NASA/CR–2012-217766

Modeling Techniques for High Dependability Protocols and Architecture

Brian LaValley, Peter Ellis, and Chris J. Walter
WW Technology Group, Ellicott City, Maryland

September 2012

NASA/CR–2012-217766

Modeling Techniques for High Dependability Protocols and Architecture

Brian LaValley, Peter Ellis, and Chris J. Walter
WW Technology Group, Ellicott City, Maryland

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NNL10AB32T

September 2012

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Contents

THIS PAGE INTENTIONALLY LEFT BLANK

Modeling High Dependability Protocols and Architectures

This report documents an investigation into modeling high dependability protocols and some specific challenges that were identified as a result of the experiments. The need for an approach was established and foundational concepts proposed for modeling different layers of a complex protocol and capturing the compositional properties that provide high dependability services for a system architecture. The approach centers around the definition of an architecture layer, its interfaces for composability with other layers and its bindings to a platform specific architecture model that implements the protocols required for the layer.

Challenges

The design of complex systems relies on the use of protocols for executing functions and coordinating system operations and components. Protocols are often described in relatively simple terms, such as the Ethernet communication protocol, where the intricate aspects are masked by the overall packaging of the protocol. In reality, individual protocols can be very complex in nature, involving numerous steps that comprise the complete sequence of behavior. When multiple protocols are combined into a composite, complex interactions are possible; some desirable and others unwelcome. Protocol stacks are an example of the former while deadlocked systems are an example of the latter.

Capturing and analyzing the low-level protocol services and their associated behavior within architecture description languages, such as AADL [9][10], has proven challenging to date. Model based architecture design of highly dependable and safety-critical systems is relatively new. The increased system complexity creates difficulties that current techniques have yet to solve. Specifically, any architectural dependability analysis needs to be cognizant of the protocol services and their respect fault masking, reconfiguration and failure modes. Previously, separate modeling efforts have been used to assess these behaviors.

This report explores modeling and analysis of a specific highly dependable protocol and using the lesson learned to propose new methods for capturing the salient properties of a representative set of protocols in an integrated architecture framework that facilitates automated extraction from a unified model for analysis.

Experiences with Modeling and Analysis of Dependable Protocols

In complex communication protocols with many connections between distributed system components it is difficult, if not impractical, for the designer to envision all the potential factors that can influence the dependability of the network. At the network level, the basic concerns involve bit error ratios, arrival time assumptions of simultaneous errors and adequate scheduling and load management algorithms. Higher level functions rely on distributed communications to perform required operations and assumptions are implicitly made with respect to network and messaging properties. If these assumptions are violated during run time operation, errors can leak throughout the system and reach areas that were not designed to tolerate the harmful error effects. Furthermore, this leakage may be manifested either as an active error that requires immediate attention or as a latent error that seeds conditions that can infect state behavior and lead to unanticipated scenarios (e.g. simultaneous multiple errors) that cause system failure.

Standard techniques typically aggregate the analysis at higher level system functions. Much of the impetus for this approach was due to the observation that it is less expensive and more flexible to manage errors at higher levels of the system, particularly those with modest response time requirements. This insight led to dynamic growth in the field of highly available systems in commercial transaction processing in the late 1980s. This approach continued with the advent of internet technologies and companies such as Google shunning rigorous fault tolerance for a strategy of using server farms comprised of cheap replaceable components. The large commercial success of this strategy left little room for the growth of networks that supported more rigorous fault tolerant architecture designs.

This trend is changing as distributed computer systems become integral parts of mission critical systems and infrastructure. To date, fly-by-wire aircraft control systems have led the way in identification of challenge problems and innovative solutions. Faster processors and communications have exponentially increased the possible behaviors, both good and erroneous, that can happen in the response time window. Keeping track of all possible system behaviors without a structured systems approach quickly equates to a problem of state explosion and intractable conditions for analysis.

As noted above, the role of a communications network as the information backbone of a system makes it an ideal candidate for analysis, especially when the sheer extent of connections to many critical parts of the system is considered. By identifying the strengths and weaknesses of the network, the system architect can determine whether sufficient capabilities exist or additional guards and protocols are needed to handle identified elements of risk. Consequently, our goal in analysis of highly dependable protocols is to first examine the ability of a protocol to suppress and mitigate error effects and, if not fully covered, to ascertain the extent of the propagation of error effects.

As part of the work we investigated a network designed to address the challenges related to mission critical avionic systems. Using the BRAIN (Braided Ring Availability Integrity Network) [11] protocol, we modeled the protocol, examined error propagation and mitigation strategies and performed analysis on portions of the protocol. The BRAIN is a fault tolerant time-triggered communication medium that is aimed at embedded, real-time systems. The BRAIN was selected as a potential use case for modeling and analysis of fault tolerant systems on this effort and may be used as part of the Asynchronous TMR case study effort.

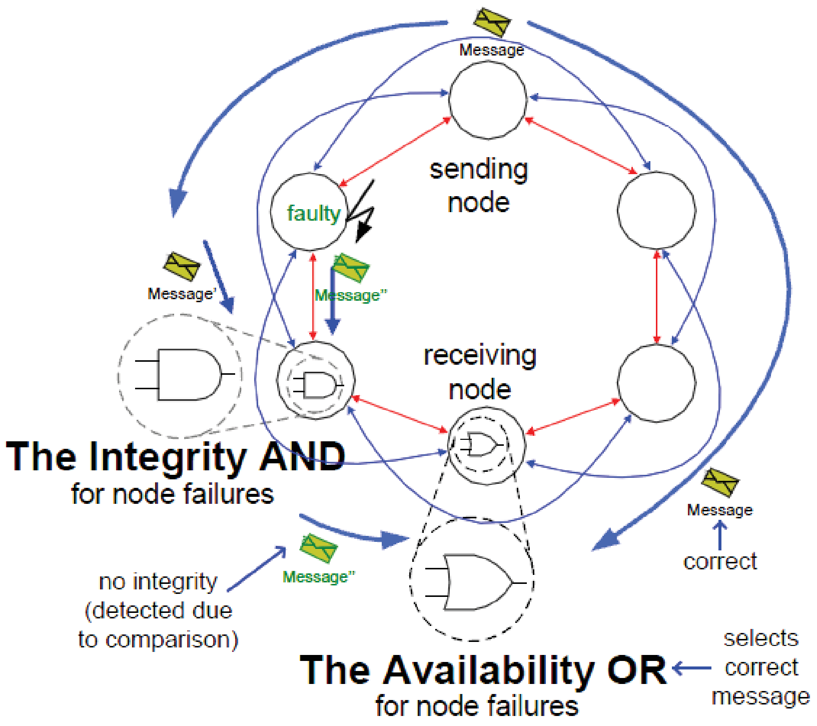


Figure 1 BRAIN Self-Checking Data Relay

The BRAIN uses several complementary fault tolerance protocols to perform reliable message delivery across the network. These protocols provide services such as clock synchronization and clique resolution, message-based self-checking pairs, data path integrity reconstitution and self-checking data relay among others. Each of these protocols plays an important role in the ability of BRAIN to provide the overall fault tolerant characteristics of the network. The fault tolerant protocol modeling and analysis effort focused on the Self-Checking Data Relay protocol (Figure 1) within BRAIN.

An error scenario was created and analyzed involving an AADL model of the BRAIN architecture using EDICT's error handling capabilities in order to evaluate the modeling capabilities and the ability of current modeling techniques to capture the behavior of the fault tolerant services used in BRAIN. The error propagation scenario was configured with a single value-symmetric error originating at the thread in a BRAIN node's host process. Mitigators were deployed on both the left and right incoming link pairs of each Bus Interface Unit (BIU). Each mitigator had the capability to detect data disagreement and perform one of two actions: (a) propagate the error, corresponding to the case where the incorrect data propagated in on the skip link, or (b) tolerate and not propagate the error, corresponding to the case where the incorrect data propagated in on the direct link.

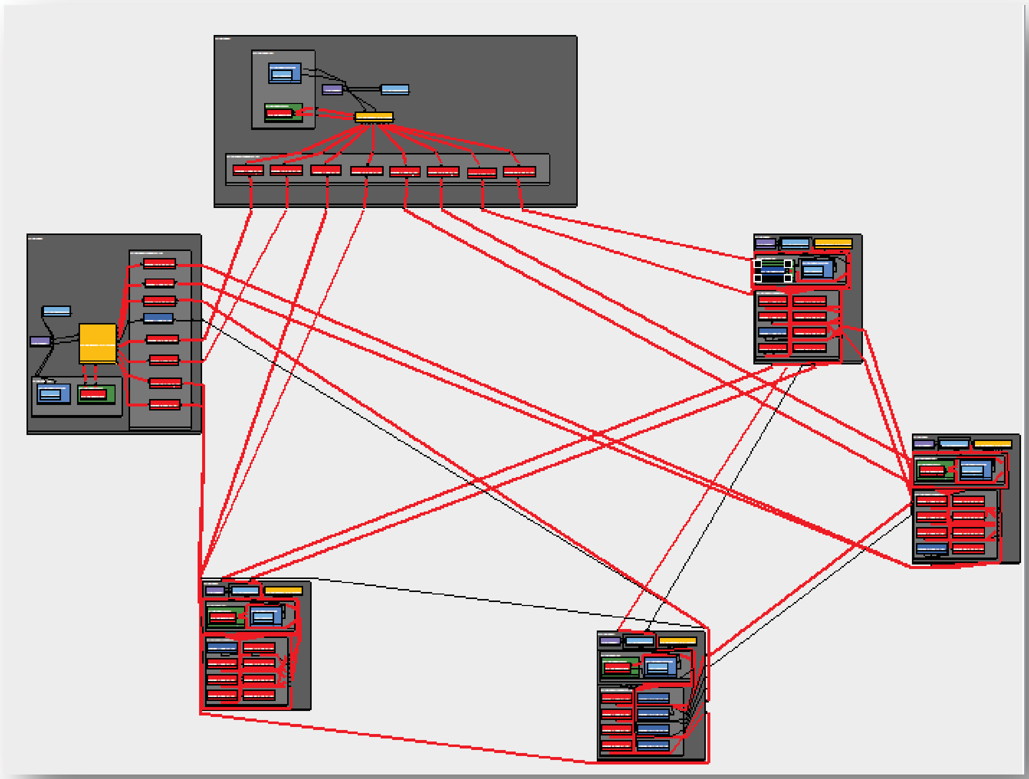


Figure 2 An EDICT single-error propagation scenario on the BRAIN architecture.

In this analysis, the error was able to compromise every node and had the potential to go undetected (see Figure 2) as expected because it was a source error without redundancy. Evaluation of these results demonstrated:

1) Validation of the concern relating to the potential for a network in a mission critical architecture to propagate and infect many other parts of the system.
2) The ability of EDICT to sufficiently represent the structure of the BRAIN architecture.
3) The ability of EDICT to model the flow of errors through the structural paths.

The experiment did identify a deficiency in capabilities needed specify the dynamic behavior of the protocols through AADL or existing EDICT extensions. Some aspects of the BRAIN protocols could not be accurately modeled and represent future areas for developing new capabilities and analysis procedures.

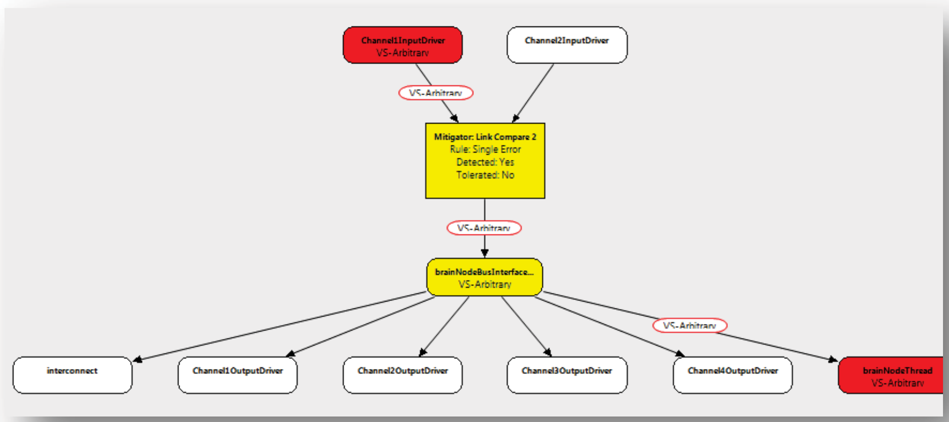


Figure 3 A mitigation scenario which assumes the error source is a skip link.

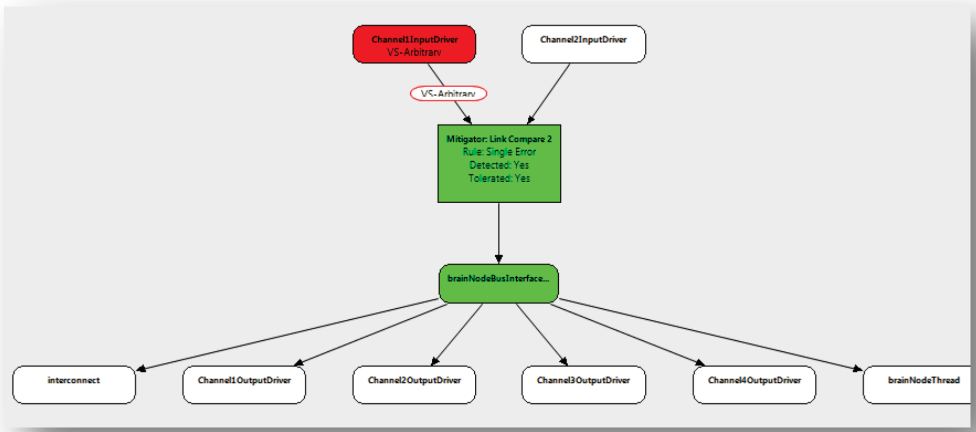


Figure 4 A mitigation scenario which assumes error source is a direct link.

The experiment also revealed several additional opportunities to improve EDICT error modeling capabilities by enhancing the behavioral specifications for components and error effects. These areas include:

- Component error mitigation models could be expanded to distinguish between the inputs over which the mitigator is deployed. EDICT analyzers apply the same mitigation behavior regardless of whether an error propagates via a direct or a skip link. Currently, to ensure that the correct result is covered in propagation analysis both possibilities must be modeled as mitigation results. (See Figure 3 and Figure 4.) This approach can be refined to make the error cases more understandable.
- In the BRAIN protocol, a single-error scenario involves disagreement between the node pair that originated the data. This disagreement results in the data being flagged as untrusted. Currently, this is captured in EDICT by creating a unique error semantic to represent flagged data or messages. The capability to directly represent status flags in propagation could streamline modeling tasks. This would not affect propagation paths, but could result in greater specificity in evaluation of error impact at components.
- Propagation and arrival times are important in a synchronous protocol. EDICT's current approach to error propagation analysis models all possible arrival combinations. This provides coverage of all cases that could result from a system's timing characteristics without requiring error models to specify timing details. The development of timing and behavior specifications in EDICT could allow further filtering of analysis results. This would be especially beneficial in a system with high connectivity between nodes, such as BRAIN.
- Some mitigation is most naturally represented in terms of a component's outputs rather than its inputs. Outgoing mitigation in EDICT can currently be captured with a combination of input mitigators and error transformations in component error models. Extending the current mitigation paradigm to apply to outputs as well as inputs could simplify error mitigation modeling in the BRAIN architecture.

In addition to these considerations pertaining to single-error scenarios, we observed that a number of behavioral aspects of BRAIN protocols are formulated in a manner that is best modeled by an approach using state machines. These include policies intended to identify babbling nodes or other ongoing data problems. Enhancing EDICT error handling capabilities with access to state-based behavioral models would allow analysis of these mitigation protocols and many others.

As a result of these efforts it became clear that current modeling approaches for fault tolerant systems and protocols are insufficient to represent the complexity of the many interdependent fault tolerant services that compose real architectures. Due to the complex nature of the implementations and component specifications, the subtle aspects of the protocols become "lost" in the overall architecture. It therefore becomes very difficult to determine what parts of the architecture provide key elements of the services and how all the service operations interact in a composite manner. This, in turn, causes difficulty in analysis since the problem cannot be properly decomposed, resulting in models that are large, unwieldy and difficult to understand and analyze.

Approach

Having identified problems and highlighting the challenges, we propose foundational concepts for the way forward. System architects rarely design complex architectures as a flat system but instead use abstractions to represent the arrangement and nesting of concepts and functions. When design and analysis models are developed separately there is no assurance that the modeling abstractions properly align with the perspective that system architects used in system composition. Towards this end, standards, protocols, and layering techniques are used to enhance flexibility and reduce complexity of designs. These elements must be considered in modeling representations and must be captured to facilitate understanding and analysis. Current architecture modeling languages are able to capture the structure of systems but do not do a good job of representing how the elements are composed to provide fault tolerant system services to the applications.

Our strategy is to refine the overall modeling approach to enable the definition of architectural layers, methods for layer composition and layer binding to system implementations. We begin by assuming the complexity of the protocol is such that a hierarchical set of relationships exist in the set of operations. This is true for distributed systems that use messaging protocols where a hierarchical relationship exists between the message and physical layers and the protocol operations that take place.

These architectural layers can be modeled individually along with their respective operations. However, in order to model the composite protocol further details are necessary. These include facets related to requirements and available capabilities to achieve the designated requirement. Specifically,

- ***Binding*** - how a layer is connected to other layers through functions and properties and how the layer is implemented by lower level specifications.
- ***Fulfillment*** - how the layer provides the stated functions and properties.

In addition to the structural specification, there are both functional and non-functional properties associated with each individual protocol. The resulting layer is the collective grouping of these properties. Analysis needs to be performed on the layer specification to determine the degree of success in achieving the desired goals.

We will also assume that the properties required for these protocols have been correctly specified. It is beyond the scope of this report to identify the steps to accurately determine these properties as derived from the higher level system goals.

The specific set of properties that are required to elicit necessary architecture details for ensuring proper composition of the overall architecture fall into the following two major categories:

- ***Non-Functional properties*** that are *qualitative* in nature and define characteristics associated with the delivered functionality. Examples of non-functional aspects include properties related to reliability, availability, safety, security. Other examples of non-functional aspects to be considered involving properties that can significantly impact architecture design and analysis include: scalability, flexibility, integrity, interoperability.
- ***Functional properties*** that are *quantitative* in nature and define what functions an abstraction layer delivers. Examples of functional aspects include properties related to communication, resource discovery, synchronization, process group management, detection and reconfiguration, health monitoring, and scheduling.

Composite Properties

Although it is argued that modeling systems hierarchically only makes the proofs for verification more difficult [2], Lamport recognizes this is a reasonable position for systems with *complete-specifications* that describe all the behaviors and the environment that provides the system context (using the notation in the form $S \wedge E$, where S describes the system and E the environment). When a component is reused in multiple ways, it becomes an *open-specification* problem. In this case he points out that it is the specification of the component itself and not the complete system containing it. The result is that, intuitively, the component's specification asserts that it satisfies S if the environment satisfies E. This suggests that the component's open-system specification should be the formula $E \Rightarrow S$.

Another challenge noted in [2] is that the state of practice is such that system designers rarely specify the system abstractions and relationships with the necessary explicitness to enable modeling without significant interpretations by the modeler. We note that even though time has passed since the author's statement, it is still an existing challenge today where not all facets of the system are concisely described to be easily modeled as layers. Therefore when reusability of a layer is the goal, one of the results we aim to achieve is to show via analysis where the design may require improvement as it relates to strong cohesion and loose coupling.

We have identified foundational concepts to build on and point to work in [3] as a good basis that has formally represented concepts similar to those we will employ. Figure 1 illustrates a compositional method referred to as the *extend-constrain-merge* pattern. The goal of this technique is to facilitate separate specification of properties that can then be used in compositional systems.

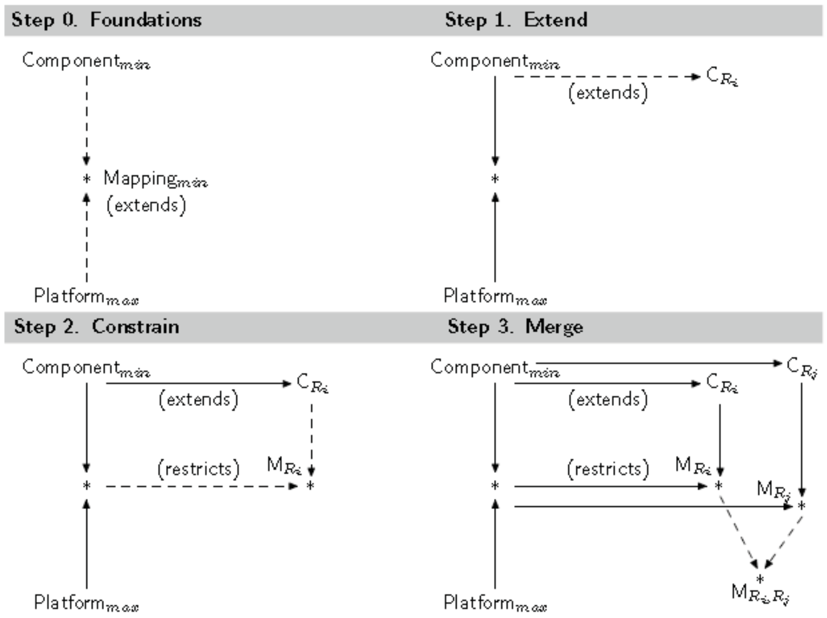
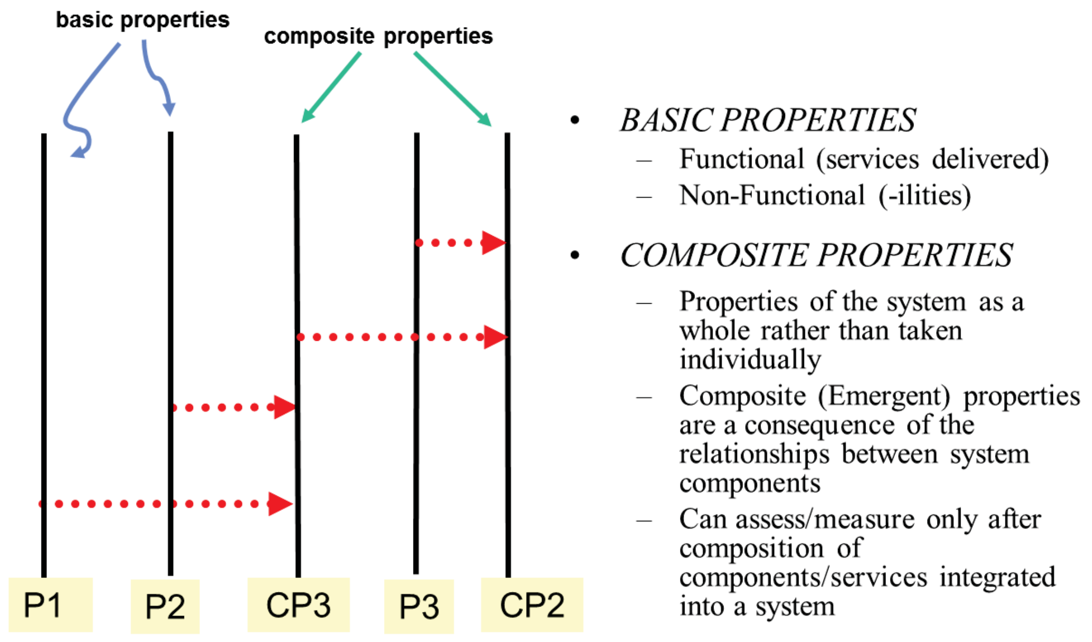


Figure 5 The Extend-Constrain-Merge Pattern [5].

Step 0 defines a minimally-complex abstraction level called *Component$_{min}$* that reflects the minimum data/constraints necessary to represent the high-level architecture as a set of components and also a *maximally-detailed* abstraction of the Platform$_{max}$ underneath. Mapping$_{min}$ extends the combination of Component$_{min}$ with Platform$_{max.}$ Step 1 adds a new requirement to extend the relation and Step 2 introduces constraints that affect the mapping. Step 1-2 ensure requirements are formalized separately and in Step 3 they are composed to formalize their interaction.

In our approach we use these concepts but with important adaptations. Portability and re-use across product lines and projects are important aspects that must be supported by the approach. As a result we are investigating architectures that may not yet be bound to physical platforms and are interested in representing the logical patterns, hierarchies and associated behaviors in the architecture. Consequently, the concepts of components and platform are too rigid for our needs. Instead, we use a logical layering approach where a component should be viewed as a service or, in its most basic form, an operation. Instead of the platform, there would be another service (or operations) that fulfill the requirements of the layer above. The mapping is equivalent to the binding procedure that guarantees the two layers satisfy the requirements.

As we move forward in our research, we can then attach properties associated with the services. Individual services can then be combined to established composite properties that are available to other calling services as shown in Figure 6 and first presented in [14].



- *BASIC PROPERTIES*
  - Functional (services delivered)
  - Non-Functional (-ilities)
- *COMPOSITE PROPERTIES*
  - Properties of the system as a whole rather than taken individually
  - Composite (Emergent) properties are a consequence of the relationships between system components
  - Can assess/measure only after composition of components/services integrated into a system

Figure 6 Property Compositions

Layered Architectures

At a macro system perspective, layered architectures are becoming more commonplace as vendors successfully segment the computing platform, both from a hardware and software perspective. Previous generations were designed as monolithic systems with customized interfaces and limited functionality. Current generation of systems utilize distributed computing with net-centric architectures and standards based components, providing increased opportunities for segmentation of the computing space and creating new layers in the architecture.

From a perspective within a layer, the context, semantic meaning and architecture quality become important as well. When a layer is meant to be used at the lowest level, such as a device driver, this implies a certain context if the layer is to be used successfully. The highest layers are structured to provide domain and application portability across a variety of underlying implementations. Layers in between can have a more uncertain context unless placed in well-defined service architecture such as Autosar or FACE. Regardless of the type of layer, the semantic meaning of the interface is important as well as the degree of non-functional qualities that are supported.

The overall composition of dependable systems therefore necessitates an order to the reasoning of how properties are integrated to realize the target objective. Our approach of identifying property compositions in conjunction with delineations of architecture layers allows an inventory of logical concepts that can be analyzed and checked to determine if the system meets the declared goal. Our goal at this time is to develop approaches for modeling and analyzing these layers and not their optimal synthesis.

The Architecture Layer Concept

Each layer is expected to be a self-contained entity so that it is possible to model an architecture layer without requiring a complete model or understanding of the overall system architecture. Using Lamport's terminology, this is an open-specification problem. For example, vendors like Honeywell may want to provide a communication layer that will be used in another company's system architecture. The architecture layering approach will allow the communication layer to be defined and evaluated with a logical description of the protocols and a set of layer bindings that define the functions and properties provided by the layer and the requirements the layer has for use.

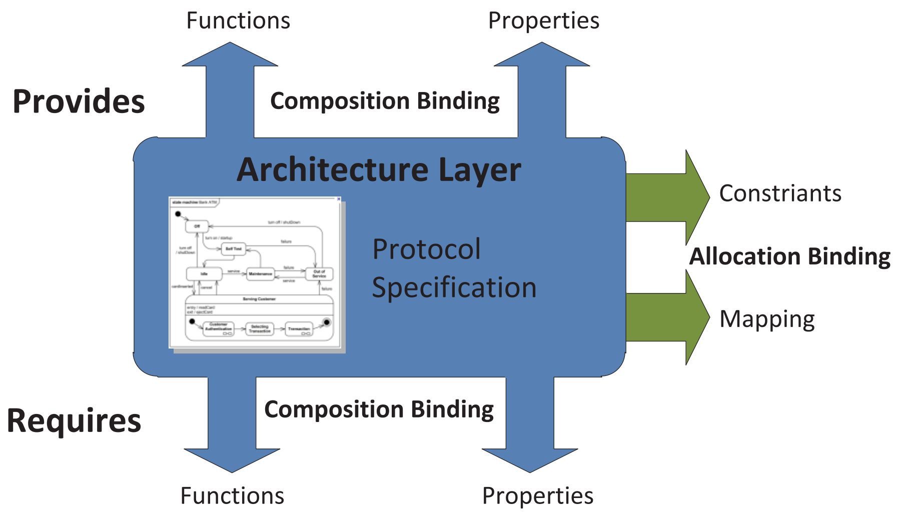


Figure 7 - Architecture Layer Concept

The architecture layer is defined by a set of public interfaces and an internal specification. The public interfaces are decomposed into two types:

- Composition Binding Interfaces – These interfaces allow the layers to be composed into a layer architecture that defines the system functions and how they interrelate.
- Allocation Binding Interfaces – These interfaces allow the layer to be bound to a detailed platform implementation model that describes how the layer is implemented in a system.

The Compositional Binding Interfaces enable the layer to define both the functions and properties that are provided and the functions and properties that are required by the layer from other architecture layers. The *Requires/Provides* relations specified at the layer boundaries make it possible to integrate layers into an overall architecture and evaluate the composition of the architecture before the system is constructed.

- *Requires:* Layers specify what is required for the layer to provide its properties
- *Provides:* Specify the properties that the layer provides

These interfaces will form the basis for formal composition of the layered architecture through matching of Provides and Requires interfaces when layers are bound together.

The *Allocation Binding Interfaces* define how the layer can be bound to an implementation in a platform specific architecture model. These interfaces are used to express the mapping of the layer protocol specification elements to the platform architecture components that define how the protocols will be implemented in a system architecture. In addition to the mapping of protocol elements there are also constraints that can be expressed. These constraints define limitations on the implementation of the protocol that must be adhered to by the platform architecture model. An example of such a constraint is that all participants in the protocol must be allocated to independent fault containment regions for failure independence. In addition to these types of structural constraints the interface may also specify performance or behavior related constraints.

The public interfaces of the layer are supported by an internal specification of the layer functionality. The internal specification is a description of the structure and behavior of the protocols that supply the layer properties in an implementation independent fashion. These models contain only the level of detail required to express the protocols and perform the required level of analysis and verification of the protocol logic.

Modeling Abstractions

The system modeling concepts can be extended to contain two primary levels of abstraction: 1) the Layer Model and 2) the Platform Architecture Model as would be available in a completely specified system. The Layer Model defines the architecture layers and compositional structure for obtaining the overall functionality required in the system. The Platform Architecture Model needs to capture the details of the hardware and software architecture description used to implement the system. Figure 8 diagrams the two levels of modeling and the key bindings between the modeling elements.

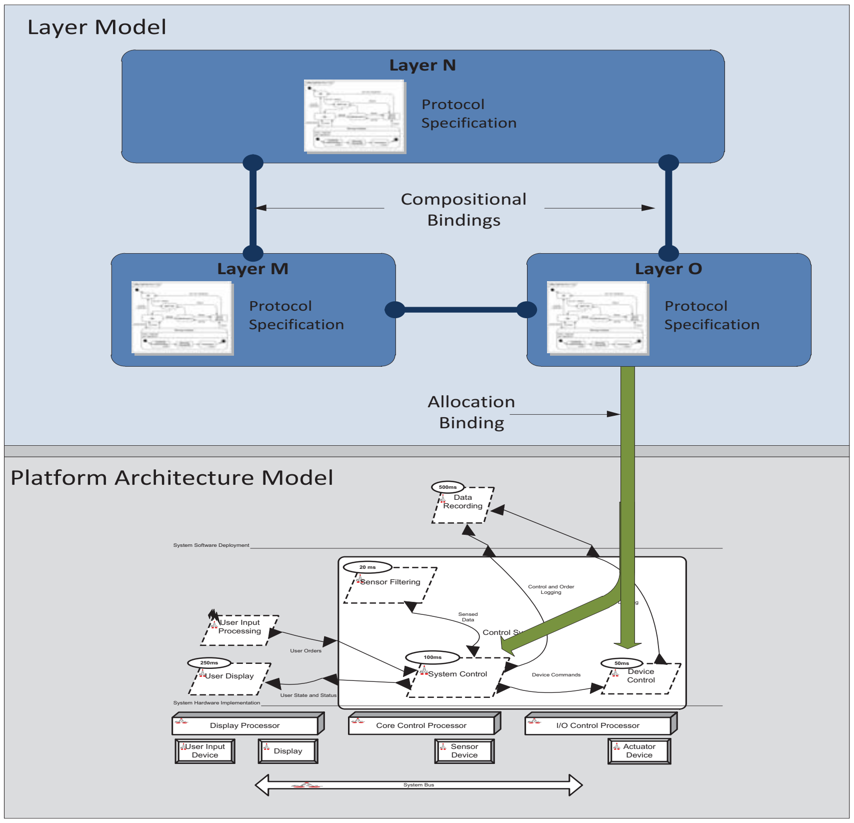


Figure 8 - Model Abstractions

Information related to the compositional bindings is captured to define the relations within the layer model. These bindings are used to define how the layers are composed and how the properties for fault tolerance are built up through the layers.

The next step requires capture of the allocation bindings that are used to bridge the models by specifying how the platform independent layer internal specifications are mapped to the platform specific Platform Architecture Model. In addition to the mapping, these bindings specify constraints on the platform architecture that must be met in order for the layer to provide its specified properties.

The modeling concepts and binding types that are the backbone of our approach are shown in Figure 9 in the context of a high level illustrative example. As illustration of how this technique can be used is shown on the left hand side of the figure where a set of architecture layers are depicted and which are used to compose the WWTG's Reliable Platform functions as applied to Virginia Class Submarine Ship Control System. On the right hand side, the lowest two layers are illustrated in the layered modeling concept with both compositional bindings between architecture layers and allocation bindings between a layer and an idealized platform architecture model.

This example reflects a compositional layering approach to architecture and how dependable system services can be built up through this organized integration of layers.

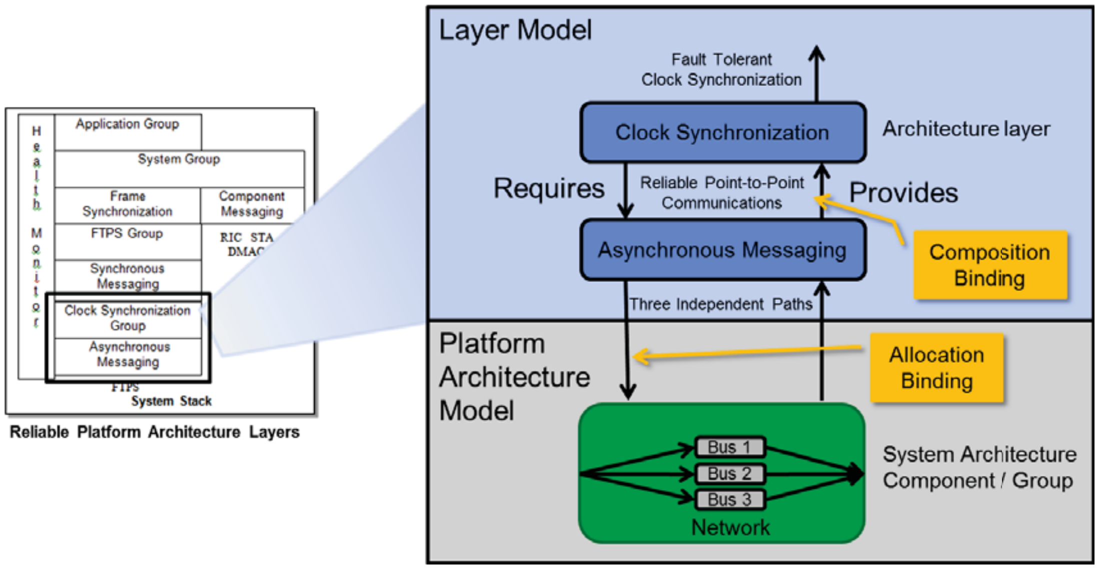


Figure 9 - Illustrative Layer Architecture Example

Figure 9 also illustrates the binding of an architecture layer, through an Allocation Binding, to a Platform Architecture Model that specifies how the layer will be implemented in a given system architecture. In this case the Allocation Binding specifies a structural mapping to the Network component along with a structural constraint that the component provides three independent data paths from source to destination.

Defining Architecture Layers

Abstractions are important to simplifying the understanding and management of the vast amount of details involved in large system architectures. The challenges for the full scale system problem are related to addressing the concerns and preferences of the variety of stakeholders and how they exert influence over the design and acceptance of the system. Our current task focuses on modeling a single layer therefore managing abstractions for complete system architectures is beyond the scope of this effort. Even so, we would still expect the concepts presented here would scale accordingly when modeling a complete architecture. For this single layer study, we focus on stakeholders that participate in the design and verification phases.

Consequently, the starting point would be a representative layer somewhere in the middle of the architecture "sandwich". Using traditional design practices, this layer would comprise many specific details that would not need to be understood or conveyed to a higher level function in the system architecture. An example would be a communication service that is callable by higher level applications, such as one based on the BRAIN communication protocol.

Such a service would be defined by declaring: (a) services performed by the designated layer, (b) the elements of the architecture to be included for its instantiation, and (c) the protocols used at this abstraction level. The first item provides the specification of the services that are available to be called. The second defines the required configuration for the service to exist and be available. The third specifies the protocol to be used when calling the service. These elements are directly supported though our approach using a) Compositional Bindings, b) Allocation Bindings and c) layer internal specifications.

Although these steps appear intuitive and might be expected from a sound engineering design, it is surprising how often they are casually documented and not fully analyzed in a full architectural context. Our goal is to capture the required information, perform checks that flag inconsistencies, gaps and poor abstractions early in the design and analysis phases rather than deferring their detection to the system integration phase. Our goal is to enhance early detection of defects and reduce the cost and effort of producing this documentation, especially in a way that is easily communicated to many system stakeholders.

Granularity of Architecture Layers

The proper granularity of layers can be determined by examining whether the abstraction first appropriately hides the right amount of information. First order checks can be performed by looking at the interfaces and dependencies among layers to ensure a clean and complete interface. Additional checks can assist in assessing whether the abstraction has the proper granularity for the required usage in a given system context. For example, if a call is made to a large (heavyweight) service when only a small operation is required, it may levy unnecessary requirements and introduce timing delay and some refactoring may be recommended [4].

Using a communication service as an example, if a higher layer call only requires unreliable messaging, a reliable messaging service would be overweight and a finer grained set of services would be preferred that allowed both types. Another example would be a heavyweight communication service with an abstraction that begins at the callable service interface (message level) and ends with the inclusion of the physical medium. Alternatively, a finer grained communication service could be specified that separates the logical messaging protocols from the physical media access in a way that enables various combinations for implementing an overall communication service with multiple layers.

When developing multiple layers that work together, it is the objective that each abstraction is cleanly defined so that all embedded operations, components and policies are within the logical boundaries of the abstraction. Exceptions should be analyzed to determine the risks and potential mitigation strategies available to eliminate the condition.

Finally, when multiple abstractions are defined, the goal is to ensure that they properly fit within the correct context of the architecture. For example, an abstraction layer at a higher level should not unnecessarily over-specify details that should be decided at a lower level. In this sense, logical abstractions should be disconnected from the need to know specific logical refinements and physical implementation of services or operations it calls.

Layer Composition

The arrangement of multiple layers into larger systems is a compositional challenge that has been a central problem in computer science since the 1960s[1]. Initially, the focus was placed on developing local operating systems with modularity so that could be customized as needed for different product families or applications. With the development of microprocessors and plug-and-play hardware, great effort was invested to further identify ways that operating systems could be layered and further modularized. Distributed systems also required a more expansive abstraction beyond the context of a single processor and the idea of distributed services took root. With the dawn of the internet, Global Grid and cloud computing the collective set of abstractions became more logically oriented to service abstractions and deferring the details of physical implementation to late binding at run-time.

The incremental development of these concepts has provided some important principles and lessons learned:

P1 Requirements should flow downward.

> *P1.1* A higher abstraction level declares what it needs as a requirement.

P2 Aspect Qualities should flow downward (e.g. safety).

> *P2.1* A lower level abstraction declares what it can provide.
>
> *P2.2* Declarations identify properties with conditions that are assumed.

P3 Computational/Information properties flow upward.

P4 Constraints can flow in both directions.

> *P4.1* Each level may have a design space that introduces a constraint.
>
> *P4.2* The Architecture design effort identifies where these spaces overlap (preserved design space) or do not (eliminated design space).

These principles indicate that there is a flow of constriants between the layers which represent the downward flow of both functional and non-functional (qualities) requirements and the upward flow of component properties that represent their capabilities. These notions are captured by the Compositional Bindings in our approach.

Layer composition is facilitated through a set of layer interfaces we refer to as Compositional Bindings as we outlined earlier in the Architecture Layer Concept section of this report. The compositional bindings are used to describe the functional and non-functional properties that are provided by the layer and similarly the properties required by the layer. The composition of layers is then achieved through a binding/matching process where the compositional bindings between two layers are evaluated for consistency and correctness to ensure that the *requires/provides* relations between the layers are properly satisfied.

In order to fully represent these bindings, the development of a Domain Specific Language (DSL) is necessary to capture the properties that are essential for composition of fault tolerant systems and provide the formal foundation on which analysis of the properties can be performed [5]. There are active research projects [12] [13] that are developing domain specific methods for the representation of requirements and constraints over system architectures and these will serve as useful starting points for a DSL that can capture layer properties.

Layer Bindings

Allocation Bindings are used to specify how the platform independent internal layer specifications are mapped to the platform specific Platform Architecture Model. The bindings provide two sets of relations that are critical for evaluating the implementation of a layer in a platform architecture model:

1. *Layer Mapping* – The layer mapping specifies how the structural elements of the layer are mapped to architecture components.
2. *Mapping Constraints* – The constraints specify any structural or behavior constraints that must be satisfied in order for the platform architecture to properly implement the layer.

These two portions of the allocation binding work in tandem to provide a full description of how the layer is bound to the system architecture implementations and the set of constraints that the platform architecture must meet when that binding is fulfilled. The definition of these binding components allows for a structure that enables architects to capture specifications associated with the bindings and for automated analysis to be performed to check for the satisfaction and integrity of the binding.

Layer Mapping

The layer mapping defines how a layer is bound to a platform specific architecture model that implements the layer protocols. The mapping defines where the elements in the layer protocol description are realized and executed in a platform architecture model such as AADL.

The Layer Mapping must provide traceability between both the structural and behavioral aspects of the layer and the platform specific architecture model.

- **Structural** – traceability relations that associate the architectural components that participate in the layer and the interfaces and flows they use to communicate.
    - *Participant Mapping* – define where the layer participants are implemented in the platform architecture. These elements of the layer specification must be mapped to active components in the platform model such as threads, process or devices.
    - *Message Mapping* – define where the message interactions defined for the layer protocol are implemented and the architecture paths that are used. These elements will be mapped to items such as component interfaces for architecture flows in the platform model.
- **Behavioral** – traceability relations that link layer behavioral specifications with component level behaviors.
    - The layer internal specifications will define the platform independent behavior that is required to provide the layer properties. These behaviors must also be adhered to / exhibited by the platform architecture model components. This mapping may be implicit through the structural aspects of the allocation binding but this aspect is important in the validation of the overall allocation binding.

The Layer Mapping provides a mechanism to easily identify the architectural components that play a role in implementing the layer and providing the serivces specified by the layer. This addresses a critical deficiency in current approaches where distributed protocols and the services they provide cannot be easily distinguished within the architectural model.

Mapping Constraints

As part of the allocation binding the layer may require certain constraints be placed on the structural and behavioral aspects of the layer mapping. These constraints are intended to capture and specify conditions that must exist in the platform architecture model for the allocation binding to be valid.

The constraints can be similarly defined into two general categories:

- **Structural** – specifications that constrain the structural mapping or the relations between structural elements of the platform model. These constraints deal with structural aspects of the platform model such as components and interfaces and their arrangement. The types of constraints that can be specified deal with issues such as number of components, separation/fault independence, component type, interface arrangement and connectivity, etc.
- **Behavioral** – the behavior aspect of the constraints governs requirements related to the behavior of the components and interfaces that are specified in the layer mapping. The behavioral constraints will govern

issues such as: required state behavior, event sequences, schedule or periodicity, interface timing (delay, jitter), etc.

Automated checking can be implemented to ensure that the constraints are met by the layer mapping and the attributes of the platform architecture.

Layer Modeling

Architectural Layers provide a powerful abstraction for the composition of dependable systems and are useful for capturing critical aspects of the design. These abstractions must be realized in an engineering tool with interfaces to standardized architectural specification languages and analysis techniques to have a positive impact on engineering practice. The Error Detection, Isolation and Containment Types (EDICT) tool is a system architecture modeling and analysis tool that is capable of accommodating the architecture layer concepts. In terms of realizing an architecture layer in EDICT, our approach is to continue to expand our Architecture Framework paradigm to integrate the layer concepts with the rest of the architectural context. An architecture layer model defines the internals of a given of a given layer and supports the public interfaces of the layer.

A layer may be modeled in a number of different ways depending on the granularity of representation desired and the nature of the analysis to be performed at this level of abstraction. With respect to layer modeling, we explore here the notion of a model that consists of a Component Model and a Behavior Model.

The layer concept that is presented here can support both top-down and bottom-up modeling approaches. The top-down approach focuses first on the independent definition of a layer and then addresses issues related to the Allocation Bindings and how the layer is implemented in a system. The bottom-up approach utilizes the layer to capture and aggregate the behavior and interfaces of a set of existing architecture components to specifically identify architectural layers and the services and properties they provide to the system. This section outlines the essential modeling elements that are required to support both of these approaches for layer definition.

Structural Model

The purpose of a layer's component model is to establish the structural aspects of the layer. This structural definition includes specifications of both the concrete interfaces of the layer as well as the abstract specification of required and provided properties.

Service Specification

The Service Specification captures information related to the general nature of the service. This includes the category of service (schedule run-time, library, etc.), the intended deployment characteristics (singleton, distributed, etc.), and the selection of a service type that might result in the need for additional information (file system, communication protocol, computational algorithm, etc.). In addition, deployment constraints (in the form of properties) are also specified here. For example, if a service is distributed, the specification can include whether or not there is a minimum dispatch requirement.

Functional Interface Specification

This specification describes the concrete interactions that the layer either:

a) Requires – In order for it to function properly, and
b) Provides – to clients of the service.

The required interactions fall into two categories. The first are those interactions that are required with other layers in order to support the layer in question and allow it to fulfill its obligations. The second are those interactions with layer component *peers* in those cases were the service is a distributed. To this end, the interface specifications may have properties associated with them to designate them as being sourced from another layer or from a peer of the same layer.

These interfaces must be further defined through the association with a Data Specification in EDICT's Data Library. This will facilitate the accurate binding of the interface to a physical interface in the System Architecture model.

Property Specifications

This specification describes the properties that the services either:

a) Requires – In order for it to function properly, and
b) Provides – to clients of the service.

There may be cases, especially with regard to non-functional properties (levels of dependability or security, etc.), where the properties may have *conditions* associated with them. They are not simply Boolean in nature (property present, property not present), but rather the property can change over time (even while the layer remains in the same state) as the result of layer configuration changes. This is especially true with distributed protocols used to define a layer, in which, for example, the degree of fault resilience depends on the number of peers currently participating in delivering the layer services [8].

Behavior Model

The purpose of a layer's behavior model is to capture the way in which the layer will behave and respond to stimulus over time. The behavior definition of a given layer will be based on a state machine. The state machine will serve as a framework for capturing two aspects of behavior:

- **State Behavior -** Describes what a layer does in a defined state, including interactions with externals that are of concern.
- **Transition Behavior -** Describes how a layer transition between states, including those stimulus that trigger transitions.

The state machgine specifiocations will be augmented with scheduling requirements that describe the scheduling characteristics that are required for the modeled behavior to operate. Furthermore, there will be a top-level specification of the behavior definition's "boundary interface." This specification will facilitate the binding of a given behavior model to a given service/layer component that is being designed.

State Behavior Modeling

There are numerous considerations that must be accounted for in modeling state behavior. We must first consider whether or not the state represents functionality that is *stateful* or *stateless*. The behavior of stateless states is governed exclusively by external stimuli. The behavior of stateful states can be more complex, as internal conditions may govern behavior as well as external stimuli.

The second consideration lies in whether or not a given state machine is abstract vs. concrete. Concrete state machines will be used to represent "real" functionality and protocols that operate within the system. Abstract state machines will allow us to reflect system-level properties outward for use in analysis and design verification. Abstract state machines can be related to each other hierarchically, mirroring the hierarchical nature of architecture models at higher levels of abstraction (system, logical, etc.).

Each state in the state machine will also support representations of run-time logic. This could be as simple as one or more guard definitions that contain simple functions for governing state transitions, or references to statements that could facilitate automated generation of verification tool formalisms or source code segments.

Transition Behavior Modeling

The critical aspect to modeling the transition behavior of a state machine is in capturing the *triggers* of state transitions. There are two sources of state transition triggers that we consider:

1. *Internal Decision/State Change* - Internal processing taking place within a stateful state concludes that conditions exist to trigger a transition.
2. *External Stimulus* - The arrival of an external stimulus (date message, event, etc.), or the detection of a monitored condition.

There are several categories of external stimuli that could trigger a state transition. The first is the arrival of a message or event received from another run-time component in the system. Note that this component could be a peer (replica instance) of the given state machine in question. Such interaction between peers is typically required to implement a distributed service or protocol.

While data/event arrival represents a *direct* external stimulus, there are also a series of *indirect* stimuli to be considered. These take the form of *monitored* conditions that may trigger a state change. For example, a state machine may monitor a property that is provided by another component. Should the property's disposition change (provided or not provided) a transition may be appropriate. State changes in lower level state machines might be monitored by higher level state machines, so that when a lower level service moves into a certain state conclusions might be drawn by an abstract state machine at a higher level (thus causing a transition at that higher level).

Scheduling Requirements

The processing that is performed while in a given state might be of an order of complexity that it must be sequenced/scheduled in order to accurately reflect over-all behavior.

Boundary Interface

The interface of the state machine must be rigorously defined to facilitate the binding of the state machine to a given layer model that is being developed. The boundary interface specification will allow EDICT to automate the process of checking intended state machine connectivity.

Layered Architecture Application

As part of the development of this modeling approach we are applying the layered architecture method to a set of challenge problems. These efforts will help to refine the method and determine the modeling techniques that will be used to capture layer behavior, composition bindings and allocation bindings.

As we apply our approach to the challenge problems we should be able to:

- Elicit information in a structured, iterative manner.
- Analyze whether layers are reasonably/sufficiently/correctly specified with respect to declared properties and flows using lightweight, semi-formal and formal techniques.
- Provide capability to extract information for analysis by additional formal tools and ensure the results are properly placed back in the correct architecture context.

These capabilities are essential for performing analysis of existing architecture in addition to composition of new dependable system architecture. The method allows for step-wise structured decomposition and analysis of the problem domain and provides an integration path where the results from formal analysis of fault tolerant algorithms and protocols can be placed into the context of a system design in a concrete way. This coupling of the formal mathematical models and the system architecture specification provides a new a powerful approach that will enhance the design and verification process and the applicability of formal methods to architecture composition.

The initial application of the method and modeling techniques has been applied to the protocol suite that composed the Reliable Optical Bus (ROBUS) [6][7] architecture and the asynchronous triple modular redundant (TMR) system that is defined in the asynchronous case study. We also plan to apply the method to the BRAIN networking architecture to supplement out initial analysis of the protocols through error propagation that was described earlier in this paper.

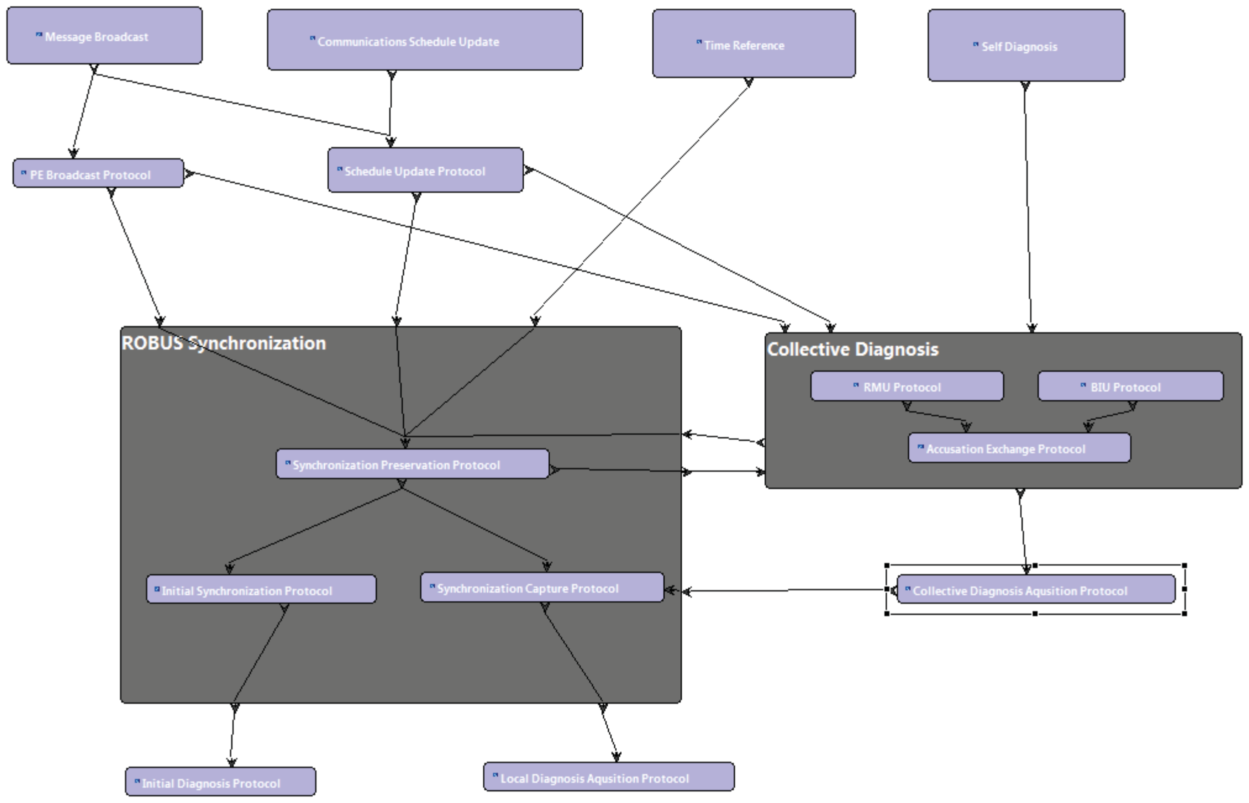

Message Broadcast
Communications Schedule Update
Time Reference
Self Diagnosis
PE Broadcast Protocol
Schedule Update Protocol
ROBUS Synchronization
Collective Diagnosis
RMU Protocol
BIU Protocol
Accusation Exchange Protocol
Synchronization Preservation Protocol
Initial Synchronization Protocol
Synchronization Capture Protocol
Collective Diagnosis Aqusition Protocol
Initial Diagnosis Protocol
Local Diagnosis Aqusition Protocol

Figure 10 - ROBUS Protocol Layers

Our preliminary work on applying the method to dependable architectures has resulted in an initial decomposition of architecture layers and their dependencies that will eventually be defined through composition bindings. In Figure 10 and Figure 11 preliminary layer specifications for ROBUS and for the asynchronous case study are depicted. These breakdowns show the architecture layers that we have defined and the dependencies between the layers that have been identified. In these diagrams the dependencies are shown as directional arrows, but in the future these will be decomposed into compositional binding such that the provides / requires relations are explicitly specified and matched.

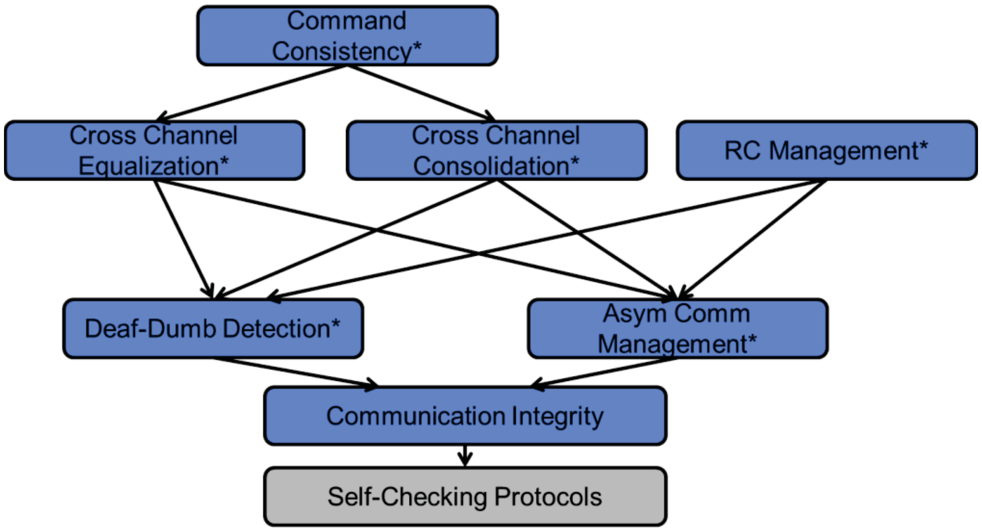


Figure 11 Asynchronous Case Study Architecture Layers

We will continue to breakdown the sample architectures as we evolve the modeling concepts and the methods for applying them. The initial results demonstrate that we have been able to determine layer decompositions for each of the architectures and give clarity to the services that are supplied by the architecture and how they are related. These types of abstractions and relations are missing from traditional Architecture Description Languages and we believe that further development of this approach is warranted.

Conclusions

The design of complex systems relies on the use of protocols for executing functions and coordinating system operations and components. Capturing and analyzing the low-level protocol services and their associated behavior within architecture description languages, such as AADL, has proven challenging to date.

We have developed a foundational approach for modeling layers of protocols that can be composed to provide high dependability services for a system architecture. The approach centers around the definition of an architecture layer, its interfaces for composability with other layers and its bindings to a platform specific architecture model that implements the protocols required for the layer. Our approach of composing layered architectures allows inventory of logical concepts and properties established in the design through the explicit specification of the functions and non-functional properties that the layers provide. These layer specifications are used to compose the overall layer architecture to meet the system requirements for dependability and safety.

The approach allows for incremental validation of layer behaviors and property fulfillment through the use of formal analysis tools on the layer internal protocol specifications and verification of the protocol implementation in the architecture deployment model. These techniques for capture and modeling of highly dependable protocols and architectures in which they are used provide rich source or analysis of system dependability properties and improved capabilities necessary for V&V of dependable and safe systems.

References

[1] Abadi, M., Lamport, L., "Composing Specifications" ACM Transactions on Programming and Language Systems, Vol 15., No. 1, January 1993, pages 73 – 132.

[2] Lamport, L., "Composition: A Way To Make Proofs Harder", Proceedings of the COMPOS'97 Symposium, 1998.

[3] Ethan K. Jackson, Markus Dahlweid, Dirk Seifert, Thomas Santen, Nikolaj Bjorner, and Wolfram Schulte: Specifying and Composing Non-functional Requirements in Model-Based Development, Software Composition 2009.

[4] Stafford, J., McGregor, J., "Top-Down Analysis for Bottom-Up Development", Proceedings of the 2004 Workshop on Component-Oriented Programming (WCOP 2004) held at the European Conference on Object-Oriented Programming (ECOOP 2004), Oslo, Norway, June 15, 2004.

[5] "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification", Object Management Group, Version 1.1 April 2008. http://www.omg.org/spec/QFTP/1.1/PDF.

[6] Torres-Pomales, W., Malekpour, M., and Miner, P., "ROBUS-2: A Fault Tolerant Broadcast Communication System", NASA/TM-2005-213540, Langley Research Center, Hampton, VA. March 2005.

[7] Torres-Pomales, W., Malekpour, M., and Miner, P., "Design of the Protocol Processor for the ROBUS-2 Communication System", NASA/TM-2005-213934, Langley Research Center, Hampton, VA. November 2005.

[8] C.J. Walter, and N. Suri, "The Customizable Fault/Error Model for Dependable Distributed Systems," Journal of Theoretical Computer Science, Volume 290 , Issue 2 (January 2003).

[9] Feiler, P., Glutch, D., Hudak, J., "The Architecture Analysis & Design Language (AADL): An Introduction", CMU/SEI-2006-TN-011, February 2006.

[10] Feiler, P., Seibel, J., Wrage, L., "What's New in V2 of the Architecture Analysis & Design Language Standard", CMU/SEI-2010-SR-008, March 2010.

[11] Obermaisser, R. (Eds.). . *Time-Triggered Communication.* Boca Raton FL: CRC Press, Oct. 2011.

[12] Gilles, O. Hugues, J. "Expressing and enforcing user-defined constraints of AADL models*", Proceedings of the 5th UML and AADL Workshop (UML and AADL 2010)*

[13] Gilles, O., "REAL: Requirement Enforcement Analysis Language", January 31, 2012.

[14] Walter, C., Ellis, P., LaValley, B. "The reliable Platform Service: A Property-Based Fault Tolerant Service Architecture" High-Assurance Systems Engineering, 2005. HASE 2005.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 01- 09 - 2012	**2. REPORT TYPE** Contractor Report	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE Modeling Techniques for High Dependability Protocols and Architecture	**5a. CONTRACT NUMBER** NNL10AB32T
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) LaValley, Brian; Ellis, Peter; Walter, Chris J.	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER 534723.02.02.07.30
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, Virginia 23681-2199	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSOR/MONITOR'S ACRONYM(S)** NASA
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) NASA/CR-2012-217766

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 62
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES
This report was prepared by WW Technology Group under NASA contract NNL10AB32T with Honeywell International, Inc.
Langley Technical Monitor: Paul S. Miner

14. ABSTRACT
This report documents an investigation into modeling high dependability protocols and some specific challenges that were identified as a result of the experiments. The need for an approach was established and foundational concepts proposed for modeling different layers of a complex protocol and capturing the compositional properties that provide high dependability services for a system architecture. The approach centers around the definition of an architecture layer, its interfaces for composability with other layers and its bindings to a platform specific architecture model that implements the protocols required for the layer.

15. SUBJECT TERMS
Fault tolerance; Layered architectures; Model-based development; Property composition; Protocol specification

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	27	(443) 757-5802