architecture of the software per se will also continue largely to define the adaptive capabilities at the level of the software development team, because ultimately, it is the system code that will have to change to remain useful over time.

Rather, with respect to adaptive capacity, it is no longer *enough* to model software artifacts and behavior per se, to the substantial exclusion of the rest of the ecosystem. Second, the ontology of *software* architecture that we have developed to date is arguably not adequate for a science or engineering discipline of *adaptive ecosystem* architecture. Third, a growing body of evidence suggests that an ontology based on the modeling of *decisions* (very broadly construed), decision options, relationships among decisions, the structure of such relationships, and related issues might provide the basis for a better unified treatment of adaptive capacity across whole ecosystems, including but not limited to the software artifacts at the center of such ecosystems. The rest of this paper focuses on early evidence that increasingly supports these ideas.

## IV. Evidence for a Reconsideration of Architecture

In this section we summarize, in one place, some of the accumulating evidence that suggests that it might be profitable to continue to develop a notion of architecture centered on what we have called decisions—the values assigned to variables of many different kinds by agents both within and outside a system—and relationships that link these decisions.

### A. Baldwin and Clark

Our work is in a line extending back to general systems theory, through Herb Simon's work on architecture and complexity [Simon96], through Parnas's work on design for change [Parnas72], on to the present. A seminal *recent* work in this area is the pioneering study by Baldwin and Clark that they published in their book, *Design Rules: The Power of Modularity* [Baldwin00]. This work provided an important set of new ideas and perspectives for architectural modeling and analysis of complex computer-based systems. First, adaptive capacity is an essential ingredient for competitive economic advantage in the computer industry. Indeed Baldwin and Clark argue that appropriate modularity in design can provide decisive value to an organization, and that organizations and the whole computer industry have been shaped by the pursuit of this value.

Second, an account of adaptive capacity in the form of real options to explore the design space for a system in a decentralized manner and to make substitutions and other adaptive changes when better results are obtained can be developed in terms of abstract models of design decisions and dependences among them. Baldwin and Clark used *design structure matrices* (DSM's) as a notation to represent the principal decisions in a design architecture and the coupling structure thereupon.

Third, adaptive capacity is achieved to a significant extent by modularizing dependences among design decisions, following the information hiding strategies of Parnas in a general sense. Modularization increases adaptive capacity by decentralizing adaptation. It allows systems to evolve as *complex adaptive systems*. Adaptive capacity is distributed across and exercised within many individual modules in a system. Modularity in turn is obtained by formulating architectural decisions that are stable, shared, and define the assumptions that decision-makers within modules can make about other modules. These decisions are interfaces, broadly construed: what Baldwin and Clark called *design rules*

Fourth, the structure of dependences both within and across many levels of a system can be modeled in this style, including software specifications, code, tests, hardware, and so forth. Baldwin and Clark, for example, published DSMs showing the coupling of hardware, software and test-related decisions.

Fifth, economic models of the present value of adaptive capacity under uncertainty can be developed on the basis of such models. At the center of the Baldwin and Clark theory is the idea that under uncertainty adaptive capacity can be valued as a kind of option; and that what a good modular design provides is a portfolio of options that can be exercised independently to adapt a system in a piecemeal fashion without system-wide redesign efforts.

Sixth, the structure and dynamics of whole ecosystems of software-intensive system development, production, and use can be comprehended and accounted for by theories expressed in such terms. In particular, Baldwin and Clark seek to explain the evolution of the organization of the computer industry as a whole.

Sullivan and his colleagues, having independently reached some of the same conclusions as Baldwin and Clark (e.g., on options valuation of flexibility in software design), have in turn

adopted the ideas of Baldwin and Clark as a foundation from which to try to advance the state of the art in software and software-intensive system design. While Baldwin and Clark did not mean to advance software architectural design per se, but rather to answer certain questions about the structure of industry, Sullivan and his colleagues saw in their work a great deal of raw material that could be exploited to develop a practical new way to conceive of software architecture.

### **B. Sullivan, Griswold, Cai**

In perhaps the first work that attempted to leverage the insights of Baldwin and Clark to advance the field of software engineering, per se [Sullivan01], Sullivan, Griswold and Cai tested the ability of Baldwin and Clark's modeling and valuation approach to model and test the ideas in Parnas's seminal case study, his famous 1972 paper *On the Criteria for Decomposing Systems into Modules*. That paper linked the adaptive capacity of programs to the criteria used to design their modular architectures. In particular, he compared the ease with which two differently designed versions of a *Key Word in Context* (KWIC) program could be adapted to changes in certain important design decisions. He found that his *information hiding* criterion outperformed the *functional decomposition* criterion, in terms of ease of change and ease of understanding, that had largely dominated the software design community to that point. The key idea was that modules should hide and thereby decouple decisions that would have to be changed as system requirements evolved. Parnas advocated in particular the use of data abstraction to hide choices concerning data representations.

What Sullivan et al. found was that it was feasible to model the decisions, the decision options, and coupling structure on decisions, i.e., the modular architecture, in each of Parnas's two designs, not as an interconnected system of data abstractions—in the style that prevails in architectural description today—but in terms of explicit variables/decisions and the coupling of these decisions to each other. Concretely, Sullivan et al. used DSMs in the style of Baldwin and Clark to develop such models. Sullivan et al. also found that the application of Baldwin and Clark's option pricing model of the economic value of the adaptive capacity embedded in the two designs was consistent with the informal conclusions of Parnas. Under defensible, albeit necessarily fictional, assumptions, the information hiding architecture was indeed found to be more economically valuable.

Beyond these conclusions, this work made several novel contributions. Most importantly, it recognized that an important ingredient was largely missing from the accounts of both Baldwin and Clark and Parnas before them: explicit modeling of the *environment* variables outside of the system and their relationship to the *design* variables inside the system. Sullivan et al. showed that it was feasible and useful to do this within the same unified decision-based modeling framework (using DSMs). The idea of modeling environment variables and how they are coupled to system variables in developing an account of adaptive capacity was not new. It appears as early as the 1950s in the work of Ashby; but it was usefully revived here for use in modern work on software architecture.

We believe that modeling the environment (even if only in a highly abstracted form) is essential. It is ultimately pressure from the outside to which systems must adapt on the inside. Parnas's exhortation to identify *design* decisions that are likely to change and to make them the secrets of modules remains correct, but this formulation doesn't get to the *root* of the problem. One must identify the *environment* decisions that are subject to change and then link them to the decisions (broadly construed) within the system that will have to adapt to maintain the system in a state of fitness for purpose in the changing environment.

### **C. Griswold, Sullivan, et al., XPIs**

Sullivan et al. [Sullivan05] and Griswold et al. [Griswold06] leveraged the initial work, above, to attack an important set of questions about a promising but somewhat controversial new area of research in software design theory, in the area known as *aspect-oriented programming (AOP)*. This work was important in validating the proposition that this new ontology for architectural modeling was useful, because it tested the ideas not against a traditional kind of information hiding architecture but against what was considered to be a radical new form of modular design.

The premise of work in AOP is that there is a class of important program design decisions that are both likely to change and not subject to being modularized using traditional mechanisms of function and data abstraction. A simple if hackneyed example is an *execution logging policy*. Such a policy dictates that a log be maintained of certain program execution events, e.g., any program execution action that affects financial accounting functions. In cases where such actions are inherently scattered across an otherwise sensible traditional program modularization, the code to implement such a policy ends up scattered across the modules of the system. Against such a structure, the logging policy can be characterized as a crosscutting concern. When the code implementing such concerns is scattered across a program, it is costly and error-prone to write the software to implement the concern in the first place and to adapt the software when the concern (e.g., the logging policy) changes. The fundamental premise of AOP is that systems whose modular structures are designed using traditional abstraction mechanisms generally lack adaptive capacity in numerous such *crosscutting* dimensions.

The broad goal of research in aspect-oriented programming to devise programming mechanisms and strategies for using them that support modularization of crosscutting concerns. The work described here addressed several issues in this area. First, AOP was based largely on the introduction of new module constructs distinctly different from the class, namely the *aspect*. An aspect module in this sense represented both a crosscutting behavior and specified (typically in a declarative sublanguage) where in the rest of the system the crosscutting behavior needed to be invoked. For example, an aspect implementing a logging policy would specify (e.g., using a regular-expression matching over the code of the rest of the system) where logging had to occur, and it would provide code to implement logging at those places. In this way, the aspect would modularize the otherwise crosscutting concern.

Several element of aspect-oriented programming were controversial at the time. First, aspects generally compromised the enforcement of traditional abstraction boundaries. They had to do this in order to ensure that crosscutting behaviors would be invoked at the correct places in the execution of the rest of the system. For example, execution events subject to a logging policy are often not visible in the interfaces of traditional program modules but only in the operation of otherwise hidden implementations. Aspects have to be able to *see* these events and thus to see the otherwise hidden code that implements them.

Second, AOP required a new ontological category that many people found (and continue to find) somewhat mystifying: the *crosscutting concern*. A crosscutting concern in some sense is a design decision that isn't easily modularized using traditional programming mechanisms. The AOP field then provided a design strategy based on this extended ontology: Traditional concerns are to be implemented in traditional object-oriented style while crosscutting concerns are to be implemented by modules a new kind: aspects. The conceptual distinction between ordinary and crosscutting concerns struck some as undesirable, nor was it clear whether *crosscuttingness* was an innate property of a concern or simply a manifestation of the limited expressive power of a

given programming language in which one wished to express such a concern. Researchers today continue to quibble on this topic.

Third, a distinguishing characteristic of aspects was that they would have the ability to *reach inside* (to *advise*) other modules across a system to modify their programming code, and that this would be done on the basis of declarative descriptions of what code needed to be so modified. Furthermore, motivated by a desire to accommodate unanticipated change, many AOP researchers insisted on the *non-enforcement* traditional abstraction boundaries as a criterion that in part defined what it meant for a language to be an AOP language. That is, AOP languages should not permit code to be closed to subsequent advising (modification) by aspects, precisely because one cannot anticipate what aspects will need to *see* in the future. This idea was and is controversial because, of course, encapsulation of details is widely viewed as critical to the management of coupling complexity in complex systems. A researcher at a large software company once commented to the author, “if we permitted our libraries to be advised, we would never be able to change them again, because to do so would break the code of tens of thousands of products that had come to depend on programming details of our libraries.”

Fourth, a new design method was introduced to which the moniker *oblivious* was attached. The idea was that traditional, non-crosscutting concerns could be programmed in a standard (e.g., object-oriented) style with code that was, or with coders who were, *oblivious* to the downstream augmentation of their *base code*. One would then add aspect modules to handle the more complex and problematical crosscutting concerns. This idea was controversial for a number of reasons. E.g., some believed that AOP mechanisms were or could become powerful enough to enable essentially arbitrarily complex changes to be made without disturbing the code being changed, while others insisted that as a practical matter, in general, one would in fact have to prepare code to be *advised* by aspects.

All in all it was suggested that AOP embodied a new conception modularity in design. Sullivan et al. and Griswold et al. agreed that the issues addressed by AOP were real and needed to be addressed. However, the controversies over AOP were troubling, and it was clear that some of the tradeoffs involved in using AOP mechanisms and strategies needed to be clarified. In this context, Sullivan et al. and Griswold et al. made several contributions. First, they showed that an architectural description framework based on *decision abstraction* using DSMs could express the modular structure of aspect-oriented code using precisely the same notations used to model the object-oriented code of Parnas’s 1972 study on information hiding using data abstraction. At this level of description there was no longer a need to distinguish ordinary and crosscutting concerns.

The key to this work was to properly identify and model the decision that were expressed in code. In particular, some *specification-level* decisions were coupled directly to corresponding individual code-level variables or to modular subsets of variables. For example, the specification of a certain service in a particular system was coupled through a corresponding class interface to the class implementation. If the specification changed in certain ways, the only decisions within the system that would have to change would concern that implementation code. These were *traditional* concerns. Other specification variables, such as one representing choice of execution *logging policy*, were coupled to implementation-level decisions spread across the set of code-level modules. These were what had been considered crosscutting concerns. In a DSM model, however, they were just decision variables coupled in more or less complex ways to decision variables at another level of the system: namely implementation code. The decision on logging policy was coupled to implementation-level variables in many modules.

Second, Sullivan et al. and Griswold et al. clarified some of basic engineering tradeoffs in terms of modular structure and adaptive capacity involved in approaches to using AOP such as had been touted by researchers and popularizers alike, especially *oblivious* design. Using design structure matrices to model dependences among decisions, Sullivan et al. and Griswold et al. showed that while the oblivious approach can initially benefit the developers of the code to be advised by aspects, it can vastly complicate development of aspect code and also the subsequent maintenance of the whole system. It simplifies development of so-called *base* code because the base code developers can ignore some complex issues. However, the DSMs of oblivious designs published in these papers revealed in an easy-to-see graphical form the extensive dependences of aspect code on hidden details of base code implementation that using an oblivious strategies created. The downsides of the oblivious approach included serialization of development (one cannot develop aspects until base code is finished); difficulty in developing base code due to the lack of assured regularity in the structure of the base code; and extreme sensitivity of aspect code to subsequent changes in implementation details of base code.

Third, rather than simply arguing that AOP is therefore inherently flawed, as some have done, Sullivan et al. and Griswold et al. showed how a decision-oriented ontology of modularity in design, including Baldwin and Clark's notion of design rules, could be used to develop a novel form of interface, and a corresponding novel form of abstraction, with the potential to reconcile the use of aspect-oriented mechanisms with the need for information hiding modularity and the control of coupling complexity.

They call this new kind interface that sits between aspect and base code the *crosscut programming interface*, or *XPI*. An XPI imposes constraints not only on the behavior of base code but on such program elements as identifier names and call graphs: aspects use such cues in composing aspect and base code. Such an interface construct provides advantages in at least three major areas. First, it enables parallel development of base and aspect code, subsequent to agreement on the XPIs by which the two forms of code will interact. Second, it allows aspect and base code to evolve independently, subject to the constraints on such issues as identifier naming. Third, an XPI makes explicit a crosscutting abstraction that in earlier forms of AOP remained implicit in details of aspects modules. For example, such an XPI might make explicit the notion of a *loggable execution event*. Such events do occur in many places across a system, and in that sense are crosscutting. A *loggableEvent* XPI provides a unified view of this behavior, and ensures that both lexical and behavioral aspects of code are managed so that the integration of base and aspect code is manageable and has predictable results.

This work thus contributed several advances. First, it provided a relatively clear way to define what it means to be crosscutting concern. For a particular system design, a concern is crosscutting if it can be modeled by decision variable at one level (e.g., a degree of freedom in the specification), the value of which affects decision variables scattered across a partitioning of variables (a modularization) at another level (e.g., the code). A stronger definition would be that a concern is *crosscutting for a language or kind of language* if the mechanisms of the language are such that the concern is crosscutting for *any* practical partitioning in the language. The field of aspect-oriented software development is largely concerned with concerns that are crosscutting in the latter sense for the class of procedural and object-oriented languages.

Second, this work showed how a *generalized* notion of *information hiding* interfaces can be used to develop an information hiding interface construct for AOP. The work contributed to AOP, but perhaps more importantly, the ability to make this advance provided evidence for the utility of a decision-based ontology of software and software-intensive system architecture. In

essence, both traditional forms of abstraction and a new one that emerged out of a vibrant but still developing and somewhat controversial research area were lifted to a more abstract level of description, and at that level could be modeled, analyzed, compared, and better understood in a uniform framework.

The unification can further be clarified by making explicit the decision-based analogy between the *XPI* and the traditional *application programming interface*, or *API*. An *API* reflects a decision in a particular dimension (in which a range of similar decisions were and remain possible) about a shared agreement on the constraints that both implementers and clients of given data abstractions must observe. Such a decision defines an interface in particular when it suffices to decouple sets of decision variables on either side of the interface, representing decisions that client and data type implementers must make in producing or maintaining their respective programs. An interface in this sense ensures, or is at least meant to ensure, that implementations compose as required and expected. It also reduces the cognitive difficulty of understanding a complex system by making an important abstraction explicit: e.g., the essential properties of service client/provider interactions independent of implementation details. These abstractions also often reflect ontologically important categories in the application domain, as in object-oriented modeling of physical systems.

An *XPI* similarly reflects a decision in a particular dimension in the form of an agreement on constraints that the implementers of *base* and *aspect* code must respect. An *XPI* is also meant to decouple remaining decisions that the base and aspect code implementers make and change over time. It does so by stating explicit and potentially enforceable constraints on properties of code that aspect programming mechanisms make visible, including variable naming rules. Like an *API*, an *XPI* also makes an important abstraction explicit: not a data or service abstraction but a crosscutting behavior, e.g., the set of loggable program execution actions. An *XPI* is meant to ensure that base and aspect code will compose properly through program evolution, even though decisions about details are made and changed independently. Perhaps most interesting, this work exhibited a form of information hiding interface that is very different than the data type interfaces that have largely been equated with information hiding for decades. We take this as evidence that an architectural ontology based not on computational components (such as classes and objects and aspects) but on decision variables has significant potential merit.

#### ***D. Cai and Sullivan, ACN***

Informal modeling of decision, options, and decision dependence structures enabled the authors of the works described above to make some real advances. In particular, DSMs proved to be a practical, lightweight modeling notation for the systems and approaches studied. However, the DSM is a highly informal and highly incomplete representation. A DSM does not model the choices available in a given dimension, and the semantics of the dependence markings in DSMs are often ambiguous and incomplete, in the sense that they do not express how or why a given decision depends on another.

The dissertation work of Cai and related publications by Cai and Sullivan [Cai06] showed that it is possible in a constrained setting to formalize an informal concept of architecture based on decision variables. In particular, they showed that decision variables with finite ranges can be formalized as variables in finite-domain constraint networks. Relationships among such variables can be formalized as constraints. Additional structures required to capture the full content of DSMs (such as the ordering and clustering of variables) can be expressed formally in these terms. Furthermore, from such formal and declarative models, one can actually derive the

corresponding DSMs mechanically. One thus precisely characterizes the sense in which a given DSM *abstracts* from a more detailed model of relationships among decision variables.

This work made several contributions to design architecture in a decision-oriented ontology. First, Cai and Sullivan showed in a limited but formal setting one way to precisely answer the question, *what does it mean for one decision to depend on another?* This question arises whenever one has to decide whether or not to mark a given cell in a DSM. It turned out to be more subtle than anticipated. The reason is that one often has several ways to re-satisfy a set of constraints over a set of variables. If a change in decision variable *A* can be accommodated by a change in either *B* or *C*, for example, then does one consider *B* or *C* to depend on *A*? Neither? Both? Moreover, in general, not every change to *A* will require a corresponding change to either *B* or *C*. Different changes to *A* can require changes to different sets of other decisions.

Informally, one might say that *B* depends on *A* if a change in *A* requires a change in *B*. It turned out that such a definition really isn't good enough. For Cai and Sullivan's to develop an adequate notion of dependence, they had to make second advance. In particular, they needed to reason about change not at the level of decision variables but at the level of specific valuations of such variables. In particular, they introduced the notion of a *design automaton* (DA). A DA is a state machine representing change dynamics in a design or configuration state space. The states of a DA are consistent valuations of decision variables under a given set of constraints. These states are then connected by arcs, each representing a transition between design states driven by a specific change in *one variable* from one value to another, where a destination state differs from an originating state in the following *minimal* sense: the variable marked as changing takes its new value in the destination state, the destination state is consistent with prevailing constraints, and no other variable that differs between the destination and origin states can be restored to its original value without invalidating a constraint. In other words, destination states embody changes in a *minimally* sufficient number of decisions to restore consistency: with no gratuitous ripple effects.

The DA provided a basis for a precise (and sensible) definition for what it means for one decision *B* to depend on another decision *A*. Sullivan and Cai defined a decision *B* to depend on a decision *A* if there is some state in which a decision *a1* was made for *A* and some change in *A* from *a1* to *a2* for which some destination state has a new value for *B*. In other words, there is some initial state in which a decision was made for *A*, and from that state some change in *A* can be compensated for by a minimal set of changes to other variables including a change in *B*.

Cai and Sullivan thus provided a firmer basis for deciding precisely when to mark a cell in a DSM. Prior to this work, it was often hard to know. Debates would emerge when there were several ways to restore consistency; when only some changes to *A* might be compensated for by changes to *B*; or when *A* might affect *B*, and *B*, *C*. Should indirect coupling be represented? The DA resolved these issues. Among other things, destination states represent all of the changes that are needed to restore consistency, including indirect ripple effects.

The DA also provided a precise operational model of the change dynamics in a design space. It is in this sense that this work most clearly starts to contribute to a science of adaptive system design. A single transition represents the impact of a single change in design. A path through a DA represents an evolutionary trajectory through a design space.

The ACN also provided a formal basis for the automatic derivation of DSMs from declarative ACNs, with DAs as an intermediate, state-machine-based representation. The basic idea here is to build the DA and then apply the definition of dependence; one then derives the set of marks that need to be placed in a DSM. A mark in a DSM thus generally represents one or

more ways of compensating with a change in B for some possible change in A. A great deal of information is lost (abstracted) in the reduction from DA to DSM. The DA provides a complete and precise model of dependences. The DSM is a highly abstracted but much easier to grasp model. Cai and Sullivan developed ACN (decision-based) models of architectures for which DSMs had been previously developed and published, and they showed that several of these DSMs had significant errors that were traceable to the difficulties mentioned above.

Work to exploit the potential of such an explicit representation has really only begun. For example, annotating arcs in such a model with costs or probability distributions on costs (and perhaps benefits) of change could enable powerful new support for value-based decision-making. Cai and Sullivan did demonstrate that it was possible to reformulate Baldwin and Clark's work on options-based valuation of modularity in design on the basis of these declarative and precise representations of system architectures. The *Simon* tool of Cai and Sullivan [Cai06] made this connection by deriving DSMs from ACNs and by then applying Baldwin and Clark's models, which require counting of dependences in DSMs.

This work remain more theoretical than practical at this time, but it appears useful for providing insights into the nature of a decision-oriented ontology of design architecture and system adaptation. We used Jackson's Alloy tool to represent variables and constraints. It has limited expressive power for our purposes. Design options are generally hard to enumerate, and many design constraints are not expressible or solvable using Alloy or similar notations or tools. More seriously, the sets of decisions that have to be made in a given design process continually changes, and, as Baldwin and Clark among others have noted, the value of one decision can change the set of decisions that have to be made subsequently. All of these limitations notwithstanding, this laboratory-scale work does provide a base for better understanding decisions, decision options, and relationships among decisions as an ontology for architectural description.

### ***E. Song, Adaptation hiding***

We end our survey of recent work on decisions, options, constraints, design/configuration state spaces and evolutionary trajectories as an ontology for system architecture and adaptation with a brief introduction to emerging work of Song and Sullivan. In a nutshell, Song proposes that models of design spaces and change dynamics based on constructs such as ACNs and DAs can be employed to develop a theory and practice, not of *design-time* evolution, but of *runtime* adaptation in the configurations and states of self-adaptive (autonomic) systems. She has shown, for example [Song07] that a *runtime-adaptive* version of Parnas's KWIC system can be produced, in which sensors detect the onset of the changes that Parnas discussed (e.g., increase in the size of the input data set); and in which built-in adaptive capacity in configuration switching options are exploited to adapt at *runtime*. For example, following Parnas's original scenario, when the size of the input is detected as exceeding a certain threshold, the implementation of the *line storage* abstract data type is replaced while the program runs by an implementation that spills input data to a file on the disk.

This emerging work reveals several features and requirements in a decision-oriented formulation of architecture as described it so far. First, it provides a conceptual unification of design-time change and runtime change. Her systems show that in one sense the structure of decisions and dependences is the same as in Parnas's study, but that the adaptive capacity is moved from one level of a system—from the human design team with a relatively slow evolution dynamics to autonomic mechanisms with a fast dynamics.

There are added costs, of course in placing adaptive capacity in the running system. More up-front work has to be done to pre-equip the autonomic system with built-in contingencies, and the adaptive capacity to switch among available options. In the design-team-based strategy, by contrast, the availability of options to develop new code are recognized but generally not exploited until needed. The bottom line, however, is that this work exhibits a unification of architectural modeling for modular adaptation across the design-time/run-time boundary.

This work does suggest the need to augment our modeling framework with the notions of (1) the *agents* that are responsible for adaptation, (2) the mechanisms needed for *sensing* the need for adaptation, and (3) the *tasks* required to effect adaptation. Baldwin and Clark introduced the *task structure matrix* to represent task interdependences parallel to *design structure matrices*. Here we make explicit the notion of *agents* to whom the tasks of changing decisions (variables) are assigned. Our example now includes members of the design team or autonomic mechanisms as agents, and could just as easily include system operators. Explicitly modeling agents will be important to understanding the costs and the dynamics of adaptation in complex systems, and limits on adaptive capacity as a function of agent-resource utilization. We believe (but leave to future work) that ecosystem architects of the future will explicitly model sensing and response mechanisms in adaptation architectures across levels of a system (design team, operator team, autonomic infrastructure, etc).

## V. Conclusion

A defining characteristic of ultra-large-scale systems of the future is that it will not be possible to get their requirements, specifications, implementations, configurations, or other crucial aspects right from the outset. Rather, such systems will have to discover and adapt over time—in many dimensions, at many time scales (from very slow to very fast), by many different mechanisms. One of the most crucial properties with which designers can imbue such systems will be the *capacity to adapt*. Today, however, we lack a science of design adequate to meet the need for principled and effective approaches to design for adaptation.

We have argued that adaptive capacity will spring largely from appropriate architectural decisions, but that our understanding of architecture today is perhaps too narrowly focused on program structure and execution behavior in the abstract. We have also argued that the ontology of architecture today, rooted as it is in the abstract description of program structure and behavior, is arguably inadequate to the needs of a more comprehensive approach to devising architectures for complex software-intensive ecosystems. What needs to be designed for adaptation today includes but now extends considerably beyond software artifacts and behaviors. We now need to be able to provide adaptive capacity across many levels of *ecosystems* of software-intensive system production and use.

In this paper we have presented an alternative conception and ontology of architecture, based on notions of decision variables (both within and outside of a system boundary), coupling of decisions, the structure of such coupling relations, and associated notions of agent, task and mechanisms supporting adaptation. We have presented in one place a still small but growing body of evidence that suggests that continued work on such a science and technology of design for adaptive capacity of complex software-intensive ecosystems is worthwhile.

A successful science provides precise, abstract, testable, and ultimately validated models of important classes of phenomena. The models of a successful science express crucial properties of the subject phenomena, enable one to predict characteristics of phenomena based on models,

and, in engineering, have prescriptive value. The subject phenomena of a science of design include design artifacts and processes, and their properties expressed in abstract forms subject to analysis and symbolic manipulation. Such models ideally capture essential properties of the subject phenomena, allow one to predict how such phenomena will unfold, and can guide one in shaping and harnessing such phenomena for useful ends. We hope the reader finds in this paper some hope for continued progress toward a useful science of design of software-intensive ecosystems in particular with respect to the crucial phenomena of adaptive capacity and dynamics.

## Acknowledgements

This work was supported by grants from the National Science Foundation, including grants CCF-0613840, CCF-0438898, and CCF-0429786.

## References

- [Baldwin00] Carliss Baldwin and Kim Clark, *Design Rules: The Power of Modularity*, MIT Press, 2001.
- [Cai06] Yuanfang Cai and Kevin Sullivan, Modularity Analysis of Logical Design Models, Proceedings of 21th IEEE/ACM International Conference on Automated Software Engineering, Tokyo, JAPAN, September 18-22, 2006.
- [Griswold06] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, Hriday Rajan, Modular Software Design with Crosscutting Interfaces, IEEE Software, January 2006, pp. 51—60.
- [Parnas72] D. L. Parnas, On the Criteria for Decomposing Systems into Modules, Communications of the ACM, Volume 15, Issue 12, Pages: 1053-1058, Year: 1972.
- [PCAST] President’s Council of Advisors on Science and Technology, Executive Office of the President of the United States of America, Federal Networking and Information Technology R&D (NITRD) Program Review, August 2007.
- [Simon96] Herb Simon, *The Science of the Artificial*, 3<sup>rd</sup> Edition, MIT Press, 1996.
- [ULS] Peter Feiler, Richard P. Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Linda Northrop, Douglas Schmidt, Kevin Sullivan, Kurt Wallnau, *Ultra-Large-Scale Systems, The Software Challenge of the Future*, Software Engineering Institute, 2006.
- [Song07] Yuanyuan Song, Adaptation Hiding Modularity for Self-Adaptive Systems, Companion to the proceedings of the 29th International Conference on Software Engineering, pp. 87-88, 2007.

[Sullivan01] Kevin Sullivan, William G. Griswold, Yuanfang Cai, The Structure and Value of Modularity in Software Design, *ACM SIGSOFT Symposium on the Foundations of Software Engineering joint with the European Software Engineering Conference*, pp. 99—108, 2001.

[Sullivan05] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, Hridesh Rajan, Information Hiding Interfaces for Aspect-Oriented Design, *ACM SIGSOFT Symposium on the Foundations of Software Engineering joint with the European Software Engineering Conference*, pp. 166—175, 2005.