Statically Defined Dynamic Architecture Evolution

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

Abstract

There are a variety of contexts where dynamic architecture evolution is needed. The context we have been working in is that of providing architecture models of NASA training simulations for manned space exploration. In this context, as in many others, we do not need unrestricted dynamic evolution, but only a limited form of dynamic evolution where the transitions and boundaries of that evolution are well understood. We present our approach to this restricted form of dynamic evolutions in the context of an abstract architecture model and use an architecture of architectures with transition connectors as the means of prescribing our statically defined dynamic architecture evolution. Further, we present an incremental mechanism for generating the needed architectures and validate our approach with an implemented prototype.

I. Introduction

Some system designs benefit from architectures that allow some flexibility at run time. For example, systems with clearly defined “modes” may exhibit different architectures in each operational mode. When these systems are modeled by a single architecture, that architecture can accurately reflect the implementation concerns of the end product, but not the individual behavioral aspects.

We have been working on architectures for software simulators applied to the simulation of manned exploration missions. This application domain exhibits architectural changes as elements of the physical system are assembled, disassembled, reassembled, and even destroyed as in the case of a discarded component that burns up on reentry into earth's atmosphere. This is more than rearranging components to obtain equivalent functionality. Instead, these changes result in new and often unique architectures.

The architecture of simulators in this application domain are strongly influenced by the architecture of the systems being simulated. Consequently the architecture of the simulator tends to be very dynamic. Traditionally architectures for these systems have been composite, with all elements that may appear in the simulation present in a single architecture. Such a view is concrete, representing the simulator as it would be constructed.

Our intent is to construct abstract architectures that allow designers to work with high level views of the system. We want to consider the individual architectures as they are *exhibited* by the system. In this view of architecture, a system has a single composite architecture that is the conjunction of a set of exhibited architectures. An exhibited (or apparent) architecture being the architecture the system appears to have, as determined by it's functionality, at some moment of the systems' operation.

This view of architecture can be obtained by constructing a complete architecture for each exhibited architecture. In our work this approach has lead to a proliferation of architecture specifications having a great deal in common. An undesirable consequence is that the architecture as a whole contains considerable redundant specification.

In this paper we propose a simplifying construct, the arch-transition, that permits the architecture to be specified as a set of apparent architectures and avoids much of the redundancy encountered when the apparent architectures must be individually specified in full.

In section 2 we review related work on dynamic architecture. In section 3 we discuss the need for separation of concerns. The approach used for the modeling is described in section 4. Section 5 discusses the proposed model while section 6 provides an example software architecture that illustrates the abstract model.

The contributions of this paper are an abstract architectural model and a statically defined mechanism for limited dynamic evolution of the architecture. These contributions are summarized in section 7.

II. Related Work

Existing work with dynamic architecture tends toward concrete architectures and runtime reconfiguration of the software implementation. Often architectural change means a change in configuration that provides the same or similar functionality, such as in the case of a fault-tolerant system. An elegant example of this type of dynamic architecture is described by Grondin, Bouraqadi, and Vercouter [7] who define a single model of the software system. A runtime engine uses the model as a goal that it attempts to satisfy in response to changes in the execution context.

Hirsch, Kramer, Magee, and Uchitel [8] apply modes as a form dynamic configuration. A mode is a sub-architectural region of the architecture that can be replaced with another region in a process called a transition. Mode transitions differ from the arch-transitions we present in that mode transitions provide reconfiguration for the purpose of maintaining existing functionality. That is, given a mode that provides a service, a transition on that mode will continue to provide the same service while utilizing different resources.

Wermelinger [9] describes a method of using chemical abstract machines to define dynamic architectures. Like many other dynamic ADLs, his method employs a central configurator that controls the dynamic aspect of the architecture. Wermelinger's configurator adjusts the architecture in response to changes in available components, equivalent to our effector functions. Not addressed in this paper is a method to select or restrict specific architectural changes and it appears that all permissible architectural changes will be reversible.

In [10] Wermelinger, Lopes, and Fiadeiro describe a language for architectural reconfiguration. Changes in configuration are implemented by scripts. Scripts are invoked at runtime by a user or some automatic trigger mechanism. Consequently their language is not intended to provide a static description of the dynamic architecture. Interestingly, the language only provides a facility add or remove elements but not to move an element within the architecture. This appears to lose continuity of element instances during reconfiguration, comparable to a loss of state in the implementation.

Magee and Kramer [11] describe an approach to dynamic architecture using the Darwin ADL. Their approach embeds constraints on dynamic configuration within the component specifications. Any assembly of components that is consistent with these constraints is valid. This approaches constrains the ability of the system to reconfigure to a lesser degree than ours, and does not contemplate enumeration of the set of possible configurations.

Le Metayer [12] describes a graph based approach to describing architectures. A collection of graph rewriting rules (called the coordinator) define allowable changes that the architecture may undergo. These changes can be applied iteratively to define any number of architectures. They apply graphs to individual architectures but not the architecture-of-architectures as we describe in this paper.

III. 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. However, the applicability of this strategy to architectural design is unclear.

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]. The idea behind the Shanley Principle is that one part can perform multiple functions. It has been wonderfully explained by Jackson [1] as “the architecture of the world has been designed with the fullest possible application of the Shanley Principle”. While efficient design is definitely our motivation, separation of concerns is not in contradiction of the Shanley Principle 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 even the creators of the solution.

We base 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 the connectors that define the component interactions. 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 distinct from capturing what individual components do or need.

IV. Approach for Specifications

Our primary goal is to create an abstract model of software architecture (i) to provide reasoning about component composition and (ii) 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).

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 the use. The differences are as follows: an *architectural prescription* defines the important constraints on the architecture – i.e., it defines important, but not necessarily all components and connectors, their critical properties (though again, not necessarily all of them), and the critical relationships 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 industry, it seems that building a descriptive architectural model would not be possible until the very last iteration, and by then most of the key architectural and design decisions would already have been made.

V. 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*: Can be either a *component* or a *connector* (while their structure for purposes of modeling and analysis is identical, they have distinct logical purposes, i.e., connectors represent interactions among components). This construct represents basic elements in the architecture.
- *Arch-composition*: An *arch-composition* represents a sub-architectural structure. As such it represents the substructure of an *arch-element* and must satisfy the interface constraints of the elements involved. 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.

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 that 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 and/or compositions 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 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 5.2.

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

A. The Elements: Components and Connectors

A software architecture specification is partitioned into several arch-elements. These arch-elements are driven initially by functional partitioning and also introduce the notion of object orientation that 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 model we have not differentiated data, processing and connecting elements but conflated them all into arch-elements.

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 can 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 so that 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-Condition constraints, while the Output constraints include the Output Data, Output Events and the Post-Condition constraints. The Pre-Condition 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-Condition Constraints capture the arch-element's state that should be satisfied upon execution of the service.

General constraints can be functional constraints or 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 arch-element may also be represented. The general constraints are shown below:

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 })

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)

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.

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.

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 requirements related to the integration of 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 captures an arch-element's deployment related constraints. The Deployment Constraints are partitioned into the Core Infrastructure Constraints and Interaction 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.

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.

B. 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 building software 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 [5], Design Patterns [4] 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 Style 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. Examples of styles in an architecture are provided in Perry and Wolf [2] and by Perry in [6]. 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 scooping 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 elements that make up *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.

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.

C. Architecture-of-architectures

When many architectures are developed having a large number of elements in common, it is convenient to merge them into a single architecture-of-architectures. An individual architecture within the composite is an arch-configuration and is represented by an arch-region. Utilizing arch-configurations reduces redundancy by allowing a single arch-element definition to appear in multiple architectures (arch-configurations). Additionally, corrections or improvements to an arch-element immediately benefit all dependent architectures.

In some architectures, specifically the NASA simulator architectures we have been developing, significant redundancy remains among the arch-configurations. The redundancy is a consequence of having distinct arch-configurations that are closely related. These arch-configurations have a large portion of their substructure in common with differences limited to a few areas. We address this redundancy by describing differences among arch-configurations rather than providing complete descriptions of each arch-configuration. We assume that at least one arch-configuration exists and is fully described. Additional arch-configurations can be described by the differences between new (derived) arch-configurations and existing (source) arch-configurations. Of course the derivation of arch-configurations can be carried out to any number of levels. Moreover, derivation need not be idempotent, so performing a single derivation repeatedly could yield many new configurations.

We represent the derivation of one arch-configuration from another by introducing connectors among arch-configurations. Connectors in this role consist of two constraints, a *predicate* and an *effector function*. The predicate restricts the source arch-configuration and the effector defines how the derived arch-configuration will differ from the source.

Using connectors among arch-configurations in this way, an architecture consists of one or more explicit arch-configurations plus some number of derived arch-configurations obtained by applying connectors to explicit and derived arch-configurations. A directed graph can be formed from the resulting architecture-of-architectures, where the vertices of the graph are arch-configurations and the edges are connectors. Provided that the description of the differences between pairs of arch-configurations is simpler than the arch-configurations themselves, the architecture as a whole is simplified.

We use arch-configurations to capture various *apparent architectures* exhibited by a highly dynamic system. A single arch-configuration captures the apparent architecture of the system over one or more intervals of time. At points where two such intervals are adjacent there exists a transition from one apparent architecture to another. Since the architecture of each interval is captured by an arch-configuration, the transition is naturally represented by a connector, and in the context of dynamic architectures we refer to these connectors as *arch-transitions*. Arch-transitions then, capture changes in the apparent architecture of the system, and in our work with simulators, a change in the physical architecture of the simulated system. Thus many arch-transitions are ultimately determined by the physics and design of the system being simulated.

Our approach to an architecture-of-architectures accommodates dynamic properties by introducing new architectures where implied by explicitly defined arch-configurations and arch-transitions. To add a bit of formality to this statement, assume an architecture a which has an arch-configuration c and arch-transition connector t. Let p and e be the predicate and effector of t. c' is an arch-configuration of architecture a if $p(c)$ holds and c' is the configuration obtained by applying e to c.

This extension to architecture allows for many (possibly infinitely many) architectures to be specified in a very compact form. The intuitive justification for this extension is that the arch-transitions are capturing those changes that the physical system is capable of performing, thus limiting the possible arch-configurations to precisely those mandated by the physics and design of the physical system. This is precisely the behavior desired from a simulator. Although we have introduced the possibility of an infinitely varying architecture, the limitations of the physical system has constrained this set to be finite.

D. 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.

VI. Application of Abstract Model

We have constructed a prototype application to read and analyze architectures. This prototype can evaluate the derivation of selected arch-configurations and show the architecture that is produced.

In the examples that follow it will be apparent that the language accepted by the prototype adds some flexibility to the description of arch-transitions, beyond what was described in section 5. Specifically, arch-transitions are not limited to a single predicate and effector function. Instead, both tasks are carried out incrementally by a sequence of statements in procedural fashion. The examples that follow illustrate how these statements are used.

Our prototype has been applied to the specification of architectures for simulators used in NASA's manned space exploration. Much of these simulator architectures reflect the architecture of the physical systems being simulated. These physical systems undergo architectural changes as the vehicles reassemble into new configurations during operation. Additionally, some elements cease to exist as missions progress, either because the element has been destroyed (such as during reentry), because a reconfiguration causes a physical element to no longer have a physical embodiment (such as *fullstack* discussed below), or because an element can no longer contribute usefully to the simulation and is removed for simplicity (such as a discarded booster engine). Our example has been simplified for space and to focus on our use of arch-transitions.

A. Architecture-of-architectures Graph

Fig. 1 shows a graph of arch-configurations within an architecture. This architecture describes a simulator for an Apollo-like vehicle from launch through docking with the ISS (International Space Station) and return to earth. Nodes in this graph are arch-configurations and edges are arch-transitions. Configuration *onpad* is an explicitly specified arch-configuration. All other configurations are derived from *onpad* by the application of arch-transitions.

B. Explicit Arch-configurations

Fig. 2 is an excerpt from the architectural specification that produces the graph in Fig. 1. Configuration *onpad* is an explicit arch-configuration for the pre-launch configuration of the system. We can derive most other arch-configurations that are needed from this initial arch-configuration. All of the components needed to perform pre-launch simulation are referenced here, the most important of which is *fullstack*, the component that represents the complete vehicle assembly at the time of launch. Additional simulation components and their functions are: *ISS* (International Space Station); *environment*, the physical properties of space, planets, moon(s), stars, etc.; *simulator*, those aspects of the system which are not being simulated, including the simulation system itself. A trivial constraint *onpad* uniquely identifies this configuration to be situated on the launch pad (i.e. it is a pre-launch configuration).

C. Arch-transitions

It can be seen from Fig. 1 that transition *launch* can be applied to configuration *onpad* to obtain the post-launch architecture *stage1*. The difference between architectures *onpad* and *stage1* is subtle and mostly consists of the fact that *fullstack* is not rigidly attached to the earth in configuration *stage1*. The specification for arch-transition *launch* is shown in Fig. 2.

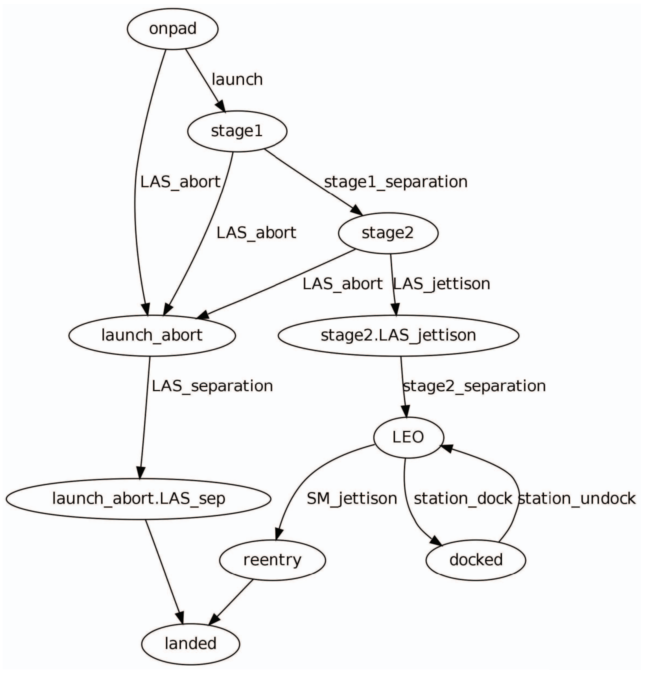

Figure 1: Example Arch-Configuration Graph

```
configuration onpad {
   component fullstack;
   component ISS;
   component environment;
   component simulator;
   constraint onpad;
}

composition fullstack {
   component cm;
   component sm;
   component stage1;
   component stage2;
   component las;
}

transition launch {
   global var conf;
   conf = rename(conf, stage1);
   if (!has_constraint(conf, onpad)) return(false);
   conf = remove_constraint(conf, onpad);
   return(true);
}

transition stage1_separation {
   global var conf;
   conf = rename(conf, stage2);
   if (!has_component(conf, fullstack)) return(false);
   if (has_constraint(conf, onpad)) return(false);
   conf = replace(fullstack, {..stack_stage_two, ..stage1});
   return (true);
}

transition LAS_abort {
   global var conf;
   conf = rename(conf, launch_abort);
   if (exists(conf, cm_las)) return(false);
   if (!exists(conf, las)) return(false);
   if (find_parent(conf, las) != conf)
      return (false);
   conf = move(las,  conf);
   conf = move(cm, conf);
   conf = add_element(conf.cm_las, conf.cm, conf.las);
   conf = add_constraint(conf, aborting);
   return(true);
}
```

Figure 2: Example Architectural Specification

Most arch-transitions will declare a global variable *conf*. When the evaluation of an arch-transition begins, the value of *conf* is an arch-configuration identical to the source configuration.

The work of the effector function is accomplished by assigning new values to *conf*. Multiple assignments may be made, with each assignment prior to the final one providing an unfinished intermediate configuration. The derived configuration produced by the transition is the value of *conf* when evaluation of the transition is complete. In the case of transition *launch*, there is one significant step in the derivation, removing the *onpad* constraint.

Statement *return(false)* indicates a condition where the arch-transition predicate does not hold and therefore the arch-transition is not valid for the source configuration. Statement *return(true)* indicates the predicate holds and the production of a new derived arch-configuration is complete.

Transition *LAS_abort* captures the change that occurs when the LAS (Launch Abort System) is activated. This transition applies to several configurations and has the effect of making the combined CM/LAS assembly a separate vehicle. The rename function specifies a new name for a configuration in the architecture-of-architectures graph.

Validity checks are performed to ensure that the transition is appropriate, given the source configuration. These checks include: the CM/LAS assembly must not be an element of the configuration, that would indicate that an LAS abort transition has already occurred; the LAS must exist somewhere in the configuration, otherwise it has been jettisoned and dropped from the simulation; the LAS must not be an independent element in the configuration, that would indicate that the LAS has been jettisoned.

The effector function is accomplished in four steps. First, the CM and LAS are moved to become children of the configuration (as if they were no longer part of any larger assembly). This is a convenience that avoids the need to specify exactly where in the source architecture these elements appear. A subsequent step combines the CM and LAS (from their now known locations) into a new composite element *cm_las*. Lastly a trivial constraint is added indicating that the derived configuration will represent an aborting configuration.

Transition LAS_abort is interesting because, generally, the LAS can be used to effect an abort procedure anytime prior to being jettisoned. This includes the pre-launch configuration and extends usually sometime into the second stage burn. This period spans several arch-configurations. Activating the LAS from each of these source configurations produces a unique new architecture. Without arch-transitions, several explicit arch-configurations are needed to capture each possible resulting architecture. Here we use just one arch-transition that captures only the changes that need to be applied to each source configuration. By applying this transition to each applicable source configuration (a process automated by our prototype), all consequent arch-configurations are included in the architecture.

Note that the graph in Fig. 1 shows *launch_abort* to be the only arch-configuration resulting from the application of *LAS_abort*. This is a simplification in the graph representation indicating that the architecture that results from each application of *LAS_abort* is logically the same even though details may differ depending on the source configuration. Thus, *launch_abort* is actually a composite of three arch-configurations.

Our prototype analysis tool can evaluate an architecture and produce an architecture-of-architectures graph. These graphs can be useful to check for errors in the architecture. Problems in the specification of transitions may introduce erroneous configurations, transitions in unexpected locations, or missing

configurations and transitions. In our work with simulators, the nature of the system being simulated makes this kind of validation easy to perform. The expected form of the graph is easily compared to expected system behavior.
The prototype can also synthesize an explicit arch-configuration from any derived configuration. This allows designers to obtain a full view of any arch-configuration even though the information is not directly available from the specification. This is a useful feature that allows designers to easily explore and validate the ramifications of their design decisions.

VII. 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. Our intent is to provide an abstract model sufficiently rich to support the compositional requirements of architectures and to be able to reason about, analyze, and evaluate architectures described in our abstract model. To better realize this intent we have utilized an architecture-of-architectures so that we can construct separate architectures to independently specify the transient behaviors of the system. We introduce connectors among these several architectural configurations so that we can specify one architecture as a variation, evolution, or modification of another. In our work with simulators we find it is natural to think of the system architectures in this way, by what changes from one architecture to the next. We have validated our approach using existing simulators and have constructed a prototype architecture analysis tool to support them.

VIII. Future Work

We are currently expanding the scope of our example architecture in space exploration to model a larger portion of the system. From this detailed model we are exploring the possibility of generating executable simulators. In addition, our current work indicates that the composite configurations displayed in Fig. 1 are a useful feature that benefits understandability. That introduces a need to consider the analytical implications of arch-configuration that represent sets of unique configurations united by arbitrary semantic (user) considerations. We are also evaluating the need for additional architectural constructs. In particular, a construct to act as a modifier of architectural transitions much like transitions are modifiers of arch-configurations.

Acknowledgement

The research described here was funded in part by NASA grant NNX08AC48G under the direction of Toby Martin, and NSF CISE grants SRS Grant CCF-0820251 and SOD IIS-0438967.

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

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

[6] Perry, D. E., “A Product Line Architecture for a Network Product”, ARES III: Software Architectures for Product Families 2000, Los Palmos, Gran Canaria, Spain, March 2000, Springer-Verlag, LNCS 1951, p39-52

[7] G. Grondin, N. Bouraqadi, and L. Vercouter, “Madcar: an abstract model for dynamic and automatic (re-)assembling of component-based applications,” in 9th Int. SIGSOFT Symposium on Component-Based Software Engineering (CBSE2006), No 4063, LNCS. Springer, 2006, pp. 360–367.

[8] D. Hirsch, J. Kramer, J. Magee, and S. Uchitel, “Modes for software architectures,” in of Lecture Notes in Computer Science. Springer, 2006, pp. 113–126.

[9] M. Wermelinger, “A simple description language for dynamic architectures,” in ISAW ’98: Proceedings of the third international workshop on Software architecture. New York, NY, USA: ACM, 1998, pp. 159–162.

[10] M. Wermelinger, A. Lopes, and J. L. Fiadeiro, “A graph based architectural (re)configuration language,” SIGSOFT Softw. Eng. Notes, vol. 26, no. 5, pp. 21–32, 2001.

[11] J. Magee and J. Kramer, “Dynamic structure in software architectures,” SIGSOFT Softw. Eng. Notes, vol. 21, no. 6, pp. 3–14, 1996.

[12] D. Le Metayer, “Software architecture styles as graph grammars,” SIGSOFT Softw. Eng. Notes, vol. 21, no. 6, pp. 15–23, 1996.