An Abstract Architectural Model for Composition, Analysis and Evaluation

Sutirtha Bhattacharya, Dewayne E. Perry
Empirical Software Engineering Lab (ESEL)
ECE, The University of Texas at Austin
Austin, TX 78712
sutirtha.bhattacharya@intel.com
perry@ece.utexas.edu

Abstract

Perry and Wolf [2] introduced a general model for software architecture. Since then a number of architecture descriptions languages (ADLs) as well as a variety of architecture definitions have been introduced. None of these languages or definitions has provided an explicit model sufficient to support the needs of architecture composition, analysis and evaluation. In this paper, we introduce such a model, an abstract model sufficiently rich to support the above mentioned needs of. We explain and illustrate this model, provide an architectural example, and outline its usefulness.

Keywords

Software Architecture, Architectural Model, Composition, Analysis, Evaluation, and Reuse

1. Introduction

Laying the foundations of software development as an engineering discipline continues to challenge both researchers and practitioners. We often look to other disciplines for inspiration and "aspire to emulate their enviably well established repertoires of theoretical foundations and practical disciplines" [1]. What makes software so different? Why has software not lived up to the promise of standardization and reuse that would make building new software much easier? Is it the immaturity of the field, or is it the complexity of a discipline that spans a wide variety of application domains?

It is fair to say that the software domain cannot be comprehensively bounded in even by the four kinds of Denial identified by Jackson [1]. It may unfortunately be true that, we software engineers have not been able to clearly define our domain of influence between the *World* and the *Machine*. We have reached out for both with an equal intensity and in the process blurred the foundations upon which standards might evolve. But off course, we can take comfort in the saying that "great undertakings involve great risks" and it is unlikely anybody would challenge the assertion that it indeed is a great undertaking and has been a fascinating journey. It is with the help of software that we have peered into planet Mars and have predicted weather patterns around the globe.

Doug McIlroy's prediction in 1968 that mass-produced software components would end the so-called "software crises" has not materialized as yet. However, research and industrial experience over the years has led to the recognition that component-based software systems do provide substantial software engineering benefits.

When a system is made up of multiple modules or components, it is not hard to infer that there needs to be a framework in which these modules or components exist and operate. This overall framework is popularly known as the software architecture. A software architecture has been defined by Perry and Wolf as a triple of *elements*, *form* and *rationale* [2]. Since then, processing, data and connecting elements have been conflated into components and connectors which make up the *elements* of a system. The *form* is a set of weighted properties and defines the relationships between components and connectors, while the *rationale* is a set of justifications for the choice of elements and formal aspects of the software. While this definition of software architecture has been widely accepted for many years now, there has been little work done to define an architectural specification model based on this definition on which rules of composition can be applied to build complex systems out of components.

In this paper we propose an approach for modeling the *elements* and for specifying the *form* of an architecture to facilitate reasoning about compositions and for evaluating software architectures early in the development cycle. In section 2 we discuss the need for separation of concerns. The approach used for the modeling is described in section 3. Section 4 discusses the proposed model while section 5 provides an example software architecture that illustrates the abstract model and outlines its usefulness. Section 7 concludes the paper.

2. Separation of Concerns

Divide and Conquer has been widely acknowledged as a fundamental strategy in software engineering and computer science. We see it in sorting algorithms, it appears in multiplication of polynomials. In fact it is the seed idea that has spearheaded progress in operating

systems and programming languages. The question is, how relevant is to architectural design?

In this context it is important to discuss the Shanley principle that was highlighted as a rule for efficient design by Arnoul de Marneffe [3], who in turn was quoted by Knuth [4] in his paper on Structured programming and then by Michael Jackson in his ICSE17 keynote talk [1]. The idea behind the Shanley principle is that one part can perform multiple functions. This has been wonderfully explained by Jackson as "the architecture of the world has been designed with the fullest possible application of the Shanley Principle". In software, the statement that any software system has multiple stakeholders has no novelty in it and is indeed trite – which essentially means that the same software satisfies different functions for the different stakeholders involved. If it did not, we wouldn't need to worry about multiple stakeholders of the system. We do not believe that the Shanley principle is in contradiction with separation of concerns and that it effortlessly steps from Jackson's, *World* (i.e. problem space from which we derive our requirements) to his *Machine* (i.e. the solution space from which we create our system that satisfies these requirements). Separation of concerns is important when we build the 'machine' for managing complexity of the interrelationships in an 'intransigently informal world', but when a solution is actually deployed, the 'world' or the deployment environment may give the implementation different functions, which are often beyond the control of the creators of the solution. A word processor of today not only helps create electronic documents, it also helps ensure that the document has correct grammar and that the document is readable with proper formatting.

Since our essential goal is to reason about architectures, we deal with components at an abstraction level that is meaningful for component composition. Quantum Mechanics has turned Newtonian physics upside down but even then Newtonian physics does a reasonable job at explaining natural phenomena of objects bigger than sub-atomic particles.

It should thus come as no surprise that we hinge our abstract architectural model on a not-so-novel idea of separation of concerns. Our architectural model is supported by the three key constructs of architectural elements, architectural composition and architectural regions. The architectural elements serve to capture the *elements* of the architecture i.e. the components and connectors. For each architectural element we capture the service specifications, dependency specifications and the general constraints. The general constraints are categorized into functional and non-functional constraints. Together with the service and dependency specifications, the functional constraints captured as part of general constraints identify the requirements of the 'world' that the architectural element solves i.e. the "What", while the non-functional constraints capture the system requirements that need to be satisfied for delivering the 'machine' – i.e., the "How". The architecture composition and architectural region constructs are intended to capture the *form* of the architecture. These two constructs focus on capturing information that is relevant for performing compositional analysis – their purpose being quite different from capturing the needs or functionality of individual components

3. Approach for Specifications

Our primary goal is to create an abstract model of software architecture components 1) to support reasoning about component composition and 2) to provide a basis for constraint based architecture evaluation. An important secondary goal is to support the reasoning about component substitution (i.e. component reuse and component evolution). We expect the structure of an abstract model that satisfies our primary goals to satisfy our secondary goal as well. In this section we discuss the approach for the specifications.

3.1 Specifying Software Architectures

Software Architectures are generally thought of in one of two ways: as prescriptions or as descriptions. There are good reasons for both approaches and the need for each is largely dependent on its use. The differences are as follows: an *architectural prescription* defines the important constraints on the architecture – i.e. it defines the important, but not necessarily all, components and connectors, their critical properties (though again, not necessarily all of them), and the critical relationship and interactions among the components of that architecture. What is prescribed is necessary; what is not mentioned is allowed as needed in completing the remaining design at both the architectural and the lower levels of design. An *architectural description* on the other hand defines the complete architecture; what is not described is not allowed. The former is usually under-constrained, while the latter is precisely constrained (though it may often be over constrained). The former is usually described with constraints while the latter requires a more descriptive (and often simpler) architectural language.

We use a prescriptive approach for this research as the constraints provide an extremely useful tie between the system drivers and the architectural design, and provide a form of self-documenting rationale. Besides, given that an iterative development model is fast becoming the norm rather than the exception in the industry, it seems that building a descriptive architectural model would not be possible till the very last iteration, by when most of the key architectural and design decisions would already have been made.

3.2 Degree of Formalism in the Specifications

Within the context of *architectural prescriptions*, the specification technique used for this research was an important consideration. We needed a computable representation that was flexible. It was desirable that the representation was expressive while not compromising on the kind of information that could be modeled.

Informal methods are very flexible in form and therefore not computable. Leaving interpretation to humans inhibits reusability, as the semantics are not tight. For this research we need to communicate specifications in a way, which would enhance reuse and architectural evaluation rather than inhibit effective communication of the features/constraints. Hence informal methods were found unsuitable for this effort.

Formal methods have not been used for this research primarily because architecture specification has been envisioned to be an iterative process, relying on refinement from multiple iterations (component capabilities evolve and/or requirements specifications evolve). The representation we use needs to accommodate both evolution and iteration. Such change is not allowed in a formal approach as the semantics are strict and constraining them becomes a problem due to the iterative nature of specification.

Hence, we rely on semi-formal techniques for specification of the software architecture. Semi-formal methods (i) enable the accommodation of changes to architectural specifications (ii) provide enough expressiveness to support effective identification and evaluation of architectures and components in an architectural context (iii) are easy to understand and communicate (iv) does not require special training and is widely used.

4. Model for Software Architecture

Our proposal for an architectural model is consistent with the initial Perry and Wolf definition of software architecture. We propose three abstract constructs as the basis for our analysis:

- *Arch-element* -- An *arch-element* can be either a *component* or a *connector* (while their structure for purposes of modeling and analysis is identical, they have distinct logical purposes – ie, connectors represent interactions among components). This construct represents basic elements in the architecture.
- *Arch-composition* – An *arch-composition* represents the sub-architectural structure of an *arch-element.* As such it represents the substructure of an *arch-element* and must satisfy the interface constraints of that element. The rules of compositional completeness govern not only the support of the arch-element interface, but the internal interdependencies as well.
- *Arch-region* – An *arch-region* is an arbitrary set of arch-elements or arch-compositions and can overlap, contain or be contained in other *arch-regions.* An arch-region provides a constraint scoping mechanism. As such it represents a collection of *arch-elements* to which a set of constraints apply.

Before we delve into the specifics of the model, we take a slight detour here to explore the main issues with component composition so that we comprehend the requirements for component composition (to the extent possible) into our architectural model. David Garlan identified the main issue to be, what he called, architectural mismatches [17] and he highlighted several implications of this mismatch: excessive code size, poor performance, need to modify external packages used during the integration (or system composition), the need to re-invent existing functionality and an error prone build and construction process. The causes for these architectural mismatches identified by Garlan were inappropriate assumptions about the nature of the components and the connectors (i.e. our architectural elements), assumptions about the global architecture structure and the construction process.

It is obvious that system integration is an inherently complex process and there are no silver bullets for the problem. However there is a lot that can be done to facilitate this difficult process. We propose to use the *rationale* in our architectural model to document the assumptions about the components, the connectors and the global architecture structure so that the information is available to the system integrator for making optimal decisions. Besides, the *form* in our model will provide insight into the global architecture structure which could potentially provide guidelines to component developers. The non-functional aspects specified in our model would also capture information that would be useful during system composition.

For the overall organization of the architecture, we introduce the notion of an *architectural region*. Essentially it represents a collection of architectural elements to which a set of constraints apply. The concept of *regions* facilitates the specification of targeted rules for a sub-architecture. These rules could be compositional rules such as architectural styles or design patterns, as well as domain specific implicit constraints. They help localize constraints and make system instantiation easier, as they can potentially help promote a loose form of packaging of a set of components. Regions influence the *form* of an architecture and will be elaborated further in section 4.2.

In the next two sub-sections we discuss the models for the different elements of our architecture prescriptions.

4.1 The Elements: Components and Connectors

A software architecture specification is partitioned into several architectural elements. These architectural elements are driven by functional partitioning and also introduce the notion of object orientation which helps identify the implementation classes later during development. The elements of an architecture are the data, processing and connecting elements that have a physical existence and deliver some services that are either functional or non functional in nature. In this preliminary model we have not differentiated data, processing and connecting elements but conflated them all into arch-elements. The reason for this is that while they are logically distinct, it is not clear that they are at all structurally distinct. Data elements, of course, are clearly structurally distinct from processing and connecting elements. If we find that there are sufficient data elements independent of processing and connecting elements, we may separate them out as a separate component.

There is one issue however that may require structural differences: multiple connecting connectors. Connectors have been usually thought of as point to point mechanisms that provide the abstractions for communication interactions. However, that is not their only use. They may be used as coordinators and mediators as well. For example, one could imagine a very complex connector that serves as a coordinator of fault handling mechanism and instead of just one to one connectors, there are obvious uses for many to one (multiple clients, one server), one to many (broadcast), and many to many (cooperating components negotiating or reaching consensus) connectors, either with a fixed set of connections or an open-ended set of them. This is an important research issue that will need to be solved to complete our architecture model. And of course, connectors may be the subjects of architectural composition just as processing and data elements are.

The abstract model captures architectural elements as

arch-element =
(name, {service specifications },{dependency specifications}, {general constraints })

As mentioned previously an arch-element is qualified by the service specifications, the dependency specifications and the general constraints. The service specifications essentially capture the interface information using which other arch-elements can integrate and leverage the capabilities provided by the arch-element being specified. The dependency specifications help capture the 'needs' of an arch-element i.e. services that a given arch-element depends on. The general constraints capture all the functional and non-functional constraint that the arch-element needs to satisfy.

A service specification has a name, a set of input, output and general constraints associated with that service. Input and output constraints may define the information itself or constraints on that information that is needed or provided by the specified service. Example I/O constraints might include things like *sorted lists of faculty descriptions*, etc (of course in a semi-formal notation). The service specification construct is shown below.

service specification =
(name,{input constraints },{output constraints }, {general service constraints })

We separate out the dependency specifications from service specifications even though dependencies are basically the same except they are usually not named. These dependency specifications must be satisfied by the service specifications of the supporting architectural elements. This separates the formal service interface constraints from an arch-element's dependency interface constraints. The representation of the dependency-specification is shown below.

dependency specification =
({input constraints }, {output constraints }, {general dependency constraints })

The Input Constraints for the Service and Dependency specifications include the Input Data, Input Event and the Pre-Conditions constraint, while the Output constraints include the Output Data, Output Events and the Post-Conditions constraints. The Input and Output Data constraints capture the Input Data required for the execution of the service and the Output Data generated by the service. The Input and Output Events capture the Input Events that trigger the execution of the service and the Output Events that are generated by the service execution. The Pre-Conditions Constraints capture the set of conditions (as captured by the arch-element state) that need to be satisfied for the service to begin execution while the Post-Conditions Constraints capture the arch-element's state that should be satisfied upon execution of the service. It is to be noted that the pre-condition and post-condition constraints help validate that the service execution for the arch-element began when the desired set of conditions were satisfied and that it delivered the desired results.

General constraints can be classified into functional and non-functional constraints, and may represent obligations, placement constraints, etc. We have lumped a number of different kinds of constraints under the name general constraints. These may be functional constraints indicating the kinds of functionality for a component or dependency or these may be non-functional constraints, such as performance, fault tolerance, etc. They may also

be topological constraints indicating placement in a distributed system. Obligations entailed by using a particular architectural element may also be represented. The construct below shows general constraints

general constraints =
({functional constraints}, {non-functional constraints)

As part of general constraints, the functional constraints are intended to lump together different kinds of constraints that are associated with the delivery of end user functional requirements. As the data managed by an arch-element is fundamental to the kinds of services that it supports, we capture the data associated with an arch-element using the attribute constraints. Behavioral constraints ensure that the arch-element specifications comprehend the various states associated with the arch-element. It is common experience that architectural mismatches often happen when integration is done just by considering the API and not the implementation logic of the associated methods. The functional constraints construct is shown below.

functional-constraints =
({attribute constraints }, {behavioral constraints })

The non-functional constraints are captured in terms of the Quality Attribute Constraints and the Deployment Constraints. The Quality Attribute Constraints specifies the constraints on the quality attributes for the architectural element. These constraints on the quality attributes are over and above the arch-element's services, dependencies and the functional constraints. It is important to capture these constraints as part of the architectural specification because it has often been seen that systems need to be re-designed not because of any deficiency in supported functionality, but because they fail to satisfy requirements associated with certain quality attributes such as reliability, availability and performance. Thus explicit knowledge of these constraints would help in avoiding unacceptable system configurations. The Deployment Constraints on the other hand capture an architectural element's deployment related constraints such as installation requirements, platform dependencies etc. The non-functional constraints construct is shown below.

non-functional constraints =
({quality attribute constraints },{deployment constraints })

Since the details associated with the specification of arch-element is quite elaborate, we summarize the above with the help of a diagram. Figure 1 summarizes all the concepts discussed so far.

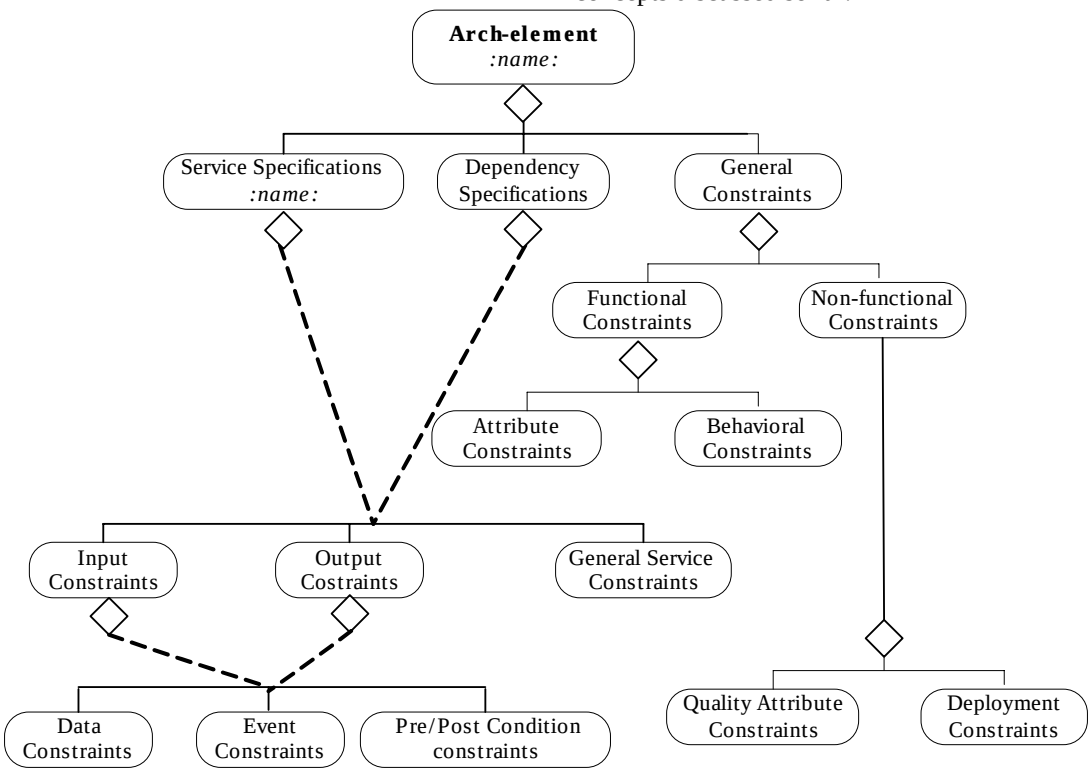

Figure 1: Architectural Element Specification

In the rest of this section we elaborate the details associated with some of the additional constructs mentioned previously.

The Attribute Constraints capture the data supported by the arch-element. An individual attribute constraint is qualified by its name, the data elements associated with it and any additional constraints that may be applicable. Information about the data elements are captured in the data element specification while general attribute constraints capture additional constraints on the data element or the attribute. As an example, the data entity 'Address' which is captured as an attribute may be further qualified by the associated data elements such as street name, city, zip code and country

attribute constraints =
(name, { data element specifications },{ general attribute constraints })

The Behavioral Constraints capture the behavioral aspects of an architectural element and is modeled using a state chart representation. The dynamic behavior of a component is modeled by the following quintuple and is termed as a behavioral unit which essentially represents a "unit of behavior".

Behavioral unit =
(state, trigger, guard, effects, target)

Figure 2 below identifies each of the above in a state chart diagram.

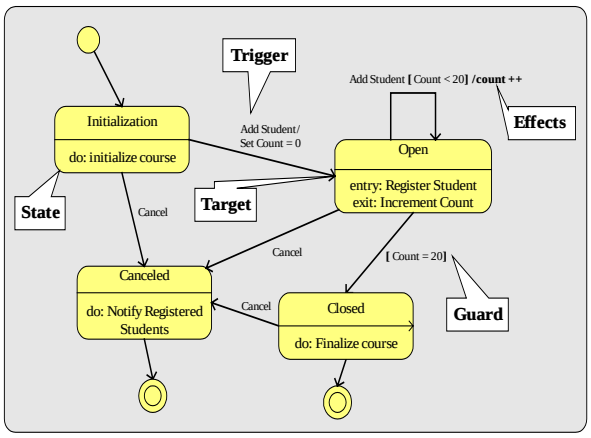

Figure 2: Behavioral Representation for Architectural Elements

The Quality Attribute Constraints specifies the constraints on the quality attributes for the arch-element. These constraints on the quality attributes are over and above the system's capabilities, services and behavior captured in the model. It is important to capture these constraints as a part of the specifications because it has often been seen that systems need to be re-designed because it fails to satisfy certain quality attributes. Hence explicit knowledge of a component's constraints would help in avoiding unacceptable system configurations.

Figure 3 below demonstrates a Quality Attributes Constraints. The Quality Attribute Constraints are composed of the Runtime Constraints and the Static Constraints. The Runtime Constraints captures the constraints of the arch-element that are relevant/observable during the execution of the element. On the contrary, the Static Constraints captures the constraints on the quality attributes of the arch-element that are not affected by the runtime characteristics. Obviously these constraints are optional for an arch-element as all of these together may not make sense in different contexts.

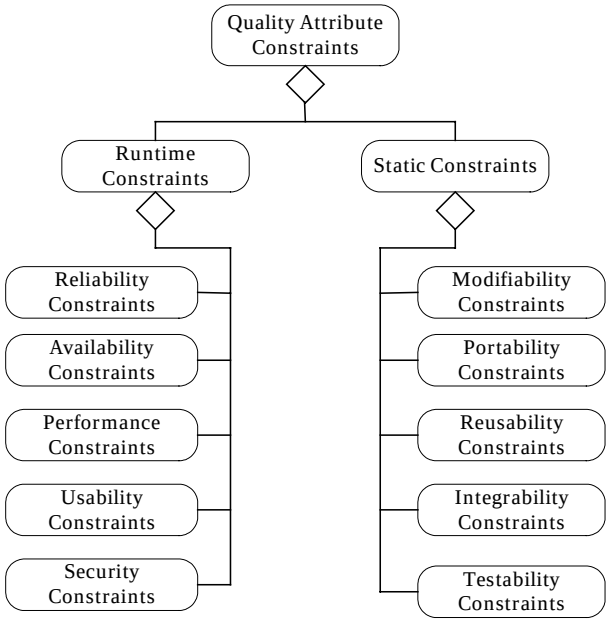

Figure 3: The Quality Attribute Constraints Map

The Runtime Constraints captures the Performance, Security, Availability, Usability and Reliability related constraints. The Performance Constraints are responsible for capturing the responsiveness of the system related to transactions per unit time, arrival rates and distribution of service request, processing times, queue sizes and latency. The Security Constraints captures the element's ability to resist unauthorized usage while continuing to provide its services to authorized users. The Availability Constraints captures the constraints on the availability of the architectural element. The usability related constraints are captured in the Usability Constraints. The Usability constraints are related to Learnability, Efficiency, Memorability, Error Avoidance and Error Handling. The Reliability Constraints captures the constraints of the component related to its consistent performance as per specifications.

The Static Constraints captures Modifiability, Portability, Reusability, Integrability and Testability constraints of the architectural element. The Modifiability Constraints captures issues related to the ease of changing or extending capabilities, ease of deleting capabilities, adapting to new operating environments, and restructuring the internals of the component. The support for the system's ability to run under different computing environment is captured in the Portability Constraints. The Reusability Constraints help specify the ability of the component to be used in different contexts. The issues related to the integration of the component to other components is captured in the Integrability Constraints while the Testability Constraints captures the testability related constraints. The testability related constraints are typically tied to the arch-element's observability and controllability.

The Deployment Constraints [Figure 4] captures an arch-element's deployment related constraints. The Deployment Constraints are partitioned into the Core Infrastructure Constraints and Interaction Constraints.

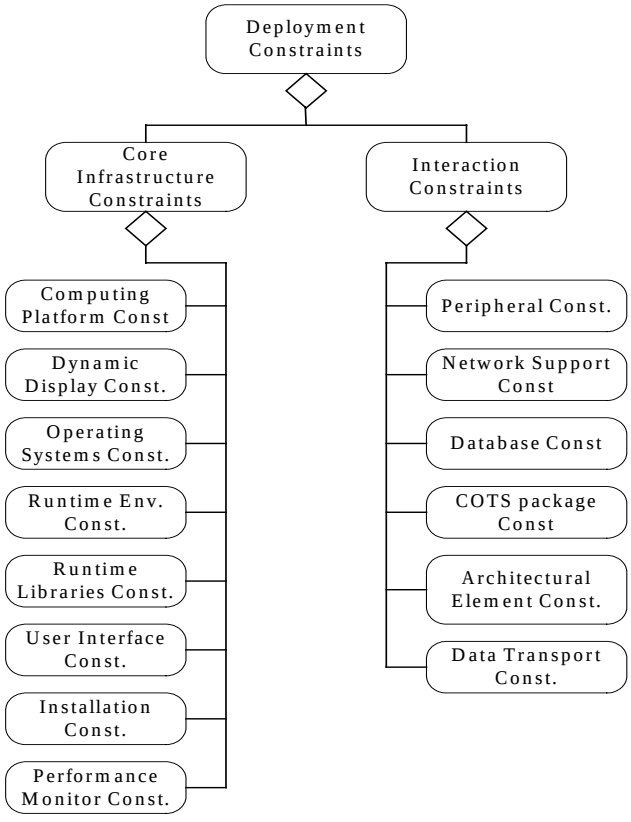

Figure 4: The Deployment Constraints

The Core Infrastructure Constraints for an arch-element captures the requirements for installation of the element on its base platform. It specifies the basic installation requirements for the component without consideration for its interaction with other system components. Hence, satisfaction of the Core Infrastructure Constraints specification does not imply proper functional operation of an arch-element. The Interaction Constraints on the other hand, captures the information about how an arch-element interacts with other elements in the architecture. Satisfaction of all the Deployment Constraint specifications, which includes both the Core Infrastructure Constraint and the Interaction Constraint specification, implies proper deployment of the component in the context of the overall architecture. The division of the Deployment Constraint into Core Infrastructure Constraints and Interaction Constraints was motivated by the goal of separately addressing the issues of an arch-element's own installation requirements versus its requirements for interaction with other arch-elements. The information captured in these two sets of constraints would help in reasoning over the deployment requirements of the arch-element from these two distinct perspectives. These constraints are optional and should be used as needed for capturing the non-functional specifications of an arch-element.

The Core Infrastructure Constraint is composed of the Computing Platform Constraint, the Dynamic Display Constraint, Operating System Constraint, Runtime Environment Constraint, Runtime Libraries Constraint, User Interface Constraint, Installation Constraint and the Performance Monitor Constraint. The Computing Platform Constraint captures information about the base platform on which the arch-element needs to be installed. For example, these constraints would specify that an arch-element should be deployed on an Intel Core 2 Duo series machine at a certain clock frequency with 1GB of memory and 80 GB of hard disk space. The Dynamic Display Constraint captures information about the display requirement of the arch-element. It captures information like the screen size, the vertical and horizontal scan frequency and viewing angle of the display for optimal viewing of the arch-element. These constraints are particularly important for graphics based arch-elements where display with a high resolution is required for proper viewing. The Operating Systems Constraint captures the possible operating systems in which the arch-element can be installed and executed. For example, this constraint specifies whether a particular software should execute on Windows 2000 as well as Windows XP. The Runtime Environment Constraint details the runtime environment information of the arch-elements while the Runtime Libraries Constraint captures information about the runtime libraries required for correct operation. The User Interface Constraint specifies the UI features that should be supported by the arch-element. The Installation Constraint captures the information of the installation requirements. It specifies information about the directory where the arch-element is

to be installed, the system files that are modified, the files that are placed in the system directory, the registry changes (in the case of Windows applications) made, etc. The Performance Monitor Constraint helps specify the details about performance monitors for the arch-element.

The Interaction Constraints are an aggregate of the Peripheral Constraints, the Network Support Constraints, the Database Constraints, the COTS Package Constraints, the Architectural Element Constraints and the Data Transport Constraints. The Peripheral Constraints details the peripheral dependencies of the arch-element. For example, if an arch-element transmits real-time data from a wireless computing platform, it would require a wireless modem. The Network Support Constraints captures information about bandwidth, throughput and other network related requirements for proper operation while the Database Constraints specifies the database(s) that the arch-element needs to interact with. The Middleware Constraints specifies the middleware requirements for the arch element and the COTS Package Constraints captures the dependencies on COTS packages. The Architecture Elements Constraints identifies the other arch-elements that the arch-element being specified interacts with. Finally, the Data Transport Constraints captures information about the way data is transported from the arch-element being specified to other arch-elements.

4.2 Form

By the Perry Wolf definition, the form is a set of weighted properties and relationships among components and connectors. A form defines constraints on the components and connectors and how they are placed relative to each other and how they interact.

Research and experience with software building over the years has resulted in the codification of collective experience of skilled designers, architects and software engineers. These proven solutions to recurring design problems are popularly known as *patterns*. Different kinds of patterns have been proposed – Architectural Patterns [7], Design Patterns [6] and Idioms. These help define the relationship between different components under given constraints and is relevant to the *form* of a software architecture. They generally impose a *rule* on the architecture that specifies how the system will handle a given aspect of functionality [2]. Architectural Styles is another concept that is relevant to the *form* of an architecture. Styles essentially abstract arch-element and the formal aspects from various architectures. They are often less constrained than specific architectures. Different architectural styles such as the pipe and filter, layered or blackboard promotes different quality attributes for a software system when they are defined at a global level. Several architectural styles can also be merged in a software architecture as long as the constraints of the two styles do not conflict. An example of two styles in an architecture is provided in Perry and Wolf 1992 [2]. Styles can also be applied in a localized fashion. Application of architectural styles helps define the *form* of an architecture.

The key constructs of our model that are relevant to the form of an architecture are architectural composition and architectural region. As explained previously, architecture composition represents the sub-architectural structure of an arch-element while architectural region provides a construct scoping mechanism and represents a collection of arch-element to which a set of constraints apply. These two constructs are demonstrated below

arch-composition =
(name, { arch-elements }, { mappings })

arch-region = (Descriptor,{ arch-elements | arch-compositions }, { general constraints })

While arch-composition and arch-region are the two fundamental concepts of our model, we also provide a construct for capturing the generic form of an architecture. The purpose of this construct is to capture in a granular fashion the different elements that make up the *form*.

The *form* of an architecture can be influenced by both functional as well as non-functional requirements. For a given software architecture model, the *form* needs to be specified at a global level and/or at a local level i.e. for an architectural region or sub-architecture , as for complex systems it may be impossible to specify the form at a global level. The *form* of architecture is modeled as in Figure 5.

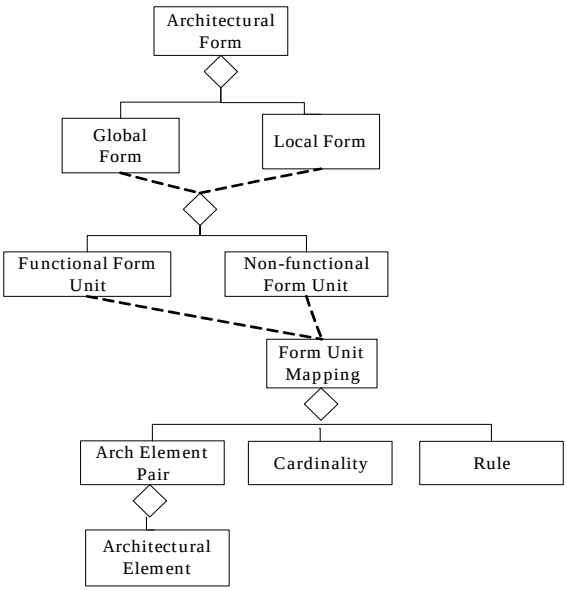

Figure 5: Architectural Form

A given style or a pattern is represented as a *Form Unit* in our model. Thus a subject observer pattern is a *form unit* with multiple *Form Unit Mappings*, where each form unit is represented by an *Architectural Element Pair*, the *Rule* for the relationship between the pair and the *Cardinality* between the pair. One arch-element is common across all the *Form Unit Mappings* for the subject observer pattern and serves as the Subject. The second component in the Arch Element Pair for the f*orm unit mappings* represents the Observers.

4.3 Rationale

The rationale in our architecture model is the set of justifications for the choice of elements and formal aspects of the architecture. A rationale ties architectural design decisions to various system drivers – for example decisions may be tied to functionality requirements from the user, non functional system constraints, market requirements and business strategies. In fact the constraints mentioned earlier in this paper provide an extremely useful tie between system drivers and the architectural design; they provide a form of self-documenting rationale.

In our model we treat rationale as atomic units that may be associated with any aspect of our specification. They are sprinkled over every facet of our architecture. Off course these can be later categorized into convenient groups but we do not model rationale using either a hierarchy or decomposition to reinforce the fact that justifications for an architectural decision is often independent of the level of abstraction for which a design decision is needed.

5. Example Software Architecture and Usefulness of Proposed Model

The proposed approach for specification was used for documenting the architecture of the tool EASE (Environment for Architecture Specification and Evaluation), that is being built as part of this research.

Summary information about the architecture is given in the table below.

	Architectural Element	**No. of Services**
1.	Architecture Compliance Metrics Evaluator	8
2.	Component Characteristic Metrics Evaluator	14
3.	System Configuration Metrics Evaluator	12
4.	Architectural Style Predictor	8
5.	Architectural Divergence Calculator	10
6.	Architectural Drift Calculator	3
7.	Architectural Erosion Calculator	3
8.	Report Generator	7
9.	Display Manager.	4

Table 1: EASE Architecture

The details associated with the architecture cannot be presented here due to constraints of space. We present the abstract model for EASE with only one of the arch-elements highlighted in Figure 6.

Figure 6: Example Architectural Specification

Several architectural analysis techniques have been developed based on our abstract architectural model. We briefly mention each of these below.

Contextual Reusability Metrics [25]: A set of contextual metrics have been developed that provide a mechanism for a quantitative evaluation of software component reuse in the context of architecture requirements (functional and data) and architecture structure. We leverage the requirements represented within an architectural specification to provide the context for a component to evaluate the compliance of these components to the architectural specification, to assess the similarity between components, the component's coverage of the architectural description, as well as numerically tracking the evolution of a component in terms of the specification. Our reusability assessment goes beyond simple interface matching and helps system integrators explore behavioral characteristics of components as well. These metrics provide a quantitative mechanism for assessing reusability leveraging the context of a component.

Predicting Emergent Properties of Systems [26]: We have also developed an approach for reasoning about architectural styles using the specification approach explained in this paper. With our style prediction proposal, not only will a System Architect have the ability to evaluate several deployment options but will also have the ability to get a sense of the quality attributes of the final system before actual construction. We can also use our approach to determine the conformance of a system

configuration to a particular style. This will be particularly useful during the evolution of a system to detect either architectural drift, or architectural erosion

Assessment of System Evolution [27]: Using our abstract architectural model, we have also developed a methodology for the architectural assessment of system evolution. We have proposed as set of architectural divergence, drift and erosion indicators that provide objective measures for the assessment of system change.

6. Summary

In this paper we propose an architectural model for documenting the specifications of architectural elements, the form of the architecture as well as the justifications for the different design decisions. It is envisioned that an architecture described using our architectural specifications model can be instantiated using asset components that are registered against individual architectural elements. The architecture specification model enables architecture evaluation for assessing reusability potential, for predicting emergent properties of system as well as for tracking system evolution.

7. References

[1] Jackson, M, "The World and the Machine", Proceedings of the 17th International Conference of Software Engineering, Seattle, WA, 1995

[2] Perry, D. E., Wolf, A. L., "Foundations for the Study of Software Architectures", ACM Software Engineering Notes, 17, 4, October 1992, 40-52

[3] de Marneffe, P. A., "Holon programming: A Survey", Universite de Liege, Service Informatique, 1973.

[4] Knuth, D. E., "Structured Programming with go to statements", ACM Computing Surveys, Volume 6 Number 4, Pages 261-301, December 1974.

[5] Sommerville, I., Sawyer, P., "Requirements Engineering – A Good Practice Guide", John Wiley & Sons, 1998

[6] Gamma E., Helm R., Johnson R., Vlissides J., "Design Patterns Elements of Reusable Object-Oriented Software", Addison-Wesley, 2002

[7] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., "Pattern Oriented Software Architecture", Wiley Series in Software Design patterns, 2001

[8] Biggerstaff, T. G., Perlis, A. J., "Software Reusability" ACM Press, 1989

[9] Krueger, C. W., "Software Reuse. Computing Surveys", 24(2): 131-183, June 1992

[10] Mettala, E., and Graham, M. H., "The Domain Specific Software Architecture Program". Technical report CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, June 1992.

[11] Karlson, E. A., Guttorm S, Stalhane, T., "Techniques for Making More Reusble Components," REBOOT Technical Report #41, 7 June 1992

[12] Karlson, Even-Andre (ed.), "Software Reuse: A Holistic Approach", Wiley, New York, NY, 1995.

[13] Shaw, M., Garlan, D., "Software Architecture: Perspectives on an Emerging Discipline", Prentice Hall, 1996

[14] Szyperski, C., "Component Software: Beyond Object Oriented Programming," Addison-Wesley, 1999

[15] Johnson, R and Foote, B, "Designing Reusable Classes," Journal of Object Orientrd Programming, 1 (2), 22-5.

[16] Bosch, Jan, "Design & Use of Software Architectures: Adopting and evolving a product-line approach," Addison-Wesley, 2000

[17] Garlan, D., Allen, R., Ockerbloom, J., "Architectural Mismatch or Why it's hard to build systems out of existing parts", Proceedings of the Seventh International Conference on Software Engineering, April 1995.

[18] Tracz, W., "Software Reuse Maxims," ACM SIGSOFT Software Engineering Notes, Vol. 13, No. 4, October 1998, pp. 28-31

[19] Tracz, W., "A Conceptual Model for Mega programming," ACM SIGSOFT Software Engineering Notes, Vol. 16, No. 3, July 1991, pp. 36-45.

[20] Perry, D. E., "Generic Descriptions for Product Line Architectures", *ARES II Product Line Architecture Workshop*, Los Palmos, Gran Canaria, Spain, February 1998

[21] Habermann, A. N., Perry, D. E., "Well Formed System Composition", Carnegie-Mellon University, Technical Report CMU-CS-80-117. March 1980

[22] Perry, D. E., "The Inscape Environment: A Practical Approach to Specifications in Large-Scale Software Development. A Position Paper", January 1990.

[23] Luckham, D. C., Vera, J., "An Event-Based Architecture Definition Language", IEEE Transactions on Software Engineering, vol. 21, no. 9, pages 717-734, September 1995.

[24] Bhattacharya, S. "Specification and Evaluation of Technology Components to Enhance Reuse," Masters Thesis, The University of Texas at Austin, July 2000

[25] Bhattacharya, S., Perry, D. E., "Contextual Reusability Metrics for Event-Based Architectures", The 4th International Symposium on Experimental Software Engineering (ISESE), November 2005, Australia

[26] Bhattacharya, S., Perry, D. E., "Predicting Emergent Properties of Component Based Systems", The 6th IEEE International Conference on COTS-based Systems (ICCBSS), February 2007, Canada

[27] Bhattacharya, S., Perry, D. E., "Architecture Assessment Model for System Evolution", The 6th Working IEEE/IFIP Conference on Software Architecture (WICSA), January 2007, India