Department: Head
Editor: Name, xxxx@email

Contracts in System Development: From Multi-Concern Analysis to Assurance with AADL

Jérôme Hugues and Sam Procter
{jjhugues, sprocter}@sei.cmu.edu

Software Engineering Institute, Carnegie Mellon University

***Abstract*— Assuring a safety-critical system requires gathering evidence on a range of concerns (e.g., safety, security, timeliness). Evidence supporting each concern relies on assumptions made about the system and provides guarantees that certain properties are upheld. Contract theory provides a general framework for reasoning using these assumptions and guarantees in a compositional way. We advocate for a broader application of the notion of contracts to support the development of software-intensive systems. In this article, we discuss how system development can benefit from a contract-driven approach and discuss examples of this approach that utilize the Architecture Analysis and Design Language (AADL). Starting from model-based systems engineering foundations, we define classes of contracts that can be defined and verified on both models and software artifacts as a foundation for system assurance.**

■ **MODERN CRITICAL SYSTEMS** are not so much individual systems as they are coordinated—and often distributed—collections of individual components that may have been created at different times, by different manufacturers, for different purposes. These systems, whose failure would cost an unacceptable amount of money, or injure / kill a human, are now being built in increasingly compositional ways: a prime contractor may delegate entire subsystems to subcontractors, who may themselves be composed of many small teams. Thus, managing the systems engineering process becomes an exercise in careful delegation, and eventual integration, of requirements, components, and analysis or test results. This is made more challenging by the range of concerns critical systems must address: not only must they be functional, but also safe, secure, and timely.

The increased complexity and sophistication of modern systems is one of the primary motivating forces behind the component basis used in modern systems development. This complexity is typically managed through decomposition of not only the system itself, but also the systems engineering process. Once the system creation process has been allocated to teams, those teams can develop and refine specialized engineering artifacts. Delegation relies on a notional contract: a team must deliver an item. If one assumes that the input

Digital Object Identifier XXXXXXXX

requirements, interfaces, or specifications are correct, then there is a guarantee that the team will deliver the correct parts of the system.

Over the past decade, Model-Based Engineering (MBE) has evolved to support system decomposition across multiple concerns: structural, behavioral, timing, etc. It has been complemented with analysis capabilities to support validation and verification. MBE methodologies and processes rely on the precise specification of assumptions and guarantees made between different components, teams, levels of the system hierarchy, and other elements of the systems engineering process, in other words, *Contracts*. Yet, this concept remains mostly implicit. We claim contracts should be a visible and explicit concept in order to adequately reason about and compose these various artifacts.

In the following, we first review contracts as they appeared in Software Engineering. Then, we expand the notion of contracts to Systems Engineering and Model-Based Systems Engineering (MBSE). Then, we propose a taxonomy of contracts supported by toolchains and validated by industrial case studies.

CONTRACTS IN SOFTWARE ENGINEERING

The notion of contracts in software engineering emerged as a metaphor to highlight the guarantees software must deliver provided the assumptions it makes on its input parameters are met. It received traction first in the Eiffel language by B. Meyer [1] and has now been incorporated into other programming languages such as C with ACSL, or Ada with SPARK2014. A contract can also be associated with a verification procedure to ensure that a contract actually holds.

For instance, Beugnard et al. [2] have shown how to define contracts and verification procedures for component-based software-engineering. They define multiple categories of contracts: basic and behavioral contracts extend the basic notion to include the observed behavior of components at their interfaces. Synchronization and quality-of-service contracts define system level properties that must be met, e.g., order of operations or timing specifications. There are a range of techniques for validating these contracts, which are used depends on the type of contract and can range from static techniques such as theorem proving to dynamic techniques like runtime verification.

An important lesson from contracts in software engineering is that a contract is made of three parts: a pair of elements, the expression of a property between these two elements, and a verification method that evaluates the contract.

This general idea can be extended to system design: contracts refer to statements of expectations that are written in a language most appropriate to facilitate communication between the involved parties, in addition to verification activities. Formal or semi-formal languages should be preferred to enable and benefit from some level of automation.

SYSTEMS ENGINEERING: ARTIFACTS

Before we discuss contracts in a Systems Engineering setting, let us first introduce a notional systems engineering process; see Figure 1. This process is inspired by the ISO15288 standard [4].

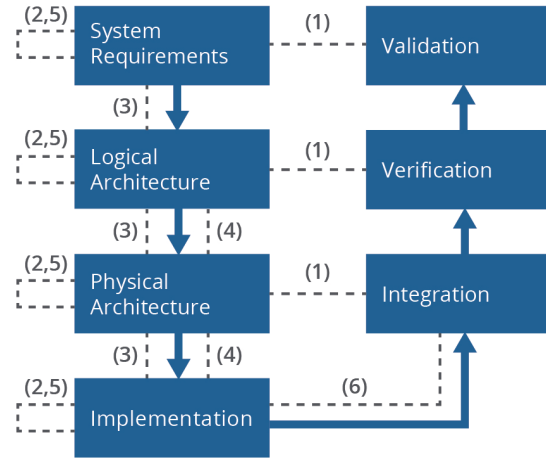

Figure 1: Notional System Development Process. Boxes represent activities; solid, blue arrows the typical progression of the process; and numbered, dashed lines contracts between activities (See section "Six Types of Contracts")

Figure 1 assumes that there are some *System Requirements*, which are used to generate a *Logical Architecture* that defines the abstract functions supported by the system, their interfaces, and interconnections. This architecture is then refined into a *Physical Architecture* that provides a concrete design solution in terms of hardware and software platforms and components. Elements in this architecture are then *Implemented*: created, integrated, verified, and

validated against the requirements. Each artifact of the process is checked for internal consistency and conformance with the previous stage.

In a model-based approach, architectures would be captured as models, e.g., SysML, UML, or the Architecture Analysis and Design Language (AADL). In the following, we assume AADL is used. AADL is an SAE standard for expressing the architecture of software-intensive safety-critical systems. It has been designed as a solution to the ever-increasing design complexity of DoD systems [5].

Architectural Models are amenable to automated examination by various *Analysis Tools*, which take the model as input and then produce output describing the modeled system, i.e., various functional (e.g., correctness, timeliness) or non-functional (e.g., safety, security) properties we can reasonably expect the constructed system to possess.

Architectural models consist of subcomponents, their hierarchical decompositions, interconnections between them, and descriptions of their behavior (both in nominal conditions and in the presence of errors). A special class of analyses, *Conformance Checks*, verify conformance to standards or practices relevant to the domain (e.g., airworthiness for an avionics system). Individual subcomponents can then be further refined, potentially down to the implementation level, whereas other analyses may be performed to assess other properties. Finally, as architectural models essentially specify a component's interface, low-level implementations can be checked against this interface to ensure conformance.

Hence, multiple verification activities can occur at different stages of a system development process. In the following section, we show how they collectively support system assurance and can be expressed as contracts.

SIX TYPES OF CONTRACTS

At the SEI, our team has been leading the development of the AADL standard since its inception. We maintain its reference implementation, the Open Source Architecture Tool Environment (OSATE). AADL and OSATE aimed to address large-scale system integration issues that arise when building software-intensive systems, e.g. for the space or avionics domain. We have shown that these issues can be solved by properly defining and verifying the properties that exist across and within hierarchical "layers" of modeling [6], in other words: contracts between logical architectures, physical architectures, and their implementation.

In Figure 1, we propose six classes of contracts that support activities in system design. In the following, paragraph numbers refer to these labels. We classify contracts by the system development artifacts they pair, i.e., informal requirements, models, analysis tools, and code. A contract may also pair an artifact with itself. The classification follows our experience using AADL and developing OSATE: in addition to defining contracts, we discuss associated verification methods and tool support.

(1) Requirement allocation and verification Requirements are developed and allocated to elements in the system. Designers can define requirements and allocate them to AADL model elements using the Architecture-Led Incremental System Assurance (ALISA) toolkit [7]. In ALISA, each requirement is associated with a set of components that it satisfies and a verification method that defines how it is satisfied (e.g., by review, executing a tool). So, this type of contract is a connection from an informal requirement to a set of elements that satisfies the requirement. This connection can be an association to some components, or the evaluation of some other contracts.

Requirements and architectures can both be organized as a hierarchy. An ALISA verification plan orchestrates the verification of requirements at any level in these hierarchies: requirements on the system as a whole, those associated with physical or logical architectures, or a component of those architectures.

(2) Analysis An AADL model can be analyzed for specific properties, e.g., performance, latency, etc. Generally speaking, there exists an implicit contract between an analysis and a system model: for an analysis's output to be valid, the model must meet certain assumptions; if those assumptions aren't met, at best the analysis won't run or at worst its output could be invalid. These contracts are defined on model entities at a given level of abstraction, e.g., the physical architecture.

In [8], we have shown how to make these analysis contracts explicit for AADL and other notations. Note that these contracts are subtly different from the others discussed here in that they evaluate whether an analysis can run and produce meaningful, correct output. Thus, rather than involving two artifacts of the system development process, these contracts are between the tooling used for system development and one process

artifact. Such contracts provide the assurance that the analysis results can be trusted.

(3) "Vertical" Integration This type of contract enables reasoning about if two models which describe the same system element at different levels of abstraction and using the same notation are compatible. These two models could be part of the system's decomposition, e.g., between a logical and physical architecture, or within a hierarchy of components in one of these architectures. Such contracts are often part of the modeling notation itself, e.g., AADL defines legality and consistency rules on what a valid model refinement is. They may also rely on interface contracts that specify the observable behavior of a set of components that must be implemented. In this case, model checking or runtime verification can used as verification techniques [9].

(4) "Horizontal" Integration This type of contract enables reasoning about if two components, at the same level of abstraction, can interoperate. Assuming a component receives valid inputs, one can guarantee it produces valid outputs. In [10], Liu et al. apply and verify assume/guarantee contracts on AADL models through Resolute (static contracts) and AGREE (behavioral contracts).

We note that (3) and (4) are similar to the contracts defined by Beugnard et al. in [2].

(5) Conformance Some special types of contracts rely on or verify that a given (sub)system conforms to an external standard or documented best practice prescribed by the requirements, e.g. style guidelines or other forms of architectural guidance inherited from domains of knowledge like safety or cyber-security. These contracts assure a model conforms to specific patterns and operates on a model itself, similar to analysis contracts. In [11], we have used Resolute to show conformance of an AADL model to ARINC653, other experiments were conducted to other standards for security or real-time performance.

(6) Interface / Implementation Of particular interest are vertical integration contracts involving a model and its software implementation, or more generally two models expressed using different notations. They are more complex than contracts defined in (3) as there is a complexity gap between a model and its realization that may hinder verification efforts. Demonstrating that an implementation preserves properties established on models requires care, consider the substantial effort involved in creating certified code generators like QGen [12]. In [13], we demonstrate how to generate code from an AADL architectural model to Ada/SPARK2014 for general distributed real-time systems. Conformance to both architectural guidelines and behavioral contracts are first assessed in AADL using Resolute contracts, then expressed again using SPARK2014 contracts. The static verification of both model-level and code-level contracts ensures the implementation conforms to its model. The usage of different notations requires subsequent activities that ensure the same properties are verified at both levels.

CONTRACTS IN SYSTEM ASSURANCE

Contracts thus support the system development process's progression from abstract to more detailed specifications and implementations. Importantly, they also align with the reverse: the consolidation of system development artifacts as the system moves up the "right side" of the V-Styled system development lifecycle (see elements on the right half of Figure 1). Horizontal (4) and Conformance (5) contracts help expedite unit testing, i.e., testing a particular component for its internal behavior against its interface. Then, Horizontal (4) and Vertical (3 and 6) contracts support the integration testing across the various subsystems' layers. Finally, requirement verification (1) supports validation testing and contributes to system assurance. Analysis contracts (2) are used across all engineering steps to ensure the models are correctly analyzed.

At the SEI, we have applied these classes of contracts using AADL and the OSATE toolset for a variety of systems, addressing safety, security, and performance concerns along with an evaluation of the impact on project execution [6]. Hence, contracts serve as a common framework to express assumptions and guarantees of system development elements. This lets them form the foundations of arguments for many system properties.

CONCLUSION

The idea of contracts permeates modern systems development. This article took as its goal both to illustrate the diversity of concerns in system assurance and how contracts can provide a common approach to structure verification and validation activities.

Indeed, simultaneously considering the multiple concerns that critical systems are expected to address is prohibitively difficult for systems of any useful scale,

so rigorous statements of assumptions and guarantees are necessary. We have proposed a mapping of system development activities to contracts and illustrated this mapping through multiple examples conducted at the SEI.

ACKNOWLEDGMENT

Copyright 2022 IEEE.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. DM22-0164

■ REFERENCES

1. Meyer, Bertrand: Design by Contract, Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986
2. Beugnard, Antoine, Jean-Marc Jézéquel, and Noël Plouzeau. “Contract Aware Components, 10 Years After.” Electronic Proceedings in Theoretical Computer Science 37 (October 12, 2010): 1–11.
3. A. Beneveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, K. G. Larsen, “Contracts for Systems Design: Theory,” *Inria Renne Bretagne Atlantique*, RR-8759, pp. 1-86, 2015
4. ISO/IEC/IEEE ISO15288 Systems and Software Engineering — System Life Cycle Processes. 2008.
5. Boydston, Alex, Peter Feiler, Steve Vestal, and Bruce Lewis. “Architecture Centric Virtual Integration Process (ACVIP): A Key Component of the DoD Digital Engineering Strategy.” In *22nd Annual Systems and Mission Engineering Conference*, 2019.
6. Hansson. Jörgen, Helton. Steve, and Feiler. Peter, "ROI Analysis of the System Architecture Virtual Integration Initiative," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU/SEI-2018-TR-002, 2018
7. John D. McGregor, David P. Gluch, and Peter H. Feiler. 2017. Analysis and Design of Safety-critical, Cyber-Physical Systems. *Ada Lett.* 36, 2 (December 2016), 31–38.
8. Brau, Guillaume, Jérôme Hugues, and Nicolas Navet. “Towards the Systematic Analysis of Non-Functional Properties in Model-Based Engineering for Real-Time Embedded Systems.” *Science of Computer Programming* 156 (May 2018): 1–20.
9. Benveniste, Albert, Benoit Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas Henzinger, and Kim Guldstrand Larsen. “Contracts for System Design,” n.d., 68.
10. Liu J., Backes J.D., Cofer D., Gacek A. (2016) From Design Contracts to Component Requirements Verification. In: Rayadurgam S., Tkachuk O. (eds) NASA Formal Methods. NFM 2016. Lecture Notes in Computer Science, vol 9690. Springer, Cham.
11. J. Hugues and J. Delange, “Model-Based Design And Automated Validation Of ARINC653 Architectures using the AADL,” in *Cyber-Physical System Design from an Architecture Analysis Viewpoint : Communications of NII Shonan Meetings*, S. Nakajima, J.-P. Talpin, M. Toyoshima, and H. Yu, Eds. Springer, 2017, pp. pp. 33–52.
12. T. Taft and M. Bordin, "Towards a lean tool qualification process: Digital avionics systems conference," *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, 2014, pp. 1-40, doi: 10.1109/DASC.2014.6979679.
13. J. Hugues, “A correct-by-construction AADL runtime for the Ravenscar profile using SPARK2014,” Journal of Systems Architecture, (123), 2022,

Jérôme Hugues is Senior Researcher at the Carnegie Mellon University/Software Engineering Institute in the Assuring Cyber-Physical Systems team. His research interests focus on the model-based design of software-based real-time and embedded systems, their semantics and their implementation. He is a member of the SAE AS-2C committee working on the AADL and a member of IEEE and the IEEE Computer Society.

Sam Procter is a Senior Researcher at the Carnegie Mellon University/Software Engineering Institute and leader of the Model Based Engineering Initiative. His research interests focus on tool support for system safety, particularly those with an architecture-centric approach. He is a member of IEEE and the IEEE Computer Society’s Technical Council on Software Engineering.