

On the Relationship of Software Architecture to Software Component Technology

Kurt Wallnau, Judith Stafford, Scott Hissam, Mark Klein
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA

+1 412-268{3265,5051,6526,7615}
{kcw,jas,shissam,mk}@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 attributes of assemblies of components prior to their actual composition. The study of software architecture, on the other hand, assumes a relationship between the large-scale structure of a software system, canonically expressed in terms of components and connectors, and system quality attributes. While the study of software architecture has produced good results, the translation from architectural theory to implementation today rests upon rather weak-kneed notions of architectural conformance, i.e., that an implementation conforms to its architecture. In short, the gap between what we may denote as abstract, architecture-level components, and concrete, implementation-level components has yet to be bridged. In this position paper, we discuss and illustrate the fundamental affinity between software architecture and component technology. We also outline criteria for their integration, so that the gap between component integration and architectural analysis can be reduced.

1. Introduction

Software component technology provides a means for composing systems from precompiled 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. The assumption is that a component makes known, through its interface, all information that is needed in order to use (and deploy) its services [27]. However, current component-based technologies do not support reasoning about system quality attributes, e.g., perform-

ance, modifiability, reliability, and safety. Instead, engineers must wait until the components have been acquired, integrated, and the system as a whole benchmarked, to determine whether a system meets its quality attribute goals.

Software architecture provides a means for analyzing system designs with respect to quality attributes. Indeed, a premise of software architecture is that quality attributes—and our ability to reason about them—adhere to particular, recurring structural patterns. These structural patterns are often referred to as architectural *styles*, and are usually defined as types of components and connectors and their allowable patterns of interactions [8,19,23]. Thus, a synchronizing concurrent pipeline *style* might be appropriate for systems with stringent latency requirements, since this style supports RMA-type analyses¹ [14]. On the other hand, a three-tiered style might be appropriate for systems where modifiability of business logic is of paramount concern². However, software architecture has not had an appreciable impact on software component technology. We could conjecture as to why this is so, but this would be beside the point we wish to make in this position paper.

Instead, we simply observe that there is a natural affinity between software architecture and software component technology. This affinity is expressed in several ways. First, and most obviously, is the central role of components and connectors as abstraction. While it is true that the *levels* of abstraction are quite distinct, the *kinds* of thing being abstracted are quite similar. Second, the correlation of architectural style, as an abstraction of design constraints, and component models and frameworks has been noted elsewhere [2,4,12,30]. For example, a component model defines style-specific interfaces that are imposed on compo-

In Proceedings of the 6th International Workshop on Component-Oriented Programming (WCOP6), in conjunction with the European Conference on Object Oriented Programming (ECOOP), Budapest, Hungary, 2001.

¹ Rate Monotonic Analysis (RMA).

² We rely upon the evocative names of these styles to convey the main point.

nents, while a framework provides the run-time mechanisms to implement a style’s connectors³. Last, as already noted, software architecture and software component technology have, to date, focused on complementary agendas: enabling reasoning about quality attributes, and simplifying component integration, respectively.

It seems clear to us that the agendas of the software architecture and software components research communities are complementary. Our goal in this position paper is to describe how they can be combined to yield component technologies that enable reasoning about—in terms of either prediction or proof—the quality attributes of assemblies of components. In section 2 we outline a reference model that relates key abstractions of software architecture and software component technology. In section 3 we instantiate this reference model in two ways, first from the perspective of augmenting component technology with architectural reasoning, and second from the perspective of augmenting architectural reasoning with component technology. In section 4 we discuss criteria for successful integration of component technology and architectural reasoning. In section 5 we briefly point to related work.

2. A Reference Model

We posit the following as a strong condition of the integration of architectural reasoning and component technology: that the architectural structure that is used to reason about the quality attributes of a system is very similar to (approaching isomorphism) the structure of a deployed component assembly that implements this system. This condition leads us to the following four-level reference model, and the instantiations of it that follow.

2.1 Level 1: Assembly

The most primitive level of abstraction in our reference model consists of an assembly of components. By *component* we mean a binary unit of implementation that is independently deployed, subject to third-party composition, etc. [26]. By *assembly* we mean that component resources have been bound in such a way as to allow their runtime interaction. We graphically depict the assembly level in Figure 1.

As illustrated, the components C_1 , C_2 and C_3 are composed into assembly a , with the arrowheads depicting the composition of these components. Each component has some property p , denoted p_{C_1} , p_{C_2} , and p_{C_3} . Likewise, the assembly a has a property, denoted p_a . Component and assembly properties are depicted as lollipops. The dashed lines linking p_{C_2} and p_{C_3} to p_a depict a dependency among these attributes. In particular, p_a is (in some un-

specified way) dependent upon p_{C_2} and p_{C_3} .

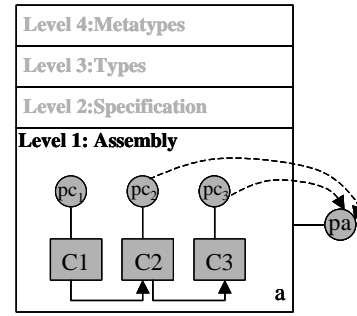


Figure 1: Component and Assembly Properties

Clearly, Figure 1 is a vast simplification of reality, but not so much as to be a gross distortion. For example, we depict the assembly as exclusively comprising components, whereas in practice there will likely be scripting or other “glue.” The point is that assembly a is itself an abstraction, and if it makes sense to say that a has some property (and we believe it does), for example end-to-end latency or level of security, then that property must come from *somewhere*. As depicted in Figure 1, p_a derives exclusively from p_{C_2} and p_{C_3} . Incidentally, in our opinion, connectors (that is, the compositional lines in Figure 1) also have properties, and hence lollipops. However, this refinement is not essential to the current discussion, and is omitted for simplicity.

2.2 Level 2: Assembly Specification

Our agenda is to reason about p_a prior to creating the assembly a , and, ideally, even prior to acquiring components C_1 , C_2 and C_3 . This requires that we work with abstractions of a , C_1 , C_2 and C_3 , that is, their specifications. We depict this specification level in Figure 2.

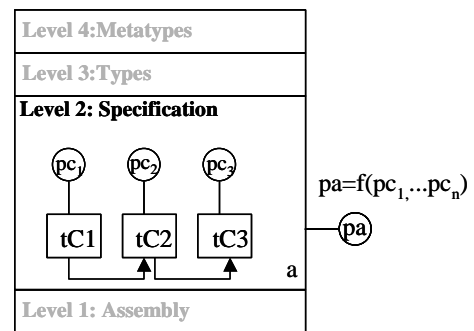


Figure 2: Specification of Assembly

There are similarities and differences between Figure 1 and Figure 2, and both are significant. The obvious similarity is the structural isomorphism between the level-1 assembly and its level-2 specification.

As for the differences between Figures 1 and 2, in place of

³ There are many alternative correlations.

actual components, we have their specification. This distinction is depicted by using clear boxes for components in Figure 2 in place of their (more) opaque counterparts in Figure 1. We also rename the components from C_n in Figure 1, for *component* n , to tC_n in Figure 2, for *component type* n . The properties of these components are likewise abstract, and these abstract properties are analogously depicted as clear lollipops. These clear lollipops represent either specifications of required properties of some conformant component, or postulated properties of some pre-existing component. The assembly property \underline{pa} is now dependent upon abstractions, so it is itself abstract, and hence is also depicted as a clear lollipop.

The most important distinction between Figures 1 and 2 is that property dependency is now expressed functionally as $\underline{pa} = \underline{f}(\underline{pc}_1, \dots, \underline{pc}_n)$. That is, the property of an assembly can be derived from the properties of its constituent components. For example, if each \underline{pc}_n in Figure 2 refers to the property “deadlock free,” then we can deduce, through a compositional reasoning function \underline{f} , that the assembly is likewise deadlock free.

We are not suggesting that all interesting quality attributes are (today) amenable to compositional reasoning—that *would be* a distortion of reality [28]. We recognize that many properties, such as transaction serializability, are not subject to straightforward compositional reasoning, and that in these cases fundamental research is required to ascertain the weakest possible pre-conditions for any \underline{f} that will also produce *useful* results. Our purpose is merely to capture the notion of compositional reasoning in the reference model and, here at least, give an informal definition of what is meant by that term.

2.3 Level 3: Type Specification

The interpretation of $\underline{f}(\underline{pc}_1, \dots, \underline{pc}_n)$ in Figure 2 depends upon the specification of component types tC_1 , tC_2 , and tC_3 , since it is these specifications that contain the properties \underline{f} depends upon. The type level is shown in Figure 3.

Type specifications define the domain of discourse for assembly specifications. As illustrated in sections 3 and 4, the domain of discourse may reflect either a functional or non-functional (i.e., quality attribute) orientation. We note again that, in principle, it would also be consistent to introduce connector *types* at this level, although, again, we do not do so for reasons of simplicity.

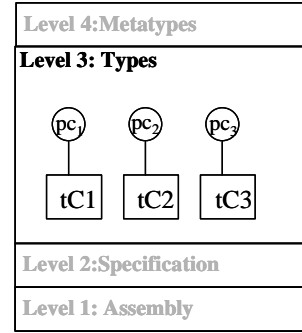


Figure 3: Specification of Types

2.4 Level 4: Metatype Specification

It is necessary to define what it means to be a component, or a property of a component (or a connector) in a particular component model. These constitute the metatype definitions, all types introduced at level-3 must adhere to these definitions. The metatype level is shown in Figure 4.

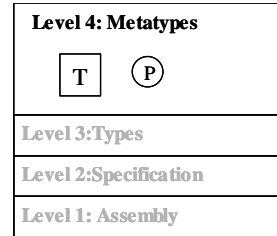


Figure 4: Specification of Metatypes

The metatype level is important to our reference model and its instantiations for the same reason that metatypes are important to a programming language system: it allows us to separate the details of a component system (the metatypes) from instantiations of that system into a domain of discourse, and, further, into assemblies within that domain. As with level-3 type specification, the metatype specification may reflect either a functional or non-functional (quality attribute) orientation.

3. Reference Model Instantiations

In the following discussion, we instantiate the reference model in two ways. The first instantiation assumes the primacy of component technology, and into which attribute reasoning capabilities are integrated. The second instantiation assumes the primacy of software architecture technology, into which component technologies are integrated.

The illustrations that follow are intentionally terse and, in places, simplistic. Our intent is to demonstrate *the modes of integrating* software architecture and software component technology, and *not* the final product of such an integration, *nor* the strengths and limitations of the technolo-

gies being integrated.

3.1 Component Technology Primacy

For this illustration we show the conceptual integration of the WaterBeans [22] component technology with compositional reasoning of end-to-end latency. WaterBeans was developed to allow graphical composition of applications that simulate water quality models. It is a simple component technology that supports a pipe-and-filter style of composition, with typed pipes, multi-input/output single-threaded components, and cyclic, non-preemptive scheduling. Although WaterBeans was developed for water quality modeling, it is sufficiently general to support other application domains. The domain we use in our example is audio input/output sampling and manipulation.

A fragment of the WaterBeans metatype specification, defined as a UML class diagram with associated OCL constraints, is shown in Figure 5.

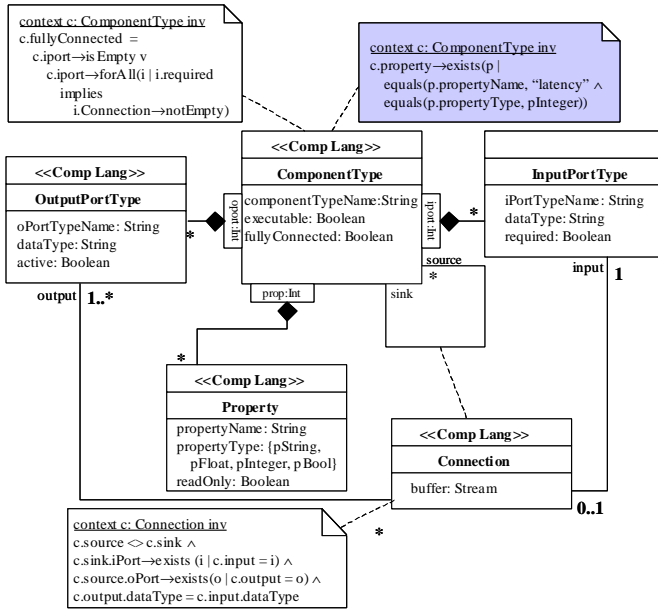


Figure 5: WaterBeans Component Metatype

This class diagram fragment defines what it means to be a component in WaterBeans, and how WaterBeans components are composed into an assembly. Although many details have been suppressed, and although the fragment describes only static properties of components, it nonetheless is representative of the kind of information a metatype specification would include.

The (shaded) annotation in the upper-right of Figure 5 stipulates that all components must possess a latency property, here defined as type pInteger. In the WaterBeans technology, this requires components to respond to queries for the value of this attribute so that (in this case) an

associated compositional reasoning tool can compute an end-to-end latency of a component assembly. We have established the property association, but have not defined the latency property. This requires that we make explicit dynamic aspects of WaterBeans, again, at the metatype level (since this definition would apply to all implementations of WaterBeans). This definition is shown in the UML state transition diagram fragment in Figure 6.

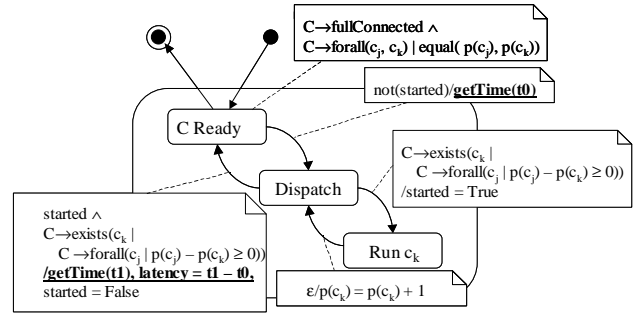


Figure 6: Operational Definition of Component Latency

The function $\underline{p}(c_k)$ returns the number of times component c_k has been run, and C is the set of components $c_k \in C$ in a WaterBeans assembly. This state transition fragment defines latency as the difference in time recorded between two readings of an abstract system clock (see boldface OCL expressions). However, while Figures 5 and 6 provide a basis for measuring a component property. To illustrate the collateral influence of quality attribute reasoning and component technology we describe compositional reasoning about WaterBeans application latency.

The simplicity of WaterBeans requires only a very limited notion of performance analysis. WaterBeans components and runtime are single threaded⁴, thus there is no notion of priority. Additionally, WaterBeans' use of non-preemptive scheduling is quite limiting, since the scheduler is free only to choose the order of component execution, and not the duration of any particular component execution. Therefore, the WaterBeans metatype model does not guarantee the type of architectural properties required by sophisticated compositional reasoning models such as RMA. Instead, such models may be modified (in this case, simplified) to accommodate those architectural constraints imposed by WaterBeans or any other component technology.

We conducted an experiment based on a simplified compositional reasoning model developed specifically for WaterBeans in which we defined assembly latency as the simple sum of all component latencies in an assembly, plus the sum of any latency introduced by the component connec-

⁴ These were some of the details omitted from the metatype definition in Figures 5 and 6.

tors, for example, buffer management overhead. Further, we speculated that if it is known that buffer latency is dominated by component latency, buffer latency can be eliminated from the compositional reasoning model, or replaced by some constant. In our experiment we used the following reasoning model:

$$la = \sum_n lc_n + \sum_m lk_m \quad \text{Compositional Reasoning for Latency}$$

where la is the end-to-end latency of assembly a , lc_n is the latency of the n^{th} component, and lk_m is a constant latency for each m^{th} connector. Since this performance model would depend only on details found in the WaterBeans metatype definitions, it too was associated with the metatype level of the WaterBeans specification.

Our laboratory experiments failed to validate this performance model, but not for the expected reasons.⁵ Instead, we discovered that different *types* of WaterBeans components had different interpretations of latency, notwithstanding its definition in Figure 6. The WaterBeans audio sampling and manipulation domain includes several component types, an instantiation of the metatype defined (partially) in Figure 5: AudioInput, AudioOutput, WaveGenerator, WaveAdder, WaveSubtractor, and AudioViewer. These type specifications inhabit the type level (level-3) of the reference model.

Of these component types, the last four have latency that is independent of the external environment i.e., their latency is a modular property (modulo processor speed and other resources which, in a real time operating system, might be regarded as parameters to the latency property). However, the first two component types are periodic: their latency depends upon a sampling rate derived from an external resource, in this case the audio buffers maintained by the operating system⁶.

Since our purpose is not an exposition of performance analysis *per se*, but rather the integration of attribute reasoning with component technology, we state only the consequence of this component type-specific interpretation of latency:

- The WaterBeans metatype level had to be extended to ensure that each component possesses a Boolean prop-

⁵ We had expected variance due to resource contention and other effects attributable to interference from the (non realtime) operating system underlying WaterBeans. These variances turned out not to be significant given our restrictive benchmarking assumptions.

⁶ We note in passing that it is a weakness of WaterBeans that such *context dependencies* are not made explicit; only dependencies on other WaterBeans components are defined in the metatype level.

erty isPeriodic.

- Each WaterBeans component must use this property to announce whether it is periodic or aperiodic.
- The reasoning model la became markedly more complex, since assembly latency depends upon partial orders of the execution schedule, among other things.
- Since the reasoning model is dependent upon the type (as opposed to metatype) specification, it must be associated with the type level rather than the metatype level of the reference model.

At this point, having made the key points, we suspend our discussion of the integration of compositional reasoning of assembly latency with WaterBeans. To reiterate, the key points of this illustration are:

- The metatype and type levels of the reference model were instantiated from a component technology perspective.
- Attribute (compositional) reasoning models were integrated into the component-centric instantiation of the reference model.
- There is a clear co-dependency between attribute reasoning models and the architectural constraints imposed by a component technology.

3.2: Software Architecture Primacy

We now continue with an illustration that conveys the conceptual integration of *quality attribute design patterns* (QADP)[3], a software architecture technology, with conventional commercial off-the-shelf component technologies. In place of latency analysis, we discuss security analysis.

3.2.1 About This Illustration

Security composes several distinct, but related, quality attributes, such as: confidentiality, integrity, authority, identity, authenticity, and non-repudiability. To keep the discussion manageable, we focus only on confidentiality, which we define, informally, as ensuring that a data asset is disclosed only to authorized parties. It should be noted, however, that security attributes are strongly related to one another, and that, in practice, achieving one (for example, confidentiality) may require achieving others as well (in the case of confidentiality, these others might be identity and authenticity).

In support of the ensuing discussion, we begin by presenting a brief tutorial of QADP. The gist of QADP is contained in Figure 7.

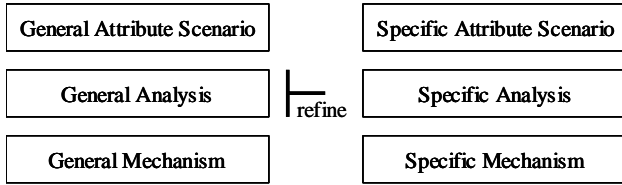


Figure 7: Quality Attribute Design Patterns

A *quality attribute design pattern* is comprised of three elements:

- **Quality attribute scenario:** A use case-like characterization of a quality attribute in terms of a stimulus, a measure of a response, and the design elements involved in producing the response.
- **Mechanism:** A pattern of interaction (collaboration) among design elements that yields the desired attribute measure. (NB: This is a quality attribute-specific refinement of Booch’s definition of *mechanism* [6].)
- **Analysis:** The analytic, formal, or heuristic reasoning that links a mechanism to a scenario’s response measure.

At present, five quality attributes (security, performance, usability, modifiability, availability) have been characterized in this way. The most abstract levels of these characterizations are denoted as *general* scenarios, *general* analysis, and *general* mechanisms. These are applied to a particular design problem through a process of refinement. QADP refinement is not yet well understood. We will touch on this issue in the illustration that follows.

3.2.2 The Illustration

As with the previous illustration, we begin the process of instantiating the reference model from the top—the metatype level. In contrast to the previous illustration, the metatypes we now define are not grounded in component technology, but in software architecture technology. It is not surprising that the resulting definitions are correspondingly more abstract than their component technology counterparts. We use the UML class model in Figure 8 to define the metatypes of architectural components.

The metatype level defines *Component* as something (the abstract class *Element*) that possesses one or more *Properties*, and is related to other components through *N*-ary *Interactions*. Each *Interaction* may have one or more associated *Data Token*. *Component*, *Property*, and *Interaction* are not more fully defined, nor need they be for what follows in the illustration, or in the practical application of the underlying ideas. The metatype level also gives a name, *Mechanism*, to a set of interactions. This is needed since a premise of QADP is that quality attributes adhere, by virtue of some analysis, to

a mechanism. This, in turn, implies that we can treat a mechanism as a named thing.

The type level in this illustration is inhabited by abstractions pertinent to a quality attribute subject (security, performance, and so forth). In the previous instantiation of the reference model (Section 3.1), component types reflected the functional domain of a component assembly, for example the *AudioInput* component type referred to components that introduce audio signals into a *WaterBeans* assembly. In the QADP instantiation, the component types refer to *basic categories* of security analysis. A basic category describes a concept at the most abstract level possible, while still allowing a gestalt perception of the classified objects. For example, ‘boat’ is a basic category, while ‘vehicle’ is too abstract, and ‘sailboat’ too specific [16].

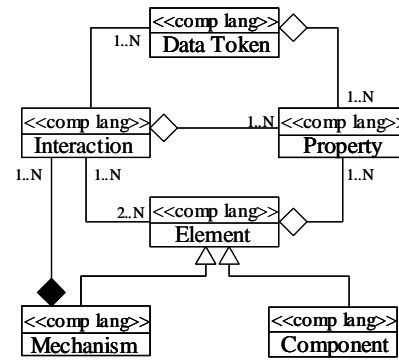


Figure 8: QADP Metatypes

These basic attribute analysis categories are assigned to elements of the metatype level, in effect creating a *de facto* composition language for quality attributes. The prerequisite for constructing such attribute-specific languages is that attribute experts⁷ typically analyze systems in terms of their structures, with structural elements (i.e., components and connectors) playing attribute-specific roles in this analysis. Table-1 summarizes the mapping for confidentiality analysis to mechanism. Thus, when constructing specifications, peers will be modeled as components, and keys as data tokens. Each security attribute (identity, confidentiality, and so forth) has its own mapping; the collected set of mappings exhibit considerable overlap

Refinement is crucial to QADP (as it is to a variety of architecture description languages), since the vast majority of sentences that can be constructed in the language of Figure 8 and Table-1 are meaningless. Instead, there are a few well-known patterns (nee *mechanisms*) that are known to

⁷ We feel justified in using this term since system architects of complex systems frequently consult, for example, security experts, performance experts, usability experts, and so forth.

strongly relate to confidentiality, i.e., have meaning with respect to this attribute. General mechanisms capture these patterns at the level of basic analysis category; refinement allows the designer (or attribute specialist) to map general mechanisms (and scenarios and analyses) to increasingly specific mechanisms (scenarios and analyses), and, ultimately, to a composition of software components.

Each general mechanism, then, is an assembly specification in the reference model. Each refinement “moves” the mechanism closer to the assembly level of the reference model. We define the boundary between assembly specification and assembly as the point where no further mechanism refinements are possible or useful. For example, basic analysis categories that are modeled as components will ultimately be refined to a particular software component that plays the role of that basic analysis category in a system; at this point no further refinements of that basic analysis category (now component) are possible.

Basic Analysis Category	Metatype Allocation
Peer (P)	Component
Link	Interaction
Data Asset (DA)	Data Token
Cryptographic Service (C)	Component
Threat Agent (TA)	Component
Key (K)	Data Token
Trusted Computing Base (TCB)	Property

Table 1: Type Level for Confidentiality

To consider a more concrete example, we begin with the general attribute scenario for confidentiality. The tripartite definition of *confidentiality* is:

- **Stimulus:** A data asset is transmitted from one communicating peer to another.
- **System elements:** Peers P1 and P2, Data Asset (DA), Link, Threat Agent (TA)
- **Response measure:** $0 < V(DA) \leq T$, where T is the time duration DA will remain undisclosed to TA, a $V(DA)$ is the time DA *must* remain undisclosed to TA.

To expedite the illustration, we bundle analysis and mechanisms with their associated analyses. We also adopt the following diagramming conventions:

- We use UML collaboration diagrams to depict assembly specifications (that is, instantiations of Figure 8 and Table-1). Metatype Properties are modeled as UML annotations, and all metatype Interactions are binary and are therefore implicit.

- We extend UML with a *group* abstraction, represented as a dash-lined box, into which model elements sharing a common property are placed, and where the box bears the name of this shared attribute. This abstraction is useful in many quality attribute settings.

Figure 9 illustrates the first, and simplest, confidentiality mechanism. It specifies a pattern that, by definition, satisfies the response measure stipulated in the general scenario. The mechanism is given the name TCB (Trusted Computing Base), although this aspect of metatype instantiation is left implicit for the purpose of this illustration. However, although the mechanism is stated as an axiom—that this pattern guarantees that for practical purposes $T = \infty$, it does make explicit the conditions under which the TCB attribute adheres to P1, P2, and DA, namely: electro-magnetic and perimeter shielding. Refinements of the TCB pattern must preserve these conditions.

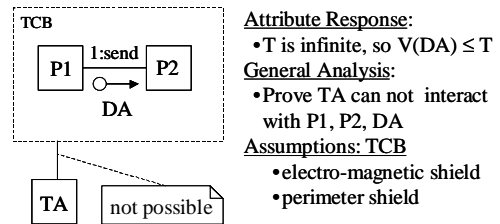


Figure 9: TCB Assembly Specification

Figure 10 illustrates the Cryptographic mechanism. Since this mechanism does not preserve the assumptions of the TCB mechanism (not all model elements share the TCB property), it cannot be a refinement of TCB. Further, since the Cryptographic mechanism contains only basic level analysis categories, it must itself be a basic mechanism.

Note, however, that the Cryptographic mechanism composes two instances of the TCB pattern, and also its associated analyses, which are implicit in Figure 10⁸. The Cryptographic mechanism introduces an additional general analysis, $\underline{f}(C, DA)$, which represents, in functional form, an analysis of the strength of the encryption technology \underline{C} . In this case, no further details can be provided on the construction of \underline{f} ; these details must be provided by refinements of this mechanism.

Two refinements of the Cryptographic mechanism are extant: the Symmetric Key and Asymmetric Key mechanisms. To facilitate their comparison, both mechanisms are shown together in Figure 11, and to preserve space we have

⁸ We also note that the Cryptographic mechanism introduces system elements not stated in the basic confidentiality scenario, and also uses a different response measure. Both modifications are refinements of the general scenario. Thus, scenarios, analyses, and mechanisms may be refined independently of one another.

omitted the analysis description. Both mechanisms are refinements of the Cryptographic mechanism since they:

- Preserve the analysis assumptions of the Cryptographic mechanism.
- Introduce new analysis categories (KeyStore) or refine categories listed in Table-1 (public and private keys, symmetric keys).

Although we do not have space to elaborate, further refinements of the Symmetric Key mechanism are needed to demonstrate how K1 is propagated to both TCBs (NB: this may or may not involve a KeyStore). The Asymmetric Key mechanism can be further refined to the type of asymmetric key algorithm used, for example elliptic key algorithms, and ultimately to cryptographic service providers and their components.

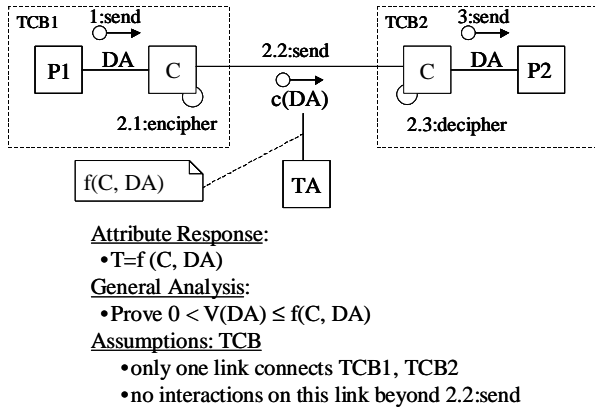


Figure 10: Cryptographic Assembly Specification

Before summarizing the key points we make the following observations. First, the illustration is silent on several difficult topics. For example, we have not discussed how specifications are composed across closely related quality attributes, such as confidentiality and authenticity, let alone across more distant attributes such as security and performance. Nor have we discussed attribute tradeoff analysis. Both matters are areas of active research. On the other hand, we have validated, from our own experience, that security-related design patterns and their refinements, such as those illustrated, are embodied in typical component-based systems [29]. In some respects, theory has not yet caught up with practice.

Returning to the summary of key points of this illustration:

- The metatype, type, and specification levels of the reference model were instantiated from an architecture technology perspective.
- Component technologies were integrated into the architecture-centric instantiation through the process of

specification refinement.

- There is a clear co-dependence between attribute reasoning models and the architectural constraints imposed by a component technology.

4. Criteria for Successful Integration

Taken together, the reference model and its two instantiations support the notion that software architecture and software component technologies are complementary, and there is ample scope for their conceptual integration. But, can these concepts be integrated in practice? We propose the following five criteria upon which to judge success.

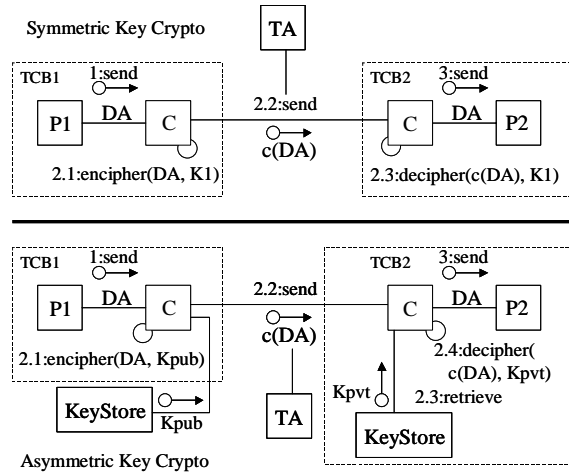


Figure 11: Mechanism Refinements

4.1 Structural Continuity

This criterion was mentioned earlier: that structural (architectural) assumptions underlying quality attribute analysis are preserved, by construction, in the deployed system. This criterion seeks to improve on the current practice of informal conformance checks between architecture documentation and implementation. We illustrated two ways of satisfying this criterion: through a component technology that enforces style constraints, and through a refinement process that preserves analysis assumptions. One of the things that make software component technology so intriguing is the possibility of satisfying this criterion.

4.2 Independent Evolution, Systematic Integration

It is useful that software architecture and component technologies pursue complementary agendas, with component technology reducing the mechanistic barriers to system assembly, and software architecture providing a foundation for reasoning about the qualities of assemblies. But complementary does not imply orthogonality; indeed, our illustrations showed strong co-dependence. Therefore, it is essential to construct a systematic approach to integrating

attribute reasoning and component model without inhibiting their independent evolution.

4.3 Compositional Reasoning Enabled Technology

We need to understand better what properties of a quality attribute make it amenable to compositional reasoning—it is quite possible that some quality attributes are not compositional and never will be. For those attributes that are compositional, we need uniform mathematical and meta-mathematical treatment. QADP reflects our thinking on what is needed.

4.4 Uniform Treatment of Non-Uniform Formality

Having identified attributes that are amenable to compositional analysis, we must avoid inflated expectations of mathematical rigor in compositional reasoning techniques. It is clear that designers have been doing some form of compositional reasoning for years without full formality. As the software engineering body of knowledge expands, our understanding of quality attributes will improve and become gradually more rigorous. More important than rigorous compositional analysis is accommodating a range of reasoning formality in a uniform way.

4.5 Situated Measures of Component Quality

Successful integration of software architecture with software component technology will also answer two vexing questions: What do we mean when we refer to the quality of software components? And, How is this quality measured? The answer to the first question is provided by the patterns of interaction in which a component plays a role to achieve some quality attribute. Components that play their roles well have more of that situation-specific quality than those that play their roles poorly. The answer to the second question is provided by the attribute-reasoning model, which is, in effect, parameterized by the properties of components that enable them to play their role in a pattern of interaction. More rigorous attribute reasoning models will yield more rigorous the measures of component properties.

5. Related Work

The software architecture community has been studying the potential for early and high-level system analysis for many years. This work includes the development of architecture description languages to support formal description of assemblies of components [1,10,17], and compositional analysis techniques based on these (or analogous) languages [5,21,24,25]. Klein et al. argue that the relationship between architectural style and quality attributes should be made explicit, and used as a basis for system design and analysis [13]. These works have had limited success in application to real systems in part because it is hard to prove consistency between implementations and architec-

tural components. Work at the Software Research Institute (SRI) has focused on provably correct architecture refinement [9,20], but their focus is on formalizing refinement across different architectural views in different languages.

At the same time, component software research has focused on developing component models, frameworks, and processes to support constructing assemblies of components. These assemblies share the characteristics of software architecture description and can therefore benefit from the application of architecture-based analysis. In his seminal book on component-based development, Szyperski describes the avalanche that can occur as a result of the failure of a single component to perform correctly. He suggests component contracts for non-functional properties to be used as a basis for locating the under-performing component [27]. Work in the component community has argued for the use of architectural style-specific information during component assembly [4,11,18,19]. Daniels and Cheesman suggest the use of architecture in combination with component-level information to reason about inter-component dependencies to support change analysis [7]. Koen et al. discuss contractual mechanisms for making interactions explicit in component assemblies [11]. Kristensen and May argue that understanding component properties leads to development of better architectures [15].

This references cited above are but a few of the works in these two major research areas but are sufficient to allow us to position our work among that of others interested in this area of research.

6. Conclusions

Software architecture and software components are two sides of the same coin. Given their mutual affinities it is not surprising that there is growing interest in bringing these technologies together. We suggest that the leitmotif for this integration is predictable assembly. In this position paper, we outlined a high level model to express, and relate, complementary aspects of software architecture and software components, with an eye toward predictable assembly. We illustrated two ways to make progress toward this end; doubtless there are many other ways. It is our hope and expectation that significant progress will be made toward predictable assembly within the next few years.

7. Acknowledgements

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

References

1. 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.
2. 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.
3. L. Bass, M. Klein, F. Bachmann, Quality Attribute Design Primitives, Technical Report CMU/SEI-2000-TN-017, December, 2000, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
4. V. Baggiolini, J. Harms, Toward Automatic, Run-Time Fault Management for Component-Based Applications, *Proceedings of the 2nd International Workshop on Component-Oriented Programming (WCOP97)*, held in conjunction with the ECOOP98.
5. S. Balsamo, P. Inverardi and C. Mangano, An Approach to Performance Evaluation of Software Architectures, *Proceedings of the 1998 Workshop on Software and Performance*, Oct. 1998, 77—84.
6. G. Booch, Object Solutions: Managing the Object-Oriented Project, Addison Wesley Longman, 1996.
7. J. Cheesman and J. Daniels, UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley, Reading, Massachusetts, 2000
8. D. Garlan, M. Shaw An Introduction to Software Architecture, *Advances in Software Engineering and Knowledge Engineering*, V. Ambriola and G. Tortora (eds.), World Scientific, 1993.
9. F. Gilham, R.A. Riemenschneider, V. Stavridou, Secure Interoperation of Secure Distributed Databases, in *Proceedings FM'99*, Vol. 1, LNCS 1708, pp. 701-717, 1999, Springer Verlag, Berlin, Heidelberg.
10. D. E. Harms, The Influence of Software Reuse on Programming Language Design. Ph.D. thesis, Ohio State University, Columbus, May 1990.
11. K. De Hondt, C. Lucas, P. Steyaert, Reuse Contracts as Component Interface Descriptions, *Proceedings of the 2nd International Workshop on Component-Oriented Programming (WCOP97)*, held in conjunction with the European Conference on Object-Oriented Programming (ECOOP98).
12. R. Keller, B. Laguë, R. Scuaer, International Workshop on Large Scale Software Composition, *Proceedings of DEXA'98 Ninth International Workshop on Database and Expert Systems Applications*, Vienna, Austria, pp. 765-833.
13. M. Klein, R. Kazman, Attribute-Based Architectural Styles, Technical Report CMU/SEI-99-TR-22, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 1999.
14. M. Klein, T. Ralya, B. Pollak, R. Obenza and M. G. Harbour, A Practitioner's Handbook for Real-Time Analysis, Kluwer Academic Publishers, 1993.
15. B. B. Kristensen, D. C. M. May, Component Composition and Interaction, *Proceedings of International Conference on Technology of Object-Oriented Languages and Systems (TOOLS PACIFIC 96)*, Melbourne, Australia, 1996,
16. G. Lakoff, Women, Fire, and Dangerous Things, University of Chicago Press, 1990.
17. D. Luckham, J. Vera, An Event-Based Architecture Definition Language, *IEEE Transactions on Software Engineering*, Vol 21, No 9, pp.717-734. September 1995.
18. M. Lycett, R. J. Paul, Component-Based Development: Dealing with Operational Aspects of Architecture, *Proceedings of 3rd International Workshop on Program Composition*
19. N. Medvidovic, P. Oreizy, R. N. Taylor, Reuse of Off-the-shelf Components in C2-style Architectures, *Proceedings of the 1997 International Conference on Software Engineering*, pp. 692-700.
20. M. Moriconi, X. Qian, R.A. Reimenschneider, Correct Architecture Refinement, *IEEE Transactions on Software Engineering*, 21(4), pp. 356-372, April 1995.
21. G. Naumovich, G. S. Avrunin, L. A. Clarke, and L.J. Osterweil, Applying Static Analysis to Software Architectures, *Proceedings, 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, No. 1301, LNCS, Springer-Verlag, 1997, pp. 77-93.
22. 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.
23. M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall 1996.
24. B. Spitznagel, D. Garlan, Architecture-Based Performance Analysis, *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering*, San Francisco, California, 1998.
25. 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.
26. C. Szyperski, C. Pfister, Workshop on Component Oriented Programming: Summary. In Mühlhäuser M. (ed) Special Issues in Object Oriented Programming – ECOOP96 Workshop Reader, Springer Verlag, Heidelberg.
27. C. Szyperski, Component Software Beyond Object-Oriented Programming, Addison-Wesley, Boston, Massachusetts and ACM Press, 1998.
28. C. Szyperski, R. Vernik, Establishing System-Wide Properties of Component-Based Systems, Proc of OMG-DARPA-MCC Workshop on Compositional Software Architecture
29. K. Wallnau, S. Hissam, R. Seacord, Building Systems from Commercial Components, Addison Wesley, (July, 2001).
30. W. Weck, Independently Extensible Component Frameworks, *Proceedings of the 1st International Workshop on Component-Oriented Programming*, in conjunction with ECOOP97.

