This article has been accepted for publication in IEEE Software but has not yet been fully edited.
Some content may change prior to final publication.

Your What is My How: Iteration and Hierarchy in System Design

Michael W. Whalen*, Andrew Gacek†, Darren Cofer†,
Anitha Murugesan*, Mats P. E. Heimdahl* and Sanjai Rayadurgam*

* Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN, USA
{whalen, anitha, heimdahl, rsanjai}@cs.umn.edu

† Rockwell Collins
Minneapolis, MN, USA
{ajgacek, ddcofer}@rockwellcollins.com

***Abstract*—Systems are naturally constructed in hierarchies in which design choices made at higher levels of abstraction levy requirements on system components at lower levels of abstraction. Thus, whether an aspect of the system is a design choice or a requirement depends largely on one's vantage point within the hierarchy of system components. Furthermore, systems are often constructed middle-out rather than top-down; compatibility with existing systems and architectures, or availability of specific components influences high-level requirements. We argue that requirements and architectural design should be more closely aligned: that requirements models must account for hierarchical system construction, and that architectural design notations must better support specification of requirements for system components. We briefly describe work to this end that was performed on the META II project and describe the gaps in this work that need to be addressed to meet practitioner needs.**

***Index Terms*—formal methods; requirements; refinement; model checking; architecture**

I. Introduction

Consider a modern aircraft, such as a Boeing 747. It is an extraordinarily complex system containing over 6 million parts, 171 miles of wiring, and 5 miles of tubing [1]. Viewed another way, it is a complex software and hardware infrastructure containing 6.5 million lines of code distributed across dozens of different computing resources [2]. To make design and construction possible, the components—physical and software—are necessarily organized as a hierarchical federation of systems that interact to satisfy the safety, reliability, and performance requirements of the aircraft.

This hierarchical aspect of design is of crucial importance; design considerations at one level of abstraction, such as how to partition a system into subsystems and allocate functionality to each, determine what the subsystems should do at the next level of abstraction. Requirements at a particular level in the hierarchy are implemented in terms of a set of design decisions (an architecture), which in turn induces sets of requirements on the components of that architecture; this is an idea that spans both physical and software architectures. Yet, we frequently speak of "the requirements" on a system as separate from, and more abstract than, "the architecture" of that system.

Although some requirements engineering techniques (notably KAOS [3], and i* [4]) do support hierarchical decomposition of requirements, these decompositions are in general not bound to the architecture of the system, nor is there a prescribed process for coevolution with architectural models. Therefore, when practitioners derive an architecture to address a systems engineering challenge, there is often little guidance on how the requirements should be decomposed and allocated to architectural components.

II. Iterative Requirements and Architecture

Even in safety-critical systems with well-understood domains, it remains difficult to correctly specify requirements. In previous work involving requirements verification in Model-Based Development, we found that the requirements were almost as likely to be incorrect as the models [5]. For example, one class of errors involves inconsistencies between requirements:

- When button X is pressed, the mode shall be A
- When button Y is pressed, the mode shall be B

These requirements are inconsistent if X and Y can be pressed simultaneously and A and B are mutually exclusive. By constructing and analyzing models, we were able to find such inconsistencies, as well as implicit assumptions about the environment in which the system was to be deployed. In fact, because the models regularly brought to light problems in the requirements, the approach used by the engineers was to iteratively refine models and requirements, using a "model a little, test a little" approach.

Given very large systems (or systems-of-systems), it appears even less likely that the top-level requirements will be correct [6]. Even if top-level requirements are correct, an additional challenge is demonstrating that, given the architectural solution, the hierarchically decomposed requirements are sufficient to meet the system-level requirements. An approach to requirements validation, architectural design, and architectural verification that uses the requirements to drive the architectural decomposition and the architecture to iteratively validate the

An earlier version of this article as a position paper is included in the Proceedings of First International Workshop on the Twin Peaks of Requirements and Architecture, Chicago, 2012.

This work has been partially supported by DARPA/AFRL on project FA8650-10-C-7081, and NSF grants CNS-0931931 and CNS-1035715.

Digital Object Indentifier 10.1109/MS.2012.173 0740-7459/$26.00 2012 IEEE

requirements would be highly desirable. Further, we would like this verification and validation to occur prior to building code-level implementations.

A well-defined set of requirements is discovered through informed deliberations between stakeholders with shared as well as competing interests. We can view the starting point as an incomplete articulation of their key concerns and in the course of these deliberations the participants make rational choices and trade-offs. The quality of the resulting requirements largely reflects how well the participants engaged in this process. Key stakeholders include the systems, safety, and software engineers, whose overriding concern is how to successfully build a system that would meet the resulting requirements. The architecture, we believe, is essential for these stakeholders to understand the main concerns that should inform their positions during the negotiations. It highlights important aspects of "how the system would work?" leaving out the minutiae, helping focus attention on concerns likely to affect system feasibility. Therefore, we assert, it is but natural that in any practical development process, requirements and architecture evolve together.

In this respect, we concur with the Twin Peaks model of Nuseibeh [7], which recognizes that requirements and architecture coevolve, and that this evolution is healthy for creating both a sound architecture and correct requirements. Nuseibeh also points out that system development often starts from candidate architectures that have been used in similar systems. These architectures may restrict the set of achievable requirements, but still be desirable for many reasons, including: (1) familiarity of designers and software engineers with the architecture, and (2) amortization of cost due to the candidate architecture getting refined over several systems, as in e.g., program families. Thus, iteration between architectural models and requirements can better deal with Boehm's sources of uncertainty for requirements [8].

In this paper, we extend this view further and posit that (a) the dynamic model of coevolution induces a static model of interrelationship that ties requirements with architectural elements in an inherently hierarchical fashion, and (b) such a mapping is equally essential for both building and verifying complex systems. The following sections attempt to articulate this position in a more concrete manner.

III. Organizing Requirements

Once systems become sufficiently complex, they are decomposed into subsystems that are created by several distinct teams. Thus, the requirements on the system as a whole must be decomposed and allocated to each of the subsystems. This decomposition touches both requirements and architecture since the structure of the decomposition will affect how requirements are "flowed down" to each subsystem. We believe that requirements can (and should) be organized into hierarchies that follow the architectural decomposition of the system. This organization promotes a natural notion of refinement and traceability between layers of requirements.

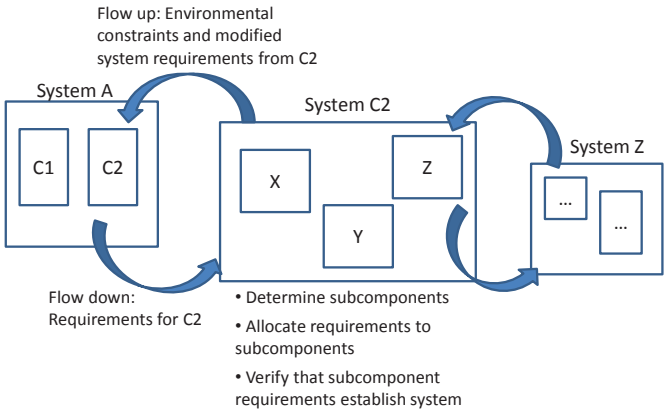

Fig. 1. Interplay between architecture and requirements

By organizing the requirements using the architectural decomposition, we highlight the idea that system decomposition is *both* an architectural and requirements exercise. The act of decomposing a system into components (and then assembling the components into a system) induces a requirements analysis effort in which we need to ascertain whether the requirements allocated to subcomponents in the architecture are sufficient to establish the system-level requirements. Equally importantly, we need to determine whether any assumptions on a component's environment made when allocating requirements to that component can be established. This is shown informally in Figure 1. As we begin to allocate requirements to components, we may find that the architecture we have chosen simply cannot meet the system-level requirements. This may cause us to re-architect the system to allow us to meet the system-level requirement, levy additional constraints on the external environment, or to renegotiate the system-level requirement.

IV. Architectural Models

Architectural models include components and component interfaces, interconnections between components, and requirements on the components (but not their implementations). Thus, the architectural models describe the interactions between components and their arrangement in the system. By annotating them with requirements for component behavior, these models become a means to support iteration between requirements allocation and architectural design.

At the leaf level, component implementations are defined separately using model-based development tools or by traditional programming languages, as appropriate. They are represented in the system model by the subset of their specifications that is necessary to describe their system-level interactions; these specifications may include information about component functionality, performance, security, bindings to hardware, and other concerns.

As we are working on the embedded safety-critical systems, we need an architectural modeling language that can support descriptions of both hardware and software components and their interactions. For this reason, we have been examining the SysML and AADL [9] notations. These languages were

Digital Object Indentifier 10.1109/MS.2012.173 0740-7459/$26.00 2012 IEEE

developed for different but related purposes. SysML was designed for modeling the full scope of a system, including its users and the physical world, while AADL was designed for modeling real-time embedded systems. While both SysML and AADL are extensible and can be tailored to support either domain, the fundamental constructs each provides reflect these differences. For example, AADL lacks many of the constructs for eliciting system requirements such as SysML requirement diagrams and use cases. On the other hand, SysML lacks many of the constructs needed to model embedded systems such as processes, threads, processors, buses, and memory. Our approach has been to use AADL as our working notation and support translation from SysML (with some additional stereotypes for certain components corresponding to AADL constructs) into AADL models.

V. System Verification

In critical systems, significant progress has been made in analyzing the behavior of "leaf-level" components against their requirements. In the 2000s, tools and techniques for unit testing of source code improved to the point where coding errors that escape detection through testing are today relatively rare [10]. During the last decade, Model-Based Development has increased the level of abstraction at which engineers design software components and moved much of the testing forward into the design phase. During that same time period, model checking has become a practical form of analysis that finds errors that testing would miss and does so earlier in the design process [11].

While we have become better at demonstrating that leaf-level components meet their requirements, checking whether component-level requirements are sufficient to demonstrate the satisfaction of higher-level requirements is still an area of ongoing research. Not surprisingly, component integration has become the most important source of errors in systems [6]. In fact, while techniques for specifying and verifying individual components have become highly automated, the most common tools used to specify the complex system architectures in which they are embedded remain word processors, spreadsheets, and drawing packages. Better support for decomposition of requirements throughout the system architecture, and subsequent verification that these decompositions are *sound*, is very important.

In the initial stages of requirements and architectural co-design, this process is relatively informal and fluid. However, for critical systems, this informality can lead to problems. It is often the case that many of the errors in system development manifest themselves in integration; each of the leaf-level components meets its requirements, but these are not sufficient to establish the satisfaction of the system requirements. In order to prevent these integration errors, we would like to perform *virtual integration* in which we can determine whether leaf-level requirements are sufficient to demonstrate satisfaction of system level requirements at arbitrary levels of abstraction.

If the requirements and architecture efforts are based on natural-language requirements and modeling notations lacking rigorous semantics, the reasoning process we promote would closely resemble the *Satisfaction Argument* advocated by Hammond et al. in their outstanding paper "Will it work?" [12]. The Satisfaction Argument, based on the *World and the Machine* model [13], attempts to establish that system requirements hold through an argument involving (i) the specification of the system behavior and (ii) assumptions about the domain of the system. When systems are decomposed, the "domain assumptions" of a subcomponent will likely include assumptions about the behaviors of other subcomponents with which it communicates. Hammond et al. introduce *rich traceability* links to argue that subcomponent specifications together satisfy system requirements using structured (but informal) justifications.

To formalize satisfaction arguments, assume-guarantee contracts [14] provide an appropriate mechanism for capturing the information needed from other modeling domains to reason about system-level properties. In this formulation, guarantees correspond to the component requirements. These guarantees are verified separately as part of the component development process, either by formal or traditional means. Assumptions correspond to the environmental constraints that were used to verify that the component satisfies its requirements. For formally verified components, they are the assertions or invariants on the component inputs that were used in the proof process. A contract specifies precisely the information that is needed to reason about the component's interaction with other parts of the system. Furthermore, contract mechanism supports a hierarchical decomposition of verification process that follows the natural hierarchy in the system model.

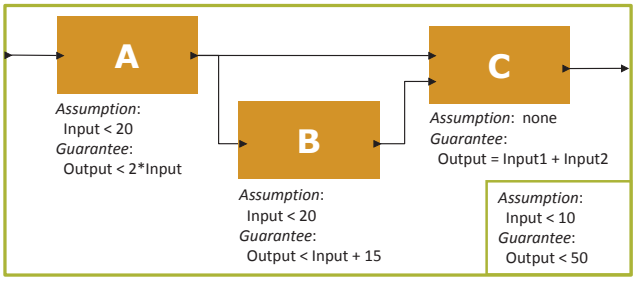

Fig. 2. Toy Architecture with Properties

The idea is that, for a given layer of the architecture, we use the contracts of the subcomponents within the architecture to establish the satisfaction of the system level requirements allocated to that level. A toy example of the idea is shown in Figure 2. In this figure, we would like to establish at the system level that the output signal is always less than 50, given that the input signal is less than 10. We can prove this using the assumptions and guarantees provided by the subcomponents A, B, and C. This figure shows one layer of decomposition, but the idea generalizes to arbitrarily many layers. To create a complete proof, it is necessary to prove that each layer establishes its system level property.

In a recent project, Rockwell Collins and the University of Minnesota have created a tool suite called the Assume Guar-

Digital Object Indentifier 10.1109/MS.2012.173 0740-7459/$26.00 2012 IEEE

antee Reasoning Environment (AGREE), which we briefly describe in the next section. The tools themselves and analysis techniques briefly mentioned here have been previously described in a longer paper [15].

The system-level properties that we wish to verify fall into a number of different categories requiring different verification approaches and tools. At the topmost level, we are interested in behavioral properties that describe the state of the system as it changes over time. Behavioral properties are used to describe protocols governing component interactions in the system, or the system response to combinations of triggering events. Currently, we use the Property Specification Language (PSL) to specify most behavioral properties of components. This allows straightforward formulation of a variety of (discrete time) temporal logic properties.

There were two goals in creating this verification approach. The first goal was to reuse the verification already performed on components. The second goal was to enable distributed development by establishing the formal requirements of subcomponents that are used to assemble a system architecture. If we are able to establish a system property of interest using the contracts of its components, then we have a means for performing *virtual integration* of components. We can use the contract of each of the components as a specification for suppliers and have a great deal of confidence that if all the suppliers meet the specifications, the integrated system will work properly. Thus, we can choose arbitrarily the "leaf level" of components (and their requirements) that we wish to analyze.

Figure 2 illustrates the compositional verification conditions for a toy example. Components are organized hierarchically into systems. We want to be able to compose proofs starting from the leaf components (those whose implementation is specified outside of the architecture model) recursively through all the layers of the architecture. Each layer of the architecture is considered to be a system with inputs and outputs and containing a collection of components. A system S can be described by its own contract (A_S, P_S) plus the contracts of its components C_S, so we have $S = (A_S, P_S, C_S)$. Components "communicate" in the sense that their formulas may refer to the same variables. For a given layer, the proof obligation is to demonstrate that the system guarantee P_S is *provable* given the behavior of its subcomponents C_S and the system assumption A_S. That is, P_S should be derivable as a consequence of C_S and A_S by applying the rules of the logic used to formulate these contracts. Such a *proof*, in effect, assures a successful integration of the contract-conforming components to realize a system that can meet its contract, reducing both the burden and the risk associated with system integration during development.

In our framework, we use *past-time linear temporal logic* (PLTL) to formulate the correctness obligations for systems. Temporal logics like PLTL include operators for reasoning about the behavior of propositions over a sequence of instants in time. For example, to say that property P is always true at every instant in time (i.e., it is "globally" true), one would write $G(P)$, where G stands for "globally". The correctness obligations are the form $G(H(A) \Rightarrow P)$, which informally means, it is always the case that if assumption A has been true from the beginning of the execution up until this instant (A is *historically* true), then guarantee P is true. For the obligation in Figure 2, our goal is to prove the formula $G(H(A_S) \Rightarrow P_S)$ given the contracted behavior $G(H(A_c) \Rightarrow P_c)$ for each component c within the system. It is conceivable that for a given system instance a sufficiently powerful model checker could prove this goal directly from the system and component assumptions. However, we take a more general approach: we establish generic *verification conditions* that together are sufficient to establish the goal formula. In the example, this means that for the system S we want to prove that *Output* $<$ *50* assuming that *Input* $<$ *10* and the contracts for components A, B, and C are satisfied. For a system with n components there are $n+1$ verification conditions: one for each component and one for the system as a whole. The component verification conditions establish that the assumptions of each component are implied by the system level assumptions and the properties of its sibling components. For this system the verification conditions generated would be:

$G(H(A_S) \Rightarrow A_A)$
$G(H(A_S \wedge P_A) \Rightarrow A_B)$
$G(H(A_S \wedge P_A \wedge P_B) \Rightarrow A_C)$
$G(H(A_S \wedge P_A \wedge P_B \wedge P_C) \Rightarrow P_S)$

In general, these verification conditions may be cyclic, but if there is a delay element in the cycle we can use induction over time as in [14]. The system level verification condition shows that the system guarantees follow from the system assumptions and the properties of each subcomponent. This is essentially an expansion of the original goal, $G(H(A_S) \Rightarrow P_S)$, with the additional information obtained from each component.

VI. Scaling To Real Systems

Of course, reasoning about toy examples is neither interesting nor useful for practitioners attempting to build large-scale systems. For the DARPA META II project, we modeled an avionics system architecture involving an autopilot, two redundant flight guidance systems, and a variety of redundant sensors (the top layer of the architecture is shown in Figure 3). Using this model, we proved properties describing limits on the transient commanded pitch behavior of the flight control system, described in detail in [15]. Even given a relatively complex architecture, the time necessary for each compositional analysis was small due to the decomposition of the analysis problem into layers; on the order of 5 seconds for each layer of the avionics system.

An important limitation in the current tool suite is that it can only deal with systems that are *synchronous* with a one-step communication delay between connected components. The synchrony hypothesis in this case means that the components share a global clock. In order to be appropriate for full-scale use, we must accurately support notions of time in our composition framework. This will likely not require any changes to the underlying formalism of composition, but we must

Digital Object Indentifier 10.1109/MS.2012.173 0740-7459/$26.00 2012 IEEE

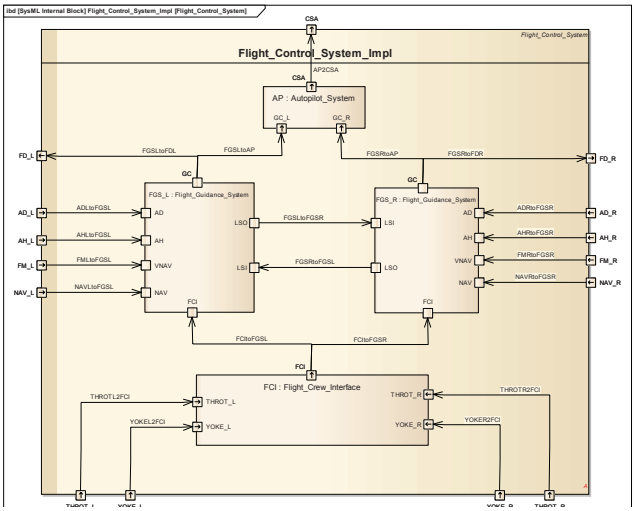

Fig. 3. Dual-Redundant FGS Avionics Architecture

account for the delays induced by computation time, network traffic, and other architectural properties of the model through extraction of this information from the AADL model and incorporation into the formal analysis model. PSL provides some support in that it is possible to add *clocks* to properties representing the instants at which they should be examined. These clocks can be used to describe instants in which a component operates in the context of a larger system. This is the major focus of the next phase of our work, in which we will be modeling more realistic avionics and medical device architectures.

A more general question has to do with the choice of representing components as sets of PSL properties as opposed to other formalisms, such as process algebras . In our work, we have found that declarative properties can be closely aligned with "shall"-style requirements that are traditionally used in avionics systems. However, complex coordination activities between multiple components within an architecture can be difficult to represent using temporal logic. In future work, we hope to examine whether the process-algebraic view of the system can be aligned with our temporal-logic view.

VII. Conclusion

In this paper, we have argued that coevolution of requirements and systems architectures is necessary for the construction of large-scale, hierarchical systems. It is necessary for many reasons, notably (1) systems often start from candidate architectures that have been used in similar systems that constrain system requirements, and (2) for large systems, whether a constraint is viewed as an aspect of (architectural) design or a requirement depends on the level of abstraction from which it perceived—a requirement for a system at a lower level of abstraction is often derived from an architectural design choice at the next higher level of abstraction. Therefore, in our view, requirements decomposition often naturally maps to architectural decomposition of a system.

If we are able to formalize requirements at different levels of architectural abstraction, we have a powerful tool for performing *virtual integration* of components, where we attempt to prove system-level requirements from the requirements allocated to components. This proof can prevent a class of integration errors that occur when subcomponents satisfy their requirements, but these requirements are inadequate to establish system-level requirements. In our experience, this is a significant source of errors in modern safety critical systems. To this end, we have been exploring approaches for embedding requirements into architectural models with support for formalization and automated analysis in our AGREE tool. Although there are several limitations with the current tool—most notably, the lack of support for asynchronous communication—we have used it on substantial models and are in the process of extending the language and tools to support asynchrony.

References

[1] Boeing, "747 fun facts," http://www.boeing.com/commercial/ 747family/pf/pf_facts.html.
[2] R. Charette, "This car runs on code," *IEEE Spectrum*, February 2009.
[3] A. van Lamsweerde, "Engineering requirements for system reliability and security," *Software System Reliability and Security*, vol. 9, 2007.
[4] E. Yu, P. Giorgini, N. Maiden, and J. Myopoulos, *Social Modeling for Requirements Engineering*. The MIT Press, January 2011.
[5] S. P. Miller, A. C. Tribble, M. W. Whalen, and M. P. E. Heimdahl, "Proving the shalls: Early validation of requirements through formal methods," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 4, pp. 303–319, 2006.
[6] R. Lutz, "Analyzing software requirements errors in safety-critical, embedded systems," in *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*. IEEE, 1993, pp. 126–133.
[7] B. Nuseibeh, "Weaving together requirements and architectures," *Computer*, vol. 34, pp. 115–117, 2001.
[8] B. Boehm, "Requirements that handle ikiwisi, cots, and rapid change," *IEEE Computer*, vol. 32, no. 7, pp. 99–102, July 2000.
[9] SAE-AS5506, *Architecture Analysis and Design Language*. SAE, Nov 2004.
[10] J. Rushby, "New challenges in certification for aircraft software," in *Proceedings of the ninth ACM international conference on Embedded software*. ACM, 2011, pp. 211–218.
[11] S. P. Miller, M. W. Whalen, and D. D. Cofer, "Software model checking takes off," *Commun. ACM*, vol. 53, no. 2, pp. 58–64, 2010.
[12] J. Hammond, R. Rawlings, and A. Hall, "Will it work? [requirements engineering]," in *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, 2001, pp. 102 –109.
[13] M. Jackson, "The world and the machine," in *Proceedings of the 1995 Internation Conference on Software Engineering*, 1995, pp. 283–292.
[14] K. L. McMillan, "Circular compositional reasoning about liveness," Cadence Berkeley Labs, Berkeley, CA 94704, Tech. Rep. 1999-02.
[15] D. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional verification of architectural models," in *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)*, A. E. Goodloe and S. Person, Eds., vol. 7226. Berlin, Heidelberg: Springer-Verlag, April 2012, pp. 126–140.

Digital Object Indentifier 10.1109/MS.2012.173 0740-7459/$26.00 2012 IEEE