

Predicting Feature Interactions in Component-Based Systems

Judith Stafford and Kurt Wallnau
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA

+1 412-268-5051 +1 412-268-3265
jas@sei.cmu.edu kcw@sei.cmu.edu

ABSTRACT

Software component technologies support assembly of systems from binary component implementations that may have been created in isolation from one and another. While these technologies provide assistance in wiring components together they fail to provide support for predicting the quality and behavior of configurations of components prior to actual system composition. We believe that all quality attributes manifested at runtime are emergent properties of component interactions, and hence arise as a consequence of planned, or unplanned, interactions among component features. In this paper we discuss the affinities among software architecture, software component technology, compositional reasoning, component property measurement, and component certification for the purpose of mastering component feature interaction, and for developing component technologies that support compositional reasoning, and that guarantee that design-time reasoning assumptions are preserved in deployed component assemblies.

1. Introduction

Software component technologies provide a means for composing systems quickly from pre-compiled parts. Technologies such as CORBA and COM have been developed to support composition of components that are created in isolation, perhaps by different people in different environments and in different languages. However, current component-based technologies do not support reasoning about system quality attributes, e.g., performance, reliability, and safety.

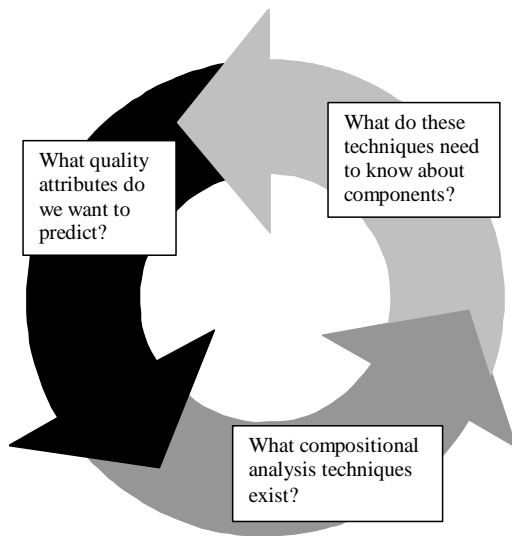
The quality of a software system is, in part, a function of the degree to which its features interact in predictable ways. Users view systems from the perspective of system features whereas developers view systems in terms of functional decomposition into components. The former is a view in the problem domain; the latter is associated with the solution domain. Turner et al. study the relationship between these two domains as they define a conceptual framework for feature engineering [23]. Quality attributes such as performance, reliability, and safety are emergent properties of patterns of interaction in an assembly of components. Ultimately, all such patterns of interaction depend upon one or more features. Therefore, many critical system quality attributes are expressions of component feature interaction. Indeed, a failure to achieve system quality attributes may be attributable to unexpected feature interaction. We suggest that predicting and ensuring system-level quality attributes and controlling component feature interactions are closely related. Moreover, we contend that the solution to both problems (to the

extent they are distinct) will be found in the form of compositional reasoning. Informally, compositional reasoning posits that if we know something about the properties of two components, $c1$ and $c2$, then we can define a reasoning function f such that $f(c1, c2)$ yields a property of an assembly comprising $c1$ and $c2$.

Many would argue that compositional reasoning is the holy grail of software engineering: a noble but ultimately futile quest for an unobtainable objective. This argument usually has as its unspoken premise that only a fully formal and rigorous $f(c1, c2)$ will do. If we accept this premise, then progress will indeed be slow. Instead, we suggest that it is possible to adopt a more incremental approach that involves many levels of formality and rigor. To begin, we suggest that three interlocking questions must be answered:

1. What system quality attributes are developers interested in predicting?
2. What analysis techniques exist to support reasoning about these quality attributes, and what component properties do they require?
3. How are these component properties specified, measured, and certified?

Since compositional reasoning ultimately depends upon the types of component properties that can be measured, these questions are interdependent. Therefore, answers to these questions are mutually constraining. Further, answering these questions will be an ongoing process: new prediction models will require new and/or improved component measures, which will in turn lead to more accurate prediction, and to demand for better or additional prediction models.



The objective of our work in predictable assembly from certifiable components (PACC) is to demonstrate how component technology can be extended to support compositional reasoning. To do this, PACC integrates ideas from research in the areas of software architecture, trusted components, and software component technology.

The rest of the paper is organized as follows: We begin by describing two areas of related work, architecture-based analysis and component certification. The former deals with issues antecedent to compositional reasoning, the latter with issues of component trust and specification.

We then describe a reference model for using component technology to link compositional reasoning with component certification, and close with a summary of our position.

2. Background and Related Work

The ideas of architectural analysis and component certification are not new but, to the best of our knowledge, their integration is. In this section we describe prior work in these areas and discuss their relationship to our work on predictable assembly.

2.1 Architectural Analysis

Software architecture-based analysis provides a foundation for reasoning about system completeness and correctness early in the development process and at a high level of abstraction. To date, research in the area has focused primarily on the use of architecture description languages (ADLs) as a substrate for analysis algorithms. The analysis algorithms that have been developed for these languages have, in general, focused on correctness properties, such as liveness and safety [2,10,14,16]. However, other types of analysis are also appropriate for use at the architecture level and are currently the focus of research projects. Examples include system understanding [13,21,27], performance analysis [3,20], and architecture-based testing [4,24]. One still unresolved challenge for architecture technology is to bridge the gap between architectural abstractions and implementation. Specification refinement is one approach that seeks to prove properties of the relationship between abstract and more concrete specifications, either across heterogeneous design notations [8] or homogeneous notations [17].

2.2 Component Certification

The National Security Agency (NSA) and the National Institute of Standards and Technology (NIST) used the trusted computer security evaluation criteria (TCSEC), a.k.a. “Orange Book.”¹ as the basis for the *Common Criteria*², which defines criteria for certifying security features of components. Their effort was not crowned with success, at least in part because it defined no means of composing criteria (features) across classes of component. The Trusted Components Initiative (TCI)³ is a loose affiliation of researchers with a shared heritage in formal specification of interfaces. Representative of TCI is the use of pre/post conditions on APIs [15]. This approach does support compositional reasoning, but only about a restricted set of behavioral properties of assemblies. Quality attributes, such as security, performance, availability, and so forth, are beyond the reach of these assertion languages. Voas has defined rigorous mathematical models of component reliability based on statistical approaches to testing [26], but has not defined models of composing reliability measures. Commercial component vendors are not inclined to formally specify their component interfaces, and it is not certain that it would be cost effective for them to do so. Shaw observed that many features of commercial components will be discovered only through use. She proposed component credentials as an open-ended, property-based interface specification [19]. A credential is a triple <attribute, value, knowledge>, which asserts that a component has an attribute of a particular value, and that this value is known through some means. Credentials reflect the need to address component complexity, incomplete knowledge, and levels of confidence (or trust) in what is known about component properties, but do not go beyond notational concepts. Therefore, despite many efforts, fundamental questions remain. What does it mean to trust a component? Still more fundamental: what ends are served by certifying (or developing trust) in these properties?

3. PACC Approach

The PACC approach is based on two fundamental premises: first, that system quality attributes are emergent properties adhere to patterns of interaction among components, and, second, that software component technology provides a means of enforcing predefined and designed interaction patterns, thus

¹ <http://www.radium.ncsc.mil/tpep/library/tcsec/index.html>

² <http://csrc.nist.gov/cc/>

³ <http://www.trusted-components.org/>

facilitating the achievement of system quality attributes by construction.

3.1 Premises of PACC

The study of software architectural styles supports the first premise. An *architectural style* is a recurring design pattern, usually expressed as a set of component types and constraints on their allowable interactions [1,7]. Architectural styles provided the first link between structural design constraints and system properties. For example, the *pipe and filter* style yields systems that can be easily restructured. However, the link between system-level quality attribute and architectural style is informal and subjective. To better formalize this link, Klein et al. have developed *attribute-based architectural style* (ABAS) [11]. Informally, ABAS associates one or more *attribute reasoning frameworks* with an architectural style. An attribute reasoning framework consists of a response variable, one or more stimuli variables, and an analysis model that links stimuli to response. ABAS is a key foundation for PACC. It provides the conceptual foundation for defining and analyzing the properties of assemblies (the response variables). It also provides the link between system properties and component properties (stimuli variables).

Component technology provides the means to realize ABAS concepts in software and, in fact, the concept of architectural style is quite amenable to a component-based interpretation [4]. In our view, a component technology can play an analogous role to predictable assembly that structured programming languages and compilers played for structured programming—it limits the freedom of designers (programmers) so that the resulting design (program) is more readily analyzed. In one of many possible examples, the Enterprise JavaBeans (EJB) specification defines component types, such as *session* and *entity* beans,⁴ and constraints on how they interact with one another, with client programs, and with the runtime environment. However serendipitous it may be, it is clear that EJB specifies an architectural style. It is our thesis that analogous component technologies can be defined that go still further to include the additional style constraints needed to support ABAS-based reasoning. The result will be component technologies that support design-time quality attribute analysis, and guarantee, by construction, that the assumptions underlying these analyses are preserved in an assembly of components.

At this point in our research, we are noncommittal about what a prediction-enabled component technology should look like. However, we postulate the outlines of such a technology with the following reference model.

3.2 A Conceptual Reference Model for PACC

Component technologies comprise four levels of abstraction. We generally depict this as a layered reference model, but omit the graphic here for brevity. We describe this model beginning with the concrete and work our way up to the abstract:

- **Assembly.** The most concrete level of our reference model comprises a set of components whose resources (features) have been bound in such a way as to enable their interaction.
- **Assembly specification.** At this level we find component specifications in place of components, and specifications of their interactions. It is at this level of abstraction that attribute analysis and

⁴ Components are denoted as *beans* in EJB.

prediction occur.

- **Types.** At this level we specify component and connector types and their features, thereby defining a vocabulary to support design, that is, assembly specification and attribute analysis and prediction.
- **Metatypes.** At this level one defines what it means *to be* a component type, or a connector type, or an assembly type, and define any constraints that must hold for all types to enable attribute prediction.

3.3 Reference Model Instantiations

We have explored two complementary approaches to instantiate the PACC reference model: one that assumes that attribute reasoning models will be integrated into a component technology, and one that assumes the converse. We refer to the first as a component-centric instantiation, and the second as an architecture-centric instantiation. We have validated both approaches with (admittedly simple) proofs of feasibility. For the component-centric instantiation we used the WaterBeans [18] technology augmented with latency prediction. For the architecture-centric instantiation we used a security ABAS for attribute reasoning, and a Web-based enterprise system for the component technology (from the case study found in [25]). Table 1 summarizes the mapping of these instantiations to the reference model.

Table 1: Complementary Instantiations

Model Level	Component Centric	Architecture Centric
Metatypes	Properties shared by all WaterBeans components, e.g., typed ports, connectors, and connection rules. Defined the latency attribute and associated it with the component metatype.	A simple, behavior-less ADL of components, interactions, assemblies, and their properties. Analogous to a simplified meta-model of UML collaboration diagrams.
Types	Component type definitions for CD audio sampling and wave manipulation. Types introduced the additional Boolean property for aperiodic or periodic behavior, and, if periodic, the execution period. A quantitative model for end-to-end latency is also defined here.	Types that represent basic-level categories for analysis of security properties, e.g., peers, trusted computing base, key, cryptographic provider, threat agent, data asset. Each category is mapped to an element in the simple ADL.
Specification	A topology of audio components annotated with their latency attributes; assembly latency prediction occurred here.	Patterns of interaction comprising only basic categories, where patterns exhibit desired security property. Informal rules of attribute preserving pattern refinement.
Assembly	A benchmarked assembly, allowing comparison of predicted versus actual assembly la-	Pattern refinements where each basic category has been refined to (bound to) a more

	tency.	specific category, ultimately grounding in specific component and interaction features.
--	--------	---

4. Closing Thoughts

In closing, we take the position that the identification of feature interactions in complex systems is closely tied to analysis of system-level quality attributes. Quality attributes of systems are a product of properties associated with both the components that comprise a system and their patterns of interaction. Designing systems as assemblies of components based on architectural styles produces systems that are analyzable by design. We are exploring the application compositional reasoning techniques to assemblies of components in order to predict properties of systems. It is our belief that this line of work can support the identification of the potential for feature interaction before actual system assembly.

5. Acknowledgements

This work was supported by the United States Department of Defense.

6. References

1. G. D. Abowd, R. Allen and D. Garlan, Formalizing Style to Understand Descriptions of Software Architecture, *ACM Transactions on Software Engineering and Methodology*, Vol. 4, No. 4, October, 1995, pp. 319-364.
2. R. Allen and D. Garlan, A Formal Basis for Architectural Connection, *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 3, July. 1997, pp. 213-249.
3. S. Balsamo, P. Inverardi and C. Mangano, An Approach to Performance Evaluation of Software Architectures, *Proceedings of the 1998 Workshop on Software and Performance*, October. 1998, pp. 77-84.
4. F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord and K. Wallnau, Volume II: Technical Concepts of Component-Based Software Engineering, Technical Report CMU/SEI-2000-TR-08, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
5. A. Bertolino, P. Inverardi, H. Muccini and A. Rosetti, An Approach to Integration Testing Based on Architectural Descriptions, *Proceedings of the 1997 International Conference on Engineering of Complex Computer Systems*, September. 1997, pp. 77-84.
6. E. Dijkstra, Structured Programming, *Software Engineering, Concepts and Techniques*, J. Buxton et al. (eds.), Van Nostrand Reinhold, 1976.
7. D. Garlan and M. Shaw, An Introduction to Software Architecture, *Advances in Software Engineering and Knowledge Engineering*, V. Ambriola and G. Tortora (eds.), World Scientific, 1993.
8. F. Gilham, R. Reimenschneider, V. Stavridou, Secure Interoperation of Secure Distributed Databases: An Architecture Verification Case Study, *Proceedings of World Congress on Formal Methods (FM'99)*, Vol. I, LNCS 1708, pp. 701-717, 1999, Springer-Verlag, Berlin.
9. G. T. Heineman and W.T. Councill (eds.), *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, Reading, Massachusetts, 2001.
10. P. Inverardi, A.L. Wolf, and D. Yankelevich, Static Checking of System Behaviors Using Derived Compo-

- nent Assumptions, *ACM Transaction on Software Engineering and Methodology*, Vol. 9, No. 3, July. 2000, pp. 238-272.
11. M. Klein and R. Kazman, *Attribute-Based Architectural Styles*, Technical Report CMU/SEI-99-TR-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
 12. M. Klein, T. Ralya, B. Pollak, R. Obenza and M. G. Harbour, *A Practitioner's Handbook for Real-Time Analysis*, Kluwer Academic Publishers, 1993.
 13. J. Kramer and J. Magee, Analysing Dynamic Change in Software Architectures: A Case Study, *Proceedings of the 4th International Conference on Configurable Distributed Systems*, May 1998, pp. 91-100.
 14. J. Magee, J. Kramer, and D. Giannakopoulou, Analysing the Behaviour of Distributed Software Architectures: A Case Study, *Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, October. 1997, pp. 240-247.
 15. B. Meyer, *Object-Oriented Software Construction, Second Edition*, Prentice Hall, London, 1997.
 16. G. Naumovich, G.S. Avrunin, L.A. Clarke, and L.J. Osterweil, Applying Static Analysis to Software Architectures, *Proceedings of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering* Lecture Notes in Computer Science, No. 1301, Springer-Verlag, 1997, pp. 77-93.
 17. J. Phillips and B. Rumpe, Refinement of Information Flow Architectures, *Proceedings of the 1st IEEE International Conference on Formal Engineering Models*, pp. 203-212, 1997.
 18. D. Plakosh, D. Smith and K. Wallnau, *Builder's Guide for WaterBeans Components*, Technical Report CMU/SEI-99-TR-024, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
 19. M. Shaw, Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does, *Proceedings of the 8th International Workshop on Software Specification and Design*, March 1996.
 20. B. Spitznagel, D. Garlan, Architecture-Based Performance Analysis, *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering*, San Francisco, California, 1998.
 21. J.A. Stafford and A.L. Wolf, Architecture-Level Dependence Analysis in Support of Software Maintenance, *Proceedings of the Third International Workshop on Software Architecture*, November. 1998, pp. 129-132.
 22. C. Szyperski, *Component Software Beyond Object-Oriented Programming*, Addison-Wesley, Boston, Massachusetts and ACM Press, 1998.
 23. C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf, A Conceptual Basis for Feature Engineering, *Journal of Systems and Software*, Vol. 49, No. 1, December 1999, pp. 3-15.
 24. M.E.R. Vieira, S. Dias and D.J. Richardson, Analyzing Software Architectures with Argus-I, *Proceedings of the 2000 International Conference on Software Engineering*, June 2000, pp. 758-761.
 25. K. Wallnau, S. Hissam and R. Seacord, *Building Systems from Commercial Components*, Addison Wesley Longman, To Appear July, 2001.
 26. J. Voas and J. Payne, Dependability Certification of Software Components, *Journal of Systems and Software*, No. 52, 2000, pg. 165-172.
 27. J. Zhao, Using Dependence Analysis to Support Software Architecture Understanding, *New Technologies on Computer Software*, September 1997, pp. 135-142.

