Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition

Felix Bachmann
Len Bass
Charles Buhman
Santiago Comella-Dorda
Fred Long
John Robert
Robert Seacord
Kurt Wallnau (*Please send questions and comments regarding this report to Kurt Wallnau at kcw@sei.cmu.edu.*)

May 2000

Volume II: Technical Concepts of Component-Based Software Engineering

Felix Bachmann
Len Bass
Charles Buhman
Santiago Comella-Dorda
Fred Long
John Robert
Robert Seacord
Kurt Wallnau (*Please send questions and comments regarding this report to Kurt Wallnau at kcw@sei.cmu.edu*.)

May 2000

Internal Research and Development

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Norton L Compton

Norton L. Compton, Lt Col., USAF
SEI Joint Program Office

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Abstract

The Software Engineering Institute (SEI) is undertaking a feasibility study of "component-based software engineering" (CBSE). The objective of this study is to determine whether CBSE has the potential to advance the state of software engineering practice and, if so, whether the SEI can contribute to this advancement. This report is the second part of a three-part report on the study. Volume I contains a market assessment for CBSE. Volume III outlines a proposed course of action for the SEI. Volume II, this report, establishes the technical foundation for SEI work in CBSE. The paper asserts that the key technical challenge facing CBSE is to ensure that the properties of a system of components can be predicted from the properties of the components themselves. The key technical concepts of CBSE that are needed to support this vision are described: *component*, *interface, contract*, *component model*, *component framework*, *composition,* and *certification*.

1 Nothing New Under the Sun?

The whole *comprises* its parts, and the parts *compose* the whole. To compose, from the Latin **com-** "together" and **ponere**[1] "to put." The parts that we compose are, etymologically speaking, *components*. Why this pedagogy? Because, by definition, all software systems comprise components. These components result from problem *decomposition*, a standard problem-solving technique. In the software world, different conceptions about how systems should be organized result in different kinds of components. Thus, two systems may comprise components, but the components may have nothing more in common than the name "component." The phrase *component-based system* has about as much *inherent* meaning as "part-based whole."

This is not to suggest that *software component technology* has not emerged in recent years as a significant factor in how systems are built. Indeed, as indicated in a companion volume to this report, "Market Assessment of Component-Based Software Engineering," software component technology is thriving and most analysts predict continued growth over the next 5-10 years. Unfortunately, these predictions are tainted by a lack of agreement among analysts about what software components are, and how they are used to design, develop and field new systems. This lack of agreement among analysts extends also to researchers, technology producers, and consumers. But this diversity is to be expected in a new technology, and we should be careful not exaggerate the extent of this diversity: although there are differences they are often quite subtle. By and large, there is general agreement on the broad outlines of what constitutes software component technology, and, by extension, component-based systems.

It is the task of this report to sharpen these broad outlines into a more coherent and detailed picture of software component technology by exposing *its* constituent components: technical concepts. Our objective is not to develop ironclad definitions of these concepts (a futile exercise), but rather to highlight what is most significant. To know what is most significant, however, requires an understanding of how software component technology is used or, more to the point, how this technology *should be used* in the service of an overall engineering discipline based in software components. Not surprisingly, we call this engineering discipline *component-based software engineering* (CBSE). Thus, it is also the task of this report to articulate a vision for CBSE that we can use to orient the discussion of technical concepts.

In Section 2 we begin with a reference model of software component technology that identifies and relates most, but not all, of the technical concepts underlying software component

---

[1] *The American Heritage Dictionary*, 1985.

technology. This, we believe, will give the reader the gestalt of software component technology that will serve as a touchstone for the more detailed discussions that follow. We then present in Section 3 a vision statement for CBSE that provides a context in which the technical concepts can be elaborated. Sections 4 through 9 examine the technical concepts of software component technology that support the engineering vision: *components*, *interfaces*, *contracts*, component *models* and *frameworks*, *composition*, and *certification*. While none of these concepts is entirely new, their relation to each other and to other technical concepts such as software architecture marks this technology as something new. Finally, in Section 10 we summarize the key points and build a bridge to Volume III.

2 Software Component Technology

Component-based systems result from adopting a component-based *design strategy*, and *software component technology* includes the products and concepts that support this design strategy. By *design strategy* we mean something very close to *architectural style*—a high-level design pattern described by the types of components in a system and their patterns of interaction [Bass 98]. Software component technology reflects this design pattern, which is depicted graphically in Figure 1. This reflection is due to the fact that software component technology does not exist only in development tools but also becomes part of the deployed application or system. This pattern is found in commercial software component technologies such as Sun Microsystems' Enterprise JavaBeans™ and Microsoft's COM+, as well as in research prototypes such as the SEI WaterBeans [Plakosh 99] and many others. A high-level understanding of this design pattern is instrumental to later, more detailed discussions.
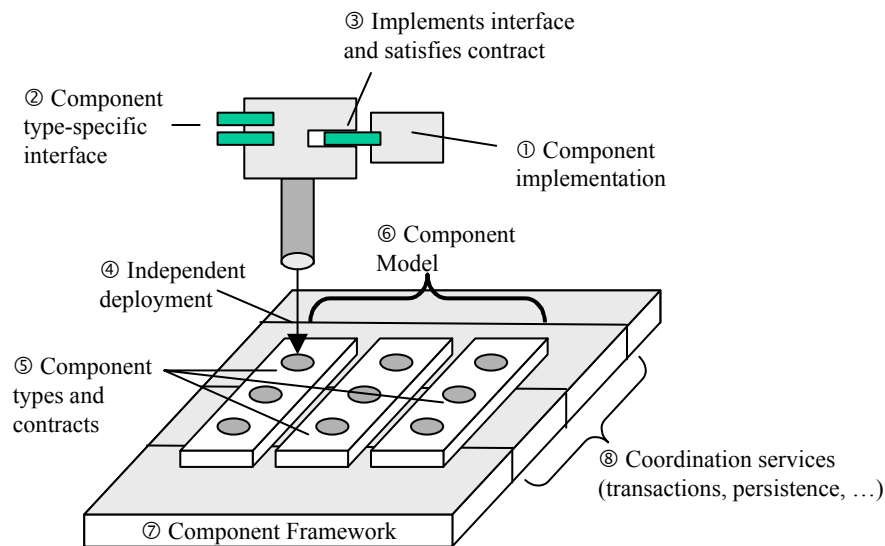
Figure 1:   The Component-Based Design Pattern

A *component* (①) is a software implementation that can be executed on a physical or logical device. A component implements one or more *interfaces* that are imposed upon it (②).This reflects that the component satisfies certain obligations, which we will later describe as a *contract* (③). These contractual obligations ensure that independently developed components obey certain rules so that components interact (or can not interact) in predictable ways, and can be *deployed* into standard build-time and run-time environments (④). A component-based system is based upon a small number of distinct component *types*, each of which plays

a specialized role in a system (⑤) and is described by an interface (②). A *component model* (⑥) is the set of component types, their interfaces, and, additionally, a specification of the allowable *patterns of interaction* among component types. A component framework (⑦) provides a variety of runtime services (⑧) to support and enforce the component model. In many respects component frameworks are like special-purpose operating systems, although they operate at much higher levels of abstraction.

Figure 1 is a reference model for component-based concepts. But, as with all reference models, there is a danger of reading too much into the picture. For example, Weck equates component model and frameworks, and suggests that the framework may or may not include services [Weck 96]. Microsoft's COM+, on the other hand, embeds the component framework into the operating system itself obviating the need for a separate entity called "component framework" [Box 98]. Indeed, it is difficult to find categorical distinctions between component frameworks and operating systems, as both provide coordination mechanisms that enforce a particular model of component interactions. Nevertheless, we assert there are qualitative distinctions; for example, component frameworks will support a more restricted range of coordination schemes than a general-purpose operating system.

Figure 1 depicts the design pattern, but what is it that motivates this pattern in the first place? The following appear to be the most significant factors:

- **Independent extensions**. One problem that plagues legacy software is lack of flexibility.[2] Components are units of extension, and a component model prescribes exactly how extensions are made. In some cases the framework itself may constitute the running application into which extensions (components) are deployed. The component model and framework ensure that extensions do not have unexpected interactions, thus extensions (components) may be independently developed and deployed.

- **Component markets**. Component models prescribe the necessary standards to ensure that independently developed components can be deployed into a common environment, and will not experience unanticipated interactions such as resource contention. The integration of support services in a framework also simplifies the construction of components, and provides a platform upon which families of components can be designed for particular application niches. The Theory Center case study discussed in the companion Volume I Market Assessment illustrates a component-based product line.

- **Reduced time-to-market**. The availability of components of the sort just described also promises to drastically reduce the time it takes to design, develop and field systems. Design time is drastically reduced because key architectural decisions have been made and are embodied in the component model and framework. Component families such as those found in the Theory Center obviously contribute to reduced time to market. Even if such component families are not available in an application domain the uniform component abstractions will reduce development and maintenance costs overall.

---

[2] One definition of "legacy" precisely describes it as a system which is no longer sufficiently flexible to be adapted in a cost-effective way.

- **Improved predictability**. Component models and frameworks can be designed to support those quality attributes that are most important in particular application areas. Component models express design rules that are uniformly enforced over all components deployed in a component-based system. This uniformity means that various global properties can be "designed into" the component model so that properties such as scalability, security and so forth can be predicted for the system as a whole. For example, EJB™ is touted as promising scalable, secure, and distributed transactions by virtue of its component model and framework services.

It might be argued that there are other benefits that accrue from a component-based approach to systems (as that approach is discussed here). However, the benefits described here are sufficient to motivate the following discussion. More details on market perceptions concerning the benefits of component-based software can be found in Volume I of this report.

Summary of Key Points

Software component technology supports building a particular style that includes components, component models, and component frameworks. The component model imposes design constraints on component developers, and the component framework enforces these constraints in addition to providing useful services.

3 Vision Statement

As one modern adage has it, "point of view is worth twenty IQ points."[3] Having a point of view means having a vision of how things are, so that what is important can be sorted from what is unimportant, and so that the tendency of things can be understood and hence predicted. It is for this reason that we express a vision statement for CBSE. Through it we hope to obtain all of these benefits (IQ, perhaps, excluded).

We must decide between a broad or narrow vision. A broad vision will include things such as organizational models, business models, life-cycle models and processes, engineering roles, skills, enabling technologies, and so forth. A narrow vision will articulate some aspect of CBSE that is a lynchpin for a broader vision. The virtue of a broad vision is its inclusiveness, while its weakness can be lack of focus; a narrow vision of course has complementary strengths and weaknesses.

We have elected to pursue a narrow vision of CBSE. The vision we describe is focused on the most fundamental aspect of CBSE–predictable composition–without which CBSE lacks meaning independent from already established thrusts in software engineering practice, for example software architecture or use of commercial off-the-shelf (COTS) software. We do not claim that this vision is novel or unique to the SEI. We do claim, however, that it expresses the nub of what *engineering* means in any meaningful CBSE practice.

> **Component-based software engineering is concerned with the <u>rapid assembly</u> of systems from components where**
>
> - **components and frameworks have <u>certified properties</u>; and**
>
> - **these certified properties provide the basis for <u>predicting the properties of systems</u> built from components**

The above statement is minimal but aggressive. The gist of the vision is expressed in the three underlined stipulations, taken out of order for expository reasons:

- Predicting the property of a solution from its constituent parts is fundamental to all mature engineering disciplines. The desire to achieve this result for software systems has motivated much research in software engineering science, for example work in so-called "formal methods" and software architecture. This stipulation can be viewed as either the

---

[3] Attributed to Adele Goldberg.

"Holy Grail" or *sine qua non* to a software engineering discipline based on components. These views are not, incidentally, mutually exclusive, since there is a limit on the scope and accuracy of system predictions in even the most mature engineering disciplines.

- It is highly improbable that an engineer could predict the property of a system without having the benefit of knowing the properties of the parts that comprise the system. Certification of component and framework properties is important because it implies that engineers will be working with software products that are (or have) "known quantities." It also implies (but possibly does not mandate) an authoritative industrial approach to obtaining trust in the properties of the fundamental building blocks of software systems.

- Rapid assembly is important because, as noted in the Market Assessment, reduced time to market is a primary motivation for adopting software component technology. It will little avail us to find a way of satisfying the previous two stipulations if doing so requires substantially more development time than is already the case. Indeed, it can be argued that prediction and certification is important precisely because they support the ultimate objective of reducing the time it takes to design and build software systems.

In the interest of concision many nuances have been omitted from the vision statement. For example, predicting system properties requires that the properties of the *interactions* among components and between components and framework must also be known in addition to the properties of components and frameworks. There is also a strong connection between the vision statement and commercial markets in components, frameworks, and certification. The commercial technology market is driving advances in software component technology, and a failure to accommodate these market realities in CBSE would be a decisive error.

Summary of Key Points

A vision of an ideal CBSE practice is essential if we are to understand software component technology and the ways in which this technology supports or hinders sound software engineering practice. The SEI vision for CBSE emphasizes *rapid* assembly of *certifi*ed components and frameworks into *systems* whose properties can be *predicted* beforehand.

4 Components

There is no shortage of definitions of *component* in the literature. This is not surprising, as different understandings of problems to be solved and approaches to solving these problems invariably lead to different understandings of the constituent parts (components) of solutions to problems. Advocates of software reuse equate components to anything that can be reused; practitioners using COTS software equate components to COTS products; software methodologists equate components with units of project and configuration management; and software architects equate components with design abstractions. There are other analogous equations throughout the literature.

Even within the self-described component-based software engineering community there is considerable variation in definitions of component, although these variations are usually quite subtle. The SEI definition is consistent with the overall thrust of these definitions, notably Szyperski's [Szyperski 98]. But the SEI approach to components strongly reflects an argument made by Wang that software components merge two distinct perspectives: component as *implementation* and component as *architectural abstraction* [Wang 99]. We will temporarily denote this union as an architectural component; we will later dispense with the adjective architectural as being implied by the definition.

The concept of component as implementation is a familiar one found in the marketplace, and as mentioned above is most often used to refer to COTS products. But COTS products may implement functionality (what a component does) and coordination (how a component interacts with the external world) in a way that is unique to the product. In contrast, *architectural components* are required to implement one or more interfaces that prescribe how components may interact or other architectural constraints. In Section 2 we stated that these interfaces are constituent parts of a component model. It is compliance with a component model that makes a component *architectural*. This leads to the following definition of software component.

> **A Component is:**
> - **an opaque implementation of functionality**
> - **subject to third-party composition**
> - **conformant with a component model**

There are two motivations for the criterion that a component is an *opaque implementation*. First, we envision a commercial market in software components. Notwithstanding the suc-

cess of Linux and "open source" software, the predominant and most successful business model for software components has been based upon software as intellectual capital that must be protected from disclosure. This is likely to remain true for the foreseeable future, and a technical agenda for CBSE must assume therefore that components will remain "black boxes" to consumers. Second, as is already a well-established precept in computer science, clients of software components should not come to rely upon implementation details that are likely to change. In computer science this has led to programming support for abstraction and information hiding; opaqueness serves the same purpose for components.

The motivation for third-party composition is straightforward: the use of components should not depend upon tools or knowledge of the component that is in the possession of only the component provider. This criterion implies that a component-based system can comprise components from multiple, independent sources, and that a system can be assembled by a third party system integrator who is not also a component supplier. This criterion should hold true even if none of the components used in a system come from external suppliers.

The last criterion, that a component is conformant with a component model, is what differentiates components from conventional COTS software products. Component models prescribe how components interact with each other, and therefore express global, or architectural design constraints. Conformance to component models transforms software implementations into architectural implementations. In contrast to COTS-based systems, which result in a hodge-podge integration of product-specific interaction schemes, component-based systems are based in uniform, standard coordination schemes.

As with the CBSE Vision, all three criteria reflect the background forces of a commercial market in software component technology.

Summary of Key Points

A software component merges two distinct perspectives: component as an implementation and component as an architectural abstraction; components are therefore architectural implementations. Viewed as implementations, components can be deployed, and assembled into larger (sub)systems. Viewed as architectural abstractions, components express design rules that impose a standard coordination model on all components. These design rules take the form of a *component model,* or a set of standards and conventions to which components must conform.

5 Interfaces

Our ability to integrate components into assemblies, to reason about these assemblies, and to develop a market of components depends fundamentally on the notion of component *interface*. The concept of interface is basic and familiar. But familiarity sometimes breeds contempt, and the concept of interface is more complex than is often appreciated. Moreover, the criticality of interfaces to CBSE exposes limitations in conventional approaches to interface specification.

5.1 Interface Abstraction and Application Programming Interfaces

Interface abstraction provides a mechanism to control the dependencies that arise between modules[4] in a program or system. An application programming interface (API) is a specification, in a programming language, of those properties of a module that clients of that module can depend upon. Conversely, clients should not depend upon properties that are not specified by the API. All modern programming languages support some form of interface abstraction, e.g., Smalltalk-80, C++, Modula-3, Ada-98, Java and ML. Some language-neutral interface specification languages have been developed such as the Object Management Group's (OMG) Interface Definition Language (IDL).

Design and implementation decisions that are unlikely to change are specified in the API, while decisions that are likely to change are "hidden" from clients. The theory is that information hiding makes modules substitutable (for example, with new versions of a component), and hence makes systems easier to change, at least insofar as module substitution is concerned. This turns out to be a weak theory, however, as it depends upon API's being silent about properties that clients should not depend upon. But the API can only be silent about properties about which it can *speak,* and programming languages are only equipped to speak about a narrow range of properties. All other properties can "leak" through the interface abstraction.

Conventional APIs—that is, interface specifications written in programming languages such as those cited above—can conveniently express what we will refer to as *functional* properties. Functional properties include the services a module provides and the signature of these services—the types and order of arguments to the service and the manner in which results are returned from the service. Conventional APIs are not so well equipped to express what we

---

[4] We will use the term *module* to refer to software implementations that have interfaces; we reserve the use of the term *component* to things that satisfy the SEI definition of that term.

refer to as *extra-functional* properties. These properties include things like performance, accuracy, availability, latency, security, and so forth. These are often referred to as *quality attributes*, or, when associated with a particular service, *quality of service*. Because APIs can not describe these properties, they can not hide them. Indeed, modules may come to depend upon any of these properties, thus reducing the probability that one module can be substituted for another.

Note that just because an extra-functional property is not expressed in an API does not *a priori* mean that this property will be the source of a dependency—it will only become a dependency if a client relies upon this property. This is cold comfort, however, since many more such dependencies arise than is usually recognized by programmers and designers.

5.2 Extending APIs to Extra-Functional Properties

Attempts have been made to extend APIs to make them more expressive of extra-functional properties. These extensions are motivated more by the desire to ensure that interface specifications are sufficiently complete to ensure correct integration than by the desire to extend the scope of information hiding to additional properties. Both ends are served by these extensions, however.

For the following discussion it will be useful to differentiate the various kinds of extra-functional properties that have been the subject of attempts to extend APIs. Beugnard et. al. define four kinds of property: *syntactic*, *behavioral*, *synchronization*, and *quality of service* [Beugnard 99]. Although no classification scheme is perfect, this one will do for our purposes. Syntactic properties correspond to functional properties as we have just described them. The other properties are extra-functional, and the means of expressing these are now discussed. Many of the topics discussed below were explored in depth at a workshop on the Foundations of Component-Based Systems held in Zurich, Switzerland in September 1997 [Leavens 97].

5.2.1 Specifying Behavior

Beugnard et al. suggest that behavioral specifications define the *outcome* of operations [Beugnard 99]. Programming languages such as Eiffel [Meyer 92b] and SPARK [Barnes 97] allow for behavioral specifications to be written into the program code using pre- and post-conditions and other forms of assertions. However, this technique is not available in the more commonly used languages. There have been at least five recent attempts to extend the Java programming language in this way: iContract [Kramer 98], JML (Java Modeling Language) [Leavens 99], Jass (Java with assertions) [Fischer 99], Biscotti [Della 99], and JINSLA (Java INterface Specification LAnguage) [Mikhajlova 99]. This diversity illustrates the fact that such approaches are still not mainstream and that the industry has not yet picked a single winner in this area.

Formal methods such as VDM, Z or Larch may also be used to specify the behavior of software components [Goldsack 97, Johnson 97, Ciancarani 97]. These methods are becoming more widely used but they are still not regarded as mainstream. These well-established formal methods were introduced before the advent of object-oriented programming. However, they can still be used in the specification of object-oriented systems, even though they, themselves, do not directly support object orientation. More recently, object-oriented extensions of these traditional methods have been developed; e.g., OOZE [Alencar 92], VDM++ [Dürr 94], and Object-Z [Duke 95]. Thus, there are signs that these "formal" methods are beginning to be applied to the specification of software components.

5.2.2 Specifying Synchronization

Although the above techniques allow for the behavioral specification of components, they deal only with sequential aspects. This will prove insufficient, as systems are increasingly distributed and concurrent.

Specifying synchronization of components is more of a challenge and is even less well understood than the methods mentioned above. The Object Calculus [Lano 97] has been proposed to meet this need. Other approaches to solving this problem involve the use of Hoare's Communicating Sequential Processes (CSP) [Allen 97] or Milner's pi-calculus [Canal 97, Henderson 97, Lumpe 97]. This latter approach has led to the development of a composition language called Piccola [Acher 99] that does seem to be particularly relevant to the problem of component composition. Along similar lines, an architectural style description language (ASDL) for giving syntactic and semantic information about the components of a software system, and the relationships between those components, using a combination of Z and CSP, has been proposed by Rice and Seidman [Rice 99].

5.2.3 Specifying Quality of Service

According to Beugnard et. al., quality of service includes attributes such as maximum response delay, average response, and precision. We might generalize this a bit to include quality attributes. Quality attributes include quality of service as well as global attributes of a component such as its portability, adaptability and so forth.

The specification of quality attributes is, perhaps, even more of a research issue and less widely undertaken than the specification of the behavior or synchronization. One notation that can be applied to this problem is called NoFun [Franch 98]. This notation allows the definition of non-functional attributes of software. Among the most widely accepted such attributes are: time and space efficiency, reusability, maintainability, reliability, and usability. Having defined these attributes for a piece of software, NoFun then allows the non-functional behavior of a component to be defined with respect to the attributes.

Although a formal notation for quality attributes is still a matter of speculation, Szyperski cites the example of Swiss banks specifying the required service level of a component sub-

contracted out to a third party [Szyperski 98b]. This covers guarantees of availability, mean time between failures, mean time to repair, throughput, capacity, latency, data safety for persistent state, etc. Finally, in the field of performance specification, it may be possible to specify the complexity of a component. The Apple/SANE™ libraries specified the computational complexity of various library functions.

5.3 Credentials

The bewildering variety of attempts to increase the accuracy and/or scope of interface specifications suggests that no consensus has yet emerged on how to describe the properties of modules (or components). With increased scope and accuracy comes (it seems) increased formality, which in turn leads to greater complexity and the need for still more specialized skills. A very different position was taken by Mary Shaw, who argued for *credentials* as a form of interface specification [Shaw 96]. Informally, a credential is a <name, value, confidence> triple, where *name* refers to the name of the property, *value* refers to the way this property is manifested by a particular component, and *confidence* refers to different degrees of certainty attached to the name/value pair. Shaw's argument was that interface specifications for "architectural components" (sic) (by which she means "large") are inherently incomplete, given the limitations of current specification formalisms, the complexity architectural components, and the variety of different ways that a component will be used.

Shaw envisions a kind of *lingua franca* of credentials, with standard properties and well-defined measurement scales for these properties. Take the illustration below. A simple functional property of a module might be that it provides a particular service. This becomes a fact that might be known with great accuracy, for example <service, sort, demonstrated> states that the module has a service called "sort" and that this service has been demonstrated in a sample program. However, describing properties can become a complicated matter. In particular, properties are not necessarily atomic but are sometimes aggregate. For example, a property of the service might be its performance under a particular security level, or its accuracy under a particular performance level. For N extra-functional properties of a service there are as many as 2^N aggregations of these properties. Even though only a small subset of these may have independent meaning, it will be difficult to itemize these aggregations to say nothing of naming them. Note, however, that Shaw argued that such completeness was impractical if not impossible.
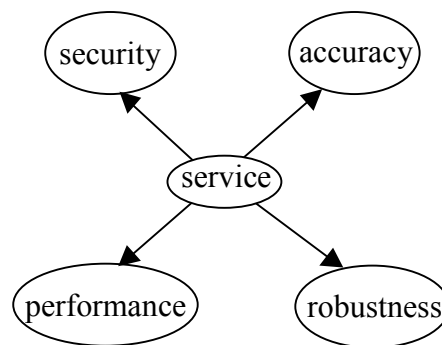
Thus, although credentials cannot be a complete answer, the idea is useful for at least two reasons. First, the distinction between truth and knowledge may be crucial in component-based systems. The distinction is certainly important to the issue of component certification since there will likely be a cost versus confidence tradeoff made in certifying properties that,

for the present at least, resist formal specification or verification techniques. Second, the open-ended nature of credentials nicely mirrors a much earlier argument made by Parnas that a module's interface is, essentially, the set of assumptions that can be made about that module [Parnas 71]. The views of Shaw and Parnas take us away from the narrow perspective of interface as formal specification, and suggest the need for a more flexible way of describing component interfaces.

5.4 Components and Multiple Interfaces

Interestingly, Shaw's credentials blur the distinction between a module's implementation and its interface. This blurring runs counter to developments in software component technology where there is a clear distinction between (and separation of) interface and implementation. The separation we refer to is much stronger than that suggested by languages that support separate compilation of interfaces and implementations, for example, C++ header files or Ada package specifications. In languages and systems that support software components an interface may be implemented by many distinct components and a component many implement many distinct interfaces.

One concrete realization of this idea is found in Microsoft's Component Object Model (COM). In COM, interface specifications are assigned a globally unique identifier (GUID) at the time they are created; each revision of that interface is assigned a new GUID. Components are binary implementations (COM is a binary standard) that are bound to the interfaces they implement via these interface GUIDs. Similarly, clients are linked to components via interface GUIDs. Thus there is a clean separation of interface and component, with clients and component alike bound directly to interfaces and only indirectly to each other. The Java programming language also distinguishes interface specification from class specification by introducing an interface *type*. In both COM and Java the idea is that clients depend upon interfaces and never upon implementations, and that components can implement any arbitrary number of interfaces.

But as we have seen from the discussion above, a complete separation is difficult or impossible to achieve in practice: a module implementation will have properties in addition to those specified on an abstract interface. Invariably, component implementations introduce new properties that might "bleed through" the interface. For example, assuming that an interface specification language could express performance properties (neither Java nor Microsoft IDL, the language used to describe COM interfaces, can do so), an interface may stipulate that a particular sorting operation must exhibit complexity no worse than M•*log*(N). Even if a component were compliant with this property

---
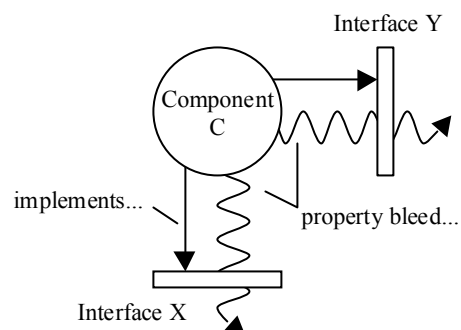
(and we would need to be precise about what this means), it would nonetheless bind a *particular value* to M. This binding would, in effect, change the performance property of the interface, since the client could conceivably come to depend upon this particular binding.

There are more formal ways of expressing interface bleed, but the point is that it is crucial to distinguish between *abstract* interfaces (those that are described independent of any implementation) and *bound* interfaces (those that are associated with an implementation). This distinction is an essential one for certification, composition and system analysis.

Summary of Key Points

Interfaces describe those properties of a component that might lead to inter-component dependencies. However, the state of the art of interface specification is still quite limited in its ability to describe the properties of components; hence unexpected dependencies may arise ("property bleed"). This is especially true where quality of service properties are concerned. It is therefore important to distinguish abstract interface, which are independent of an implementation, and bound interfaces, which are associated with an implementation and therefore may exhibit properties not found in the abstract interface.

6 Contracts

Interfaces were described earlier as specifying a one-way flow of dependencies from modules that implement services to the clients that use these services. That is, the client has some assumptions about the service and hence comes to depend upon these assumptions, while the module interface specifies those assumptions that are, in some way, sanctioned by the designer of the module. However, it is more accurate to say that a client and module are co-dependent. That is, a client depends upon a module to provide a service in a certain way, and the module depends upon the client to access and use these services in a certain way.

While co-dependence is a factor in even simple client/module interactions, its implication to component-based software is magnified because of the premium placed on component substitutability, and because the component interactions that define the context in which substitution occurs can be significantly more complex than in traditional systems. It is partly in recognition of this that the idea of *interface contract* has become prominent in component-based research literature.

The use of contract as a metaphor for specifying software predates software component technology (for example, see "ISTAR and the Contractual Approach" [Dowson 87]). However, the idea of interface contract is most closely linked to the work of Bertrand Meyer and the developments inspired by that work [Meyer 92a, Meyer 97].

Interface contract is a metaphor with connotations that are useful to CBSE. For example

- Contracts are *between* two or more parties.

- Parties often negotiate the details of a contract *before* becoming signatories.

- Contracts prescribe normative and measurable behaviors on *all* signatories.

- Contracts can *not be changed* unless the changes are agreed to by all signatories.

The first bullet highlights the issue of co-dependence, and generalizes this to the more general case that arises in component-based systems where multiple components coordinate to implement a single, logical interaction. The second bullet can be useful in understanding component composition, as all parties must agree to any "bleed through" properties introduced by a component that implements an otherwise abstract interface. The third bullet has implications for component certification, and the last bullet on stable standards for building markets for components.

There are no doubt other connotations as well, but these serve to motivate what contracts are about. The following discussion focuses on the first bullet, and in particular on the idea of specifying *reciprocal obligations* among the parties of an interaction.

6.1 Contracts and Reciprocal Obligations

Component interaction involves a number of (usually) tacit agreements between a client and a component. For example, a client depends upon the component to provide a service, and perhaps depends upon a number of extra-functional properties as well. In return, the component may depend upon the client to provide data arguments within certain bounds or to have properly initialized the component service. To some extent these co-dependencies can be specified in an API through the use of an assertion language, most commonly the use of pre- and post-conditions. A *contractually specified* interface makes these co-dependencies explicit.

```
1  INTERFACE Directory;   (* introduce a new interface type *)
2  IMPORT Files;          (* import an existing interface type *)
3  TYPE Name=ARRAY OF CHAR;
4
5  PROCEDURE ThisFile(n:Name):Files.File;
6      (* PRE n/="")
7      (* POST result=File named n OR result=NIL AND no such file *)
8
9  PROCEDURE AddEntry(n:Name; f:Files.File);
10     (* PRE n/="" AND f=/NIL *)
11     (* POST ThisFile(n)=f *)
12
13 ...other details deleted
14 END Directory;
```

Figure 2:   Contractually Specified Interface (Fragment)

Consider the example interface in Figure 2, adapted from Szyperski[5] [Szyperski 98]. This interface specification describes a number of functional properties of a component that provides a directory service—specifically, the names and signatures of two operations of the directory service (lines 5 and 9). Also specified are behavioral properties that are related to these operations; these are specified in an assertion language of pre- and post-conditions that are, in this illustration, embedded as comments in the language of the API specification. Note that the specification of a contract does not depend upon the use of pre- and post-conditions. Other techniques have been used; for example, a set of rules that map sequences of input events to sequences of output events [Berry 98].

Returning to Figure 2, the post-condition for the `AddEntry()` operation (line 11) asserts that after invoking this operation a file can be retrieved at some future time using the `This-File()` operation (line 5). The post condition for the `ThisFile()` operation (line 7) as-

---

[5] We have altered the illustration slightly for the purposes of this discussion.

serts that it returns a file of a given name or else it returns no file, in which case the operation guarantees that no such file of that name exists within the directory. Thus, the post conditions specify properties of the directory service that clients may depend upon. In contrast, pre-conditions specify those properties that the directory service *requires of the client*. For example, the precondition for `AddEntry()` operation (line 10) is that a client provide a valid file name and file object. In a real sense, the directory service depends upon the client to satisfy this precondition. Earlier it was demonstrated that dependencies arise from assumptions about properties and that a specification of these properties is, by definition, an interface specification. So, the preconditions in Figure 2 specify an interface (a specification of properties) that it requires clients of the directory service to possess.

Thus, it is useful to think of an interface specification as comprising a *provides* part and a *requires* part. This is depicted graphically to the right for directory component D and a client component C. In this illustration D warrants that it will implement the *provides* part (labeled as PRO_D in the graphic), but to do so it needs C to implement the *requires* part (labeled REQ_D). Likewise C has its own interface comprised of provides and requires parts, labeled PRO_C and REQ_C, respectively.
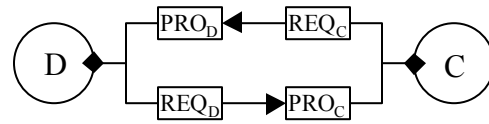
For D and C to interact C must provide what D requires, and D must provide what C requires. The directory service *provides* an operation `AddEntry()` but it *requires* that client components call this operation with file names that are non-empty strings; the client *provides* that it will do so.

The contract specified in Figure 2 specifies reciprocal obligations *between* a directory service and clients of that service. In doing so it also specifies a simple *interaction model*, and the obligations that a directory and service must fulfill to participate in the interaction model. A simple interaction is specified by the pre- and post conditions of the operation: *if* the client provides a valid name, *then* the operation will return a file or NIL. Contracts may also specify more complex *patterns of interaction*. By pattern of interaction we mean a sequence of inter-actions among two or more components that obey some specified rules. The `ThisFile()` interaction is a trivial pattern of interaction—it involves only two components, the directory component and the client component, and applies only to a single invocation of `This-File()`.

```
1 DEFINITION Directory; (* introduce a new interface type *)
2 IMPORT Files;        (* import an existing interface type *)
3 IMPORT Notifier;              (* interface type for callbacks *)
4 TYPE Name=ARRAY OF
5
6
7 PROCEDURE Init();
8     (* PRE True *)
9     (* POST Init=True *)
10
11 PROCEDURE
12     (* PRE Init=True AND n/="")
13     (* POST result=File named n OR result=NIL AND no such file *)
14
15 PROCEDURE AddEntry(n:Name;
16     (* PRE Init=True AND n/="" and f=/NIL *)
17     (* POST ThisFile(n)=f AND FORALL Notifiers CB:              )*)
18
19 PROCEDURE
20     (* PRE CB/=NIL *)

...other details deleted
END Directory;
```

Figure 3:   Extended Contract with More Explicit Patterns of Interaction

More complex patterns of interaction are illustrated (as highlighted text) in Figure 3, which extends the contract shown in Figure 2. For example, the preconditions on `ThisFile()` (line 12) ensures that that prior to calling `init()` no interactions involving `ThisFile()` are allowed. Similarly, `AddNotifier()` and the interface type `Notifier` introduce a pattern of interaction commonly referred to as a *callback*. This pattern of interaction involves the Directory component and an unspecified number of components that implement the `Notifier` interface, each of which, after being registered with `AddNotifier()`, will be called whenever the `AddEntry()` operation is invoked by any client (line 17). Thus a pattern of interaction may span several components and several invocations. Arbitrarily complex patterns of interaction may be specified in a component contract.

6.2 Two Senses of Contract

The contracts in Figures 2 and 3 may have specified a pattern of interaction, but the pattern is rooted on the specified component. This points to two distinct senses of contract, one where the subject of the contract is a component and the other where the subject is the interaction itself. For lack of better terms we will denote the

A component contract specifies a pattern of interaction rooted on that component. The contract specifies the services provided by a component and the obligations of clients and the environment needed by a component to provide these services.

former as a *component contract* and the latter as an *interaction contract*.

Component contracts are natural and necessary. A supplier of components needs to describe the services offered by a component and its other properties. The contractual specification of that component goes further and also describes the conditions that must hold for that component to work properly. Component contracts are necessary, even though a component model may impose some or all of the interfaces required of a component. For as we have already noted, a component implementation will introduce other properties, including obligations on other components, not specified by a component model.

Where component contracts describe components, which, as we know from an earlier definition, are implementations, *interaction contracts* specify the reciprocal obligations among *interface types* that may be implemented by arbitrary components. Interaction contracts play the role of design specification that will be "filled in" by com-

> **An interaction contract specifies a pattern of interaction among different roles, and the reciprocal obligations of components that fill these roles.**

ponents. Since a component can implement multiple interface types, it can "fill" several such roles. Thus, each interaction contract describes a pattern of interaction among different roles in a system. Component models such as EJB™ define interaction models (and hence contracts) between, for example, *Containers* and *SessionBeans*. It is interesting to note that the term *contract* is most often used to describe component contract sense, but the original sense of the term more closely follows the interaction contract sense [Holland 92]. Nonetheless, D'Souza's influential CBSE design methodology has a separate notation for interaction contracts [D'Souza 99]. Berry and Kaplan's Finesse language and environment provides a way of specifying both senses and combining them [Berry 98].

Summary of Key Points

Contracts shift the focus from specification of components to specification of patterns of interactions, and the mutual obligations of participants in these interactions. There are two senses of contract that are necessary to CBSE: component contacts and interaction contracts. Component contracts describe patterns of interaction that are rooted on a component. Interaction contracts describe abstract patterns of interaction among roles that are filled by components. Systems are assembled from components through a process of filling roles with components.

7 Component Models and Frameworks

There is some terminological confusion in the literature concerning component models and frameworks. As already noted, Weck disdains to distinguish these concepts, preferring to define a component framework as standards and conventions that may or may not include support services [Weck 96]. D'Souza and Wills use the term *component kit* in a similar vein to Weck's *framework*, and use the term *component model* to refer to the interfaces, assumptions and so forth of individual components—that is, each component has its own component model [D'Souza 99, Wills 99]. Nevertheless, these are differences in how categories are labeled rather than in the categories themselves. There is consensus that component-based systems rely upon well-defined standards and conventions (what we call a component model) and a support infrastructure (what we call a component framework).

A *component model* specifies the standards and conventions imposed on developers of components. Compliance with a component model is one of the properties that distinguish components (as we use the term) from other forms of packaged software. A *component framework* is an implementation of services that support or enforce a component model. Both are examined more closely, below.

7.1 What are Component Models?

There is as yet no agreement on what should or must be included in a component model. We can obtain some leverage on this question by being more explicit about the purpose of a component model—what is it that we expect to achieve by imposing standards and conventions on component developers?

Uniform composition. Two components can interact if and only if they share consistent assumptions about what each provides and each requires of the other. Obviously some of these assumptions will refer to some unique aspect of each component, usually the function computed by a component. But there are other assumptions that might be standardized across all components, thereby reducing chances for accidental mismatches that inhibit composition of components. These standards might address how components are located, how control flow is synchronized, which communication protocol is used, how data is encoded and so forth.

Appropriate quality attributes. It is a matter of general agreement that the quality attributes of a system will depend upon its "software architecture." Standardizing the *types of component* used in a system and their *patterns of interaction*—what has been defined as *architectural style* —is one way to ensure that a system composed from third-party components will possess the desired quality attributes [Bass 98]. Closely related to quality attributes is quality

of service, which can also be obtained by specifying that patterns of interaction are transactional, or encrypted, and so forth.

Deployment of components and applications. The success of CBSE depends on the emergence of a robust market in third-party components. A precondition for component composition is that components can be deployed from the developer environment into the composition environment, and that applications that have been composed from components can be deployed from the composition environment into the customer environment. This is part of the motivation for component frameworks—they provide a standard compose-time and run-time infrastructure that will clearly simplify the deployment of components and applications.

Given the above motivations, component models will impose standards and conventions of the following kind:

- *Component types*. A component's *type* may be defined in terms of the interfaces it implements. If a component implements three different interfaces X, Y and Z, then it is of type X, Y and Z. Moreover, a component that implements X, Y and Z is polymorphic with respect to these types—it can play the role of an X, Y, or Z at different times. This is an important aspect of components found in both Microsoft/COM and Sun/Java technologies. A component model requires that components implement one or more interfaces, and in this way a component model can be seen to define one or more component types. Different component types can play different roles in systems, and participate in different types of interaction schemes (see next item).

- *Interaction schemes*. Component models will specify how components are located, which communication protocols are used, and how qualities of service such as security and transactions are achieved. The component model may describe how components interact with each other, or how they interact with the component framework. The former class of interactions includes constraints on which component types can be clients of other types, the number of simultaneous clients allowed, and so forth (topology constraints). The latter class of interaction includes things relating to resource management such as component lifecycle (activation, deactivation), thread management, persistence and so forth. Interaction schemes may be common across all component types or unique to particular component types.

- *Resource binding*. The process of composing components is a matter of binding a components to one or more resources. A resource is either a service provided by a framework or by some other component deployed in that framework. A component model describes which resources are available to components, and how and when components bind to these resources. Conversely, a framework sees components as resources that must be managed. Thus, deployment is the process by which frameworks are bound to components, and a component model will describe how components are deployed.

7.2 Component Framework

A good way to think of a component framework is as a mini-operating system. In this analogy, components are to frameworks what processes are to operating systems. The framework manages resources shared by components, and provides the underlying mechanisms that enable communication (interaction) among components. Like operating systems, component frameworks are active and act directly upon components in order to manage a component's lifecycle or other resources, for example to start, suspend, resume, or terminate component execution. However, unlike general-purpose operating systems such as Unix, which support a rich array of interaction mechanisms[6], component frameworks are specialized to support only a limited range of component types and interactions among these types. In exchange for a loss in flexibility there is improved prospects for component composition.

Although this is a good analogy, and holds up well when used to describe a variety of component-based technologies, there are other possible realizations of component framework. It is not necessary that the framework have a runtime existence independent of components. For example, the framework implementation may be bundled with the component implementation. An example of this can be found in [Koutlis 98], where a component-based system originally implemented in the OpenDoc framework was re-hosted to a Java implementation. In the re-hosted version the component types and coordination schemes were implemented as Java classes that were invoked by components. Nevertheless, the trend in component technologies seems to be towards framework as independent implementation, making the operating system analogy quite apt.

Many examples of component frameworks (of the "operating system" sort) can be seen in practice. The Enterprise JavaBeans™ (EJB) specification defines a framework of *servers* and *containers* to support the EJB component model, with servers responsible for providing persistence, transaction and security services while containers are responsible for managing component life cycle[7]. The WaterBeans component framework is specialized to component interactions for real-time visualization of data streams, and visual composition of components. The WaterBeans framework supports high-performance data streams and calculates execution schedules; based on these schedules it invokes component operations to fill these streams. And of course Microsoft's VisualBasic framework, which is specialized for visual composition of components (called VBXs) rather than for any particular runtime quality attribute, is perhaps the most successful commercial framework yet produced. In this case the framework is the VisualBasic interpreter for scripting and composition coupled with the COM deployment and communication services provided by the native operating system.

Note that the overtly component-based illustrations above have analogues in established software engineering practice. For example, Simplex is a component framework but for the

---

[6] These are often referred to as interprocess communication mechanisms, and in the case of Unix includes signals, remote procedure calls, pipes, sockets, shared memory and the file system.
[7] To date, commercial implementations of EJB bundle server and container into one "framework."

fact that the replacement units it manages do not fully satisfy our above-stated definition of component [Sha 95]. The Simplex "framework" supports online upgrade of components while maintaining fault-tolerant, real-time behavior. Similarly, many cyclic executives and other minimal operating systems that were developed for real-time systems are essentially component frameworks. The motivation for developing all of these is the same: to ensure control, and hence predictability, of one or more quality attributes (fault tolerance and real-time performance in the case of Simplex, real-time performance in the case of cyclic executives). This is also the motivation for developing custom component frameworks.

7.3 Custom Frameworks and Programmable Middleware

Tension exists between two tendencies in software component technology: standard versus custom component models and frameworks.

On the one hand, we have argued that component models have a natural correlation with the idea of *architectural style*. Just as different architectural styles or combinations of styles are needed to achieve different mixes of quality attributes in systems, so too, it would seem, different component models and supporting frameworks are needed for the very same reason. Minimally, this would argue for a catalogue of component models and off-the-shelf frameworks. This argument is well supported in the literature [Baggiolini 97, Gannon 98, Garlan 98, Lycett 98, Olken 98, Szyperski 98a, Yucel 98, Fellner 99, Lauder 99, Plakosh 99]. In fact, the tendency in these illustrations argues that no simple catalogue will suffice, but that instead what is needed is a technique for constructing component models from a kit of model fragments known to support particular quality attributes (see closely related ideas about attribute-based architectural styles) [Klein 00].

On the other hand, this tendency toward differentiation of component models runs counter to what is needed to establish a robust market in software components: standard component models and frameworks. Besides being intuitively obvious, the need for viable component standards is described in Szyperski and elsewhere [Szyperski 98b, Bellur 98]. Simply put, component developers need a "sufficiently large" market for deploying their products. Although what is "sufficiently large" may vary across business sectors and component types, a market fragmented into a number of non-standard frameworks will certainly inhibit the emergence of component markets. Norman makes the case that standard "non-substitutable" infrastructures are needed (frameworks) to support markets in substitutable products (components) and he backs up this argument with numerous illustrations from the time of Thomas Edison to current computing technologies[8] [Norman 98]. He also argues that competition in non-substitutable infrastructures is "winner take all," and that technologies will often fail where no such winner emerges (again, he sites numerous examples).

---

[8] Although Norman was arguing for information appliance technologies his arguments apply with equal force to component technologies.

So how is this tension between variation in quality attribute requirements and the need for standard component models and frameworks to be resolved? One way is to adopt a "wait and see" attitude and hope that the market produces component standards in market sectors that are sufficiently large and cohesive to benefit from such standards. This is a reasonable approach, given that the impetus for component technology is market pressure for reduced time-to-market and system development costs, rather than the emergence of any particularly innovative technology. Microsoft's COM+ and Sun's Enterprise JavaBeans have emerged to address the needs of management information systems (i.e., distributed, secure, transactional business logic), as has the less proprietary OMG component specification (as yet not implemented in commercial products). Component technologies are also emerging in more narrowly-scoped market sectors, for example geographic information systems.[9]

There are also technological approaches to resolving this tension. One approach is to treat component frameworks as an assembly of components rather than as a monolithic infrastructure. Robben proposes a scheme based in *meta-object protocols* (MOP) for composing quality attributes from components that implement each such attribute [Robben 98]. Venkatasubramanian also proposes MOP as a means of parameterizing frameworks to allow dynamic modification of protocols to accommodate context-specific quality attribute requirements [Venkatasubramanian 98]. Thompson et. al. describe an approach that dynamically interposes plug-ins between web clients and servers to provide quality of service attributes, yielding what they call an intermediary architectural approach to frameworks [Thompson 98]. These proposed techniques could be understood in the context of the "framework as operating system" analogy: each attempts to extract what should be separately configurable (parameterizable) components from the framework micro-kernel. Berry and Kaplan treat the framework as a programmable abstraction rather than an assembly of components (i.e., "programming the middleware") [Berry 98]. This is also a plausible approach that has been seen elsewhere. See Bergstra, for example [Bergstra 95].

Each of the above approaches focuses on making component frameworks more adaptable. If successful, these techniques would result in fewer standard frameworks, each of which can be adapted to a variety of quality attribute and quality of service requirements. Another approach is to make the components more flexible so that they may be more easily adapted to different frameworks. Deline developed a method and proof of concept prototype for *flexible packaging* to separate the framework-specific aspects of a component (the "packaging") from the specific function implemented by the component (the "ware") [Deline 99]. The separation of packaging from ware is also reflected in the more general idea of aspect-oriented programming (AOP), which allows separation of the core functionality of a module from a specification of various non-functional aspects (concurrency, persistence, distribution and so forth) [Kiczales 97b]. An aspect weaver, essentially an intra-component composition mechanism, composes component implementations from the functional core and a set of aspects.[10] Tarr et. al. describe an alternative approach to achieving separation of concerns

---

[9] See for example, http://www.esri.com/news/arcnews/winter9899articles/02-arcinfov8.html.

[10] See http://wwwtrese.cs.utwente.nl/aop-ecoop98/position.html to get a sense of the interest in AOP.

which they refer to as *hyperslices* and *hypermodules* [Tarr 99]. A hyperslice is an encapsulation of some dimension of concern; hyperslices, with the aid of a composition rule, are composed into hypermodules. Each of the above approaches is predicated on separating the functional core of a component from other concerns that may be varied independent of component functionality.

Summary of Key Points

Component models specify the design rules that must be obeyed by components. These design rules improve composability by removing a variety of sources of interface mismatch (i.e., mismatched assumptions). The rules ensure that system-wide quality attributes are achieved, and that components may be easily deployed into development and runtime environments. Component frameworks provide the services to support and enforce a component model.

A robust market in software components will require standard component models and frameworks. However, experience has shown that different application domains have different requirements for performance, security, availability and other quality attributes. This argues the need for more than one, and possibly many component models and frameworks. Market forces are working to find application domains that are sufficiently large and coherent to justify an industry-standard component model, but no clear winners have yet emerged. Technologies to support adaptability of frameworks to different quality attribute requirements may serve to reduce the need for competing frameworks. Similarly, technologies to support adaptability of components to different frameworks may reduce the adverse consequences of a fragmented framework market.

8 Composition

Composition, rather than integration, is the term used in component-based development to refer to how systems are assembled. Other than being more evocative of a development process based on software building blocks, there is no inherent difference between these terms. Nevertheless, the term "composition" is preferred in this report because it is so widely used in the software component technology literature.

Components are composed so that they may interact. As discussed earlier, the component model specifies patterns' types of components and their allowable patterns of interactions. We first examine a classification of these patterns of interaction in order to clarify the connection between component model and component composition. We then turn to a discussion resource binding, the mechanism by which composition takes place, and in particular we discuss the time of resource binding.

8.1 Compositional Forms
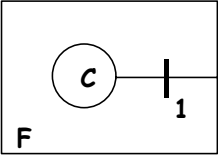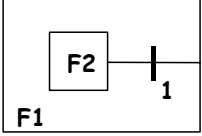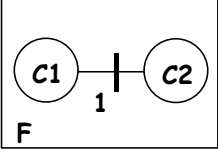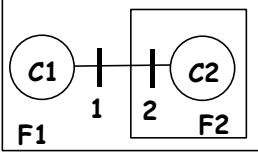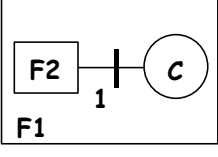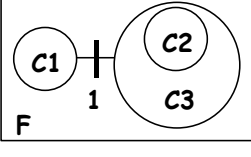
From the reference model depicted in Figure 1 we identify two kinds of entities that are composed: *components* and *frameworks*. Given this, there are three major classes of interactions that arise in component-based systems:

- **Component–Component (C–C)**: composition that enables interaction among components. These interactions deliver application functionality, and so the contracts that specify these interactions might be classified as *application-level* contracts.

- **Framework–Component (F–C)**: composition that enables interactions between a component framework and its components. These interactions enable frameworks to manage component resources, and so the contracts that specify these interactions might be classified as *system-level* contracts.

- **Framework–Framework (F–F)**: composition that enables interactions between frameworks. These interactions enable composition of components that are deployed in heterogeneous frameworks, and so these contracts might be classified as *interoperation* contracts.

There are additional special cases that arise if we allow higher-order components and frameworks. Higher-order components would allow component aggregates (assemblies) to be treated as "first-class" components (certified properties, contractual interface, independently deployable, etc.). Higher-order frameworks correspond to Szyperski's idea of tiered framework; this would become an essential concept should numerous application-specific component models emerge [Szyperski 98b].

The following table itemizes and briefly discusses the major classes of contracts, including the special cases. We refer to these as compositional forms

Table 1:   Compositional Forms

 Component Deployment	Components must be deployed into frameworks before they can be composed or executed. The deployment contract(s) (1) describes the interface that components must implement so that the framework can manage their resources.
 Framework Deployment	Frameworks may be deployed into other frameworks, corresponding to Szyperski's *Tiered Frameworks*. The EJB specification partially realizes this idea with EJB *containers* deployed into EJB *servers*. Contract (1) is analogous to the component deployment contract.
 Simple Composition	Components deployed in the same framework can be composed. The composition contract (1) expresses component- and application-specific functionality; the interaction mechanisms to support this contract are provided by the framework.
 Heterogeneous Composition	Support for tiered frameworks implies composition of components across frameworks, whether across hierarchical (as illustrated) or peer frameworks. In either case bridging contracts (1) are needed in addition to composition contracts (2) in order for interactions to span generic component models.
 Framework Extension (Plug-In)	Frameworks may be treated as components, and may be composed with other components. This form of composition most commonly allows parameterization of framework behavior via "plug-ins." Standard plug-in contracts (1) for service providers are increasingly common in commercial framework.
 Component (Sub)Assembly	A component-based system is an assembly of components. The ability to predict the properties of assemblies suggests a similar ability for subassemblies. Contract (1) is used to compose C1 and subassembly C3, which contains one or more components. A question that arises is whether C2 is visible outside of C3 and whether it is separately deployed.

The definitions in the above table beg the question, "Which of these forms of composition (and therefore which class of contracts) are, or should be, part of a component model?" The following comparison of component models is revealing. The comparison is not intended to be complete or exhaustive, but rather to reveal different perspectives on what should and should not be included in component models.

A ✓ indicates that the component technology includes this compositional form as part of the component model.

Table 2:   Compositional Forms in Illustrative Component Models

Form/ Technology	EJB	COM+	Java Beans	Water Beans	OMG/ Orbos
Component Deployment	✓	✓	✓	✓	✓
Framework Deployment	future (container contract)		✓ (JVM plug-in)		✓ (portable object adapter)
Simple Composition			✓	✓	
Heterogeneous Composition	✓ (IIOP)				✓ (IIOP)
Framework Extension		future (policy objects)			✓
Component (Sub)Assembly					

What can be deduced from this? First, each component technology defines deployment contracts. This is not surprising, since a component technology is not of much use unless components can be deployed into the build-time and run-time environments. Contracts for the other compositional forms are not uniformly included in component technologies. Note that none of the illustrated technologies addresses sub-assemblies in any way. Research papers

have been written on how to support assemblies of COM components, but these ideas have not found their way into commercial products [Peltz 99].

More surprising (at least at first blush) is the relative lack of attention to contracts for simple composition. Is it true that major technologies such as EJB and COM+ ignore application-level contracts even though component technology means nothing if applications can not be assembled from components? In truth, there is a gray area here. EJB and COM+ clearly say something about application-level contracts; for example, how transactions are initiated and propagated. But both EJB and COM+ are relatively silent about what the interfaces to components look like. To understand this statement it is useful to compare EJB with JavaBeans and WaterBeans, each of which requires component interfaces to conform to a particular syntactic convention, input and output events in the case of JavaBeans, input and output pipes for WaterBeans.

Because JavaBean and WaterBean components adhere to these conventions, components can be written with a high degree of independence from each other, and the assembly of applications from components can be left to an assembly environment such as the JavaBeans "BeanBox." Lacking these conventions, however, EJB component developers must make some assumptions about how other EJB components will present their services, and these assumptions must be embedded in the implementation of EJB components. The same, of course, is true of COM+. The result is a higher degree of coupling between components and reduced prospects for a market in commodity components. Instead, EJB and COM+ will rely upon companies such as the Theory Center to develop product lines of components, or, perhaps, for industry groups to develop application-level contracts for particular market niches.

8.2 Binding Time of Composition

The above discussion touched on *what* is composed. A discussion of *how* composition occurs involves the question of *resource binding*. Essentially, two components are composed when the resources provided by one component become accessible to a client component, or are bound to the client. Naming services are one mechanism for resource binding. However, to be complete, even after a naming lookup, many other mechanisms are involved in binding the client to the service provider (dynamic libraries, remote procedure calls, etc.).

Although the many mechanisms involved in resource binding can present a confusing picture of the binding process, one useful way to simplify things is to consider the question of *when* resource binding occurs. We can think of binding time as a timeline, with early binding at one end and late binding at the other. Looking at the development process from the perspective of software components, early binding requires that the component developer make some decisions which effectively bind some resource to the component, or, more to the point, constrain how this binding will occur later. Late binding means just the opposite: the component developer will make no decisions that constrain future resource binding.

```
                        System
                        Assembly
  Early ◄─────────────────────────────► Late
              |                      |
          Component              Application
          Development              Runtime
```

Two concrete examples will illustrate this distinction. In the case of EJB components, resource binding for application-level (i.e., simple-composition) assembly requires early binding because the component developer must make explicit reference to the interface of the type of component that will be provide some service. Although the actual resource binding may happen at runtime, perhaps as a result of a naming service lookup, this binding is constrained to only those components that implement a particular interface. Moreover, the EJB component model does not specify this interface, so it is likely to be non-standard. In the case of WaterBeans components, resource binding for application-level assembly requires no early binding decisions on the part of component developers, since all components must write obtain their input and deliver their output to typed pipes—this is required by the WaterBeans component model.

However, the component model must be designed to encourage late binding. Component technologies such as WaterBeans and JavaBeans support late binding by imposing additional constraints on component developers on how components expose their services, and on how components may interact once their services are bound to each other. Thus, the price for late binding is additional complexity in the component model, and added constraints on the way components are developed and on how they expose their services to potential clients. In exchange, component models that allow late binding offer greater flexibility for component substitution, for third-party integration, and for component markets.

Interestingly, the design constraints needed to support late binding of component resources represent a kind of *early binding of design decisions*, in particular those that deal with how components coordinate their activities with each other. This early binding of design decisions is consistent with the overall "architecture first" philosophy of CBSE, and leads to systems with properties that are more readily analyzed and predicted *prior to system assembly*.

Summary of Key Points

Composition enables interaction among the composed entities. In component-based systems the primary entities of concern are components and frameworks. There are six significant forms of interaction (and hence composition) between components and frameworks that arise in component-based systems. A component model will define contracts that govern these six forms of interaction. However, looking at representative software component technologies makes clear that although all define deployment contracts, there is no consensus what other kinds of composition should be supported, and none of the technologies examined dealt with component subassemblies.

The different treatments of application-level contracts are likewise revealing. The major commercial component technologies, EJB and COM+, deal with some aspects of application-level composition, but leave most of these details to be decided by the component market. This is an understandable position to take, given that each vendor wants to appeal to as broad a market as possible. Component technologies such as JavaBeans and WaterBeans take a different course, and constrain both the syntactic forms of a component's application-level interface and the coordination model underlying component-component interactions.

These different treatments of application-level interfaces are closely related to the question of binding time of composition. The more constraining application-level contracts of Water-Beans and JavaBeans allows late binding of components to each other, while the less constraining application-level contracts of EJB and COM+ require component developers to bind their components to specific, possibly non-standard interfaces, at component development time. In general, late binding for application-level composition is preferred to early binding, because it better supports component substitution and the development of component markets. Late binding of application-level composition also leads to better prospects for predicting overall system qualities prior to composition, precisely because late component binding requires early binding of architectural decisions in a component model.

9 Certification

The opaqueness of implementation details means that system assemblers may find it difficult to diagnose the causes of unexpected system behavior, and having diagnosed the cause affect repairs. This is certainly true where conventional COTS products are used (see Plakosh and Hissam for illustrations), and use of software component technology does not fundamentally alter this situation [Plakosh 99b, Hissam 99]. Given the difficulty and expense of diagnosing these kinds of failures, the long-term success of software component technology will hinge upon the availability of high-quality software components, and *trust* on the part of consumers that the components that they purchase are of high quality. This leads to the topic of *certification* of components.

Software certification is not a new idea. In fact, certification has a long history in the software industry. In the most frequently encountered use of certification, some authoritative organization attests to the fact that some software system satisfies some criteria, or conforms to some specification. For example, the National Security Agency (NSA) might certify that a particular operating system complies with level B2 of the Trusted Computer Security Evaluation Criteria (TCSEC) a.k.a., Orange Book Criteria.

This illustration reveals two separate and distinct facets of certification:

1) Technical claims are made about the subject of the certification.

2) An authority stands behind these claims to generate trust in these assertions.

These different facets are important insofar as they reveal the dual purpose of certification. The first is to establish *facts* about the subject being certified, and the second is to establish *trust* in the validity of these facts. While engineers may make use of established facts about a system, trust has other purposes. While both of these facets apply equally to certification in the context of component-based systems, the approaches that are needed to make assertions and build trust may be different than is otherwise the case.

9.1 Certification in Component-Based Systems

The NSA example might be termed the *traditional* approach to certification. In the traditional approach, the subject of the certification is a *system*. An authority attests to the fact (certifies) that the system satisfies some objective target criteria.

But matters are not so straightforward in a component-based approach. For one thing, the subject of certification includes, in addition to the end system, the components and frameworks that make up the end system.
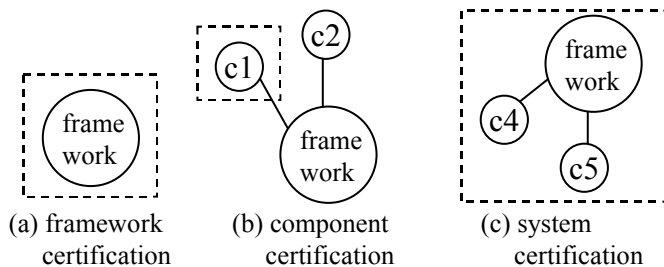

Figure 4:   Subjects for Certification in Component-Based Systems

This differentiation of certification subjects is already having in impact on industrial software, in particular for certification of components (if we momentarily equate component with commercial software product). For example, in response to the pressures of the COTS software marketplace, NSA and the National Institute of Standards and Technology have joined to create the National Information Assurance Partnership (NIAP) [11] to augment, if not wholly replace, the traditional NSA security certification regime. One of the noteworthy approaches of NIAP is the idea of a *protection profile*, which is essentially a slice through the TCSEC.[12] One use of protection profiles is to describe certification criteria for particular classes of COTS products. For example, NIAP has developed and approved a protection profile for COTS *firewalls*. At least eleven vendors are using NIAP certification against these criteria for competitive advantage.[13]

But while NIAP has begun to address the question of certifying the security properties of COTS components, it is instructive to note the limitations of the NIAP approach:

- Protection profiles are produced *as a response* to the emergence of classes of commercial products. Therefore, the scope of protection profiles will be only as stable as the products from which they emanate. Also, since product vendors will continue to introduce non-standard and differentiating features, the profile is also bound to provide only partial coverage of the security properties of products, and the mapping from product features to profile criteria will often be highly subjective.

- NIAC has partitioned the TCSEC into product-specific profiles, but has *not* produced the means of aggregating the different profiles exhibited by different products into a coherent system-level model of security. How should products be combined that have overlapping

---

[11] See http://niap.nist.gov/howabout.html for NIAP homepage.

[12] Actually, it is a slice through NIAP *common criteria*, which are derived from but reformulate and generalize the TCSEC.

[13] See http://www.lucent.com/press/0199/990120.coa.html for a vendor press release, and http://www.niap.nist.gov/cc-scheme/ValidatedProducts.html for a list of certified products.

profiles? Which criteria of one profile are dependent on criteria of other profiles? Most importantly, how are *system* security properties predicated on the properties of the individual components?

These failings of NIAP are not the result of naïveté on the part of NIAP, but rather stem from the fact that NIAP is a reaction to, and is bound up with, the dynamics of the COTS software marketplace. This marketplace differs from the software component marketplace in one important respect: the COTS marketplace is *products first*, *architecture second*; the component marketplace it is application *architecture first*, *components second*. That is, unlike COTS software, software components must comply with a component model that expresses numerous architectural decisions. This difference provides a foundation upon which a component-based certification regime can be established.

To understand this assertion it is useful to re-examine our motivation for certifying components and frameworks. We then relate this motivation to software architecture and to the benefits of the "architecture first" approach inherent in CBSE. Only then do we turn to a discussion of the technical issues involved in certification.

9.2 Certification, Prediction, and Compositional Reasoning

The unspoken premise behind component[14] certification is that there is a causal link between those properties of a component that are certified and the properties of an end system that uses that component. The more confidence we have in this link the more value will accrue from component certification. At the extreme we may achieve 100% confidence (or trust) in the causal link. At this extreme, once a property has been established for a component it is unnecessary to certify that the end system has obtained this same property, since we know this to be true *by definition*.

It is unlikely that 100% confidence will be achieved for all (if any) of the different kinds of properties of interest. In the absence of 100% confidence we move from the realm of certainty to the realm of probability and prediction. In this realm the value of component certification is proportional to the strength of the predictions that can be made about end-system properties. Consider the graphic at right. System S comprises two components, C1 and C2. C1 possesses property p1 and p2, while C2 has property p3. Our interest is in ensuring that system S exhibits property p4. In the illustration we assert that p1 and p3 are causally linked to p4.
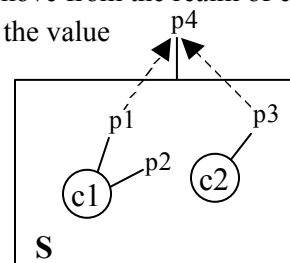
For the above diagram, assume that property p4 is end-to-end latency. If property p1 and p3 refer to the quality of documentation of components C1 and C2, then the link between p1, p3

---

[14] The following discussion applies equally to components *and* frameworks. We refer only to components to avoid excessively awkward phraseology.

and p4 is non-existent and certification of documentation quality would be of no value. Alternatively, if property p1 and p3 refer to some performance attributes of C1 and C2 that contribute to end-to-end latency, then the value of certifying these performance properties increases somewhat. How much this value increases depends upon the strength of the theory we will use to predict p4 from the values of p1 and p3. If we have a theory that can predict the latency p4 from latency p1 and p3 with only a small margin of error, then our knowledge of p1 and p3 is useful. Conversely, if our theory is weak then our knowledge of p1 and p3 is much less useful.

Such theories of prediction support what we refer to as *compositional reasoning*. The adjective compositional reflects the belief (as software architects have long asserted) that end-system properties are most often attributable to a collection of interacting components rather than to a single component. Thus, the properties of these several parts must be combined ("composed") to predict the properties of the whole. There are well-established prediction theories, for example Rate Monotonic Analysis (RMA), that support compositional reasoning about performance attributes of systems.

Note that the value of these theories goes beyond predicting system properties. Such theories also tell us which properties can be predicted (p4), and, just as important, which component properties we need to know about in order to make these predictions (p1 and p3). *The value of a certification regime for component-based systems is directly linked to the strength of our compositional reasoning*. Without compositional reasoning we can not know which properties of components and frameworks to certify. With weak compositional reasoning the value of certification will be questionable and therefore the economic incentives for third-party certification will never be sufficient to spur industry investment.

9.3 Compositional Reasoning and Software Architecture

The connection between compositional reasoning and software architecture was just alluded to—the motivation underlying the study of software architecture is prediction of system properties from the types of components in a system and their patterns of interaction. One promising research avenue, attribute-based architecture styles (ABAS), seeks to make compositional reasoning based on architectural decisions more formal, or at least more structured [Klein 99]. In brief, using ABAS terminology, an ABAS is an architectural style and an associated attribute-reasoning framework. An ABAS has four major parts:

- a description of the analysis problem solved by the ABAS

- a characterization of the stimuli to which the ABAS responds and the quality attribute measures of the response

- a description of the architectural style in terms of its components, connectors, topologies, and their properties. This will be used to structure the analysis.

- a reasoning framework that links stimuli and architectural properties to response. The rigor of these frameworks range from heuristics to mathematical formulae.

The fourth bullet contains the theory and compositional reasoning that relates properties of components and frameworks (the second and third bullets) to end-system properties (the response in the second bullet).

There is a useful connection between ABAS and software component technology in that a component model expresses architectural decisions that are imposed on component (and framework) developers. If these component models are equipped with an ABAS-style reasoning framework, then two things become possible. First, application builders can use the reasoning framework(s) bundled with the component model to predict end-system properties (one facet of the SEI vision for CBSE). Second, the reasoning framework(s) identifies those properties that must be known about components and frameworks, and hence certified.

More will be said about the connection of ABAS to a certification regime in Volume III of this report.

9.4 Certifying Components and Component Frameworks

Certification depends upon compositional reasoning. The theories underlying compositional reasoning identify those properties of components and frameworks that are material for predicting some end-system properties (performance, safety, scalability, etc.). These properties will be contractually specified in a component model. Thus, certification of components and frameworks reduces to establishing that they conform to these contractual specifications. Nonetheless, at least three significant technical challenges must be addressed before this "reduction" can become a routine practice.

First, contractual specifications must be expressive enough to capture all of the properties imposed by frameworks that will lead to end system quality attributes, and measurement techniques must be in place to determine, to a desired level of confidence, the degree to which a component exhibits these properties. Interface specification techniques must be adopted that strike a practical balance between formality and usability. More formal specifications are more expressive but introduce severe transition challenges. Measurement requires that each property expressed in an interface be defined precisely according to some measurement scale. In addition, sufficient laboratory testing techniques must be defined to produce accurate measurements on software components—ideally where these components remain "black boxes."

Second, having specified and measured all of the relevant properties, it is still necessary to define the *conformance* relation between a component and a contractual specification of a role it must fill. Some aspects of conformance are straightforward: a component property that over-satisfies a minimum threshold on a criterion should, we expect, be conformant. But

what if the desired property is associated with an interaction among three components, for example, and the entire interaction must occur within 60ms for the property to be satisfied? Is it proper to assume that each component must execute within 20ms, or might one execute in 10ms while each of the others execute in 25ms? In this case it might be best to certify the performance *value* of a component without imposing a minimum threshold. But without a minimum threshold how can conformance be defined? This suggests that components and frameworks have properties whose values are certified, but that the question of conformance needs to be deferred until later in the development process, possibly as late as runtime.

Third, component models are not monolithic, and a component technology may specify some aspects of a component model but leave other aspects to be specified by industry groups or application developers. Thus, as we saw earlier, the EJB component model is nearly silent about application-level contracts (*simple composition* in Table 1). WaterBeans is explicit about application-level contracts, specifying the form that these interfaces take, how they can be connected, and how control and data flow through these connections. By virtue of this greater detail, certifying components for WaterBeans will be more effective for predicting application-level properties than would be the case for EJB components. The point is that the component certification regime will be only as effective as component models allow. Moreover, *component models must be designed with component certification in mind.*

9.5 Process Certification

The entire discussion so far has focused on the direct technique of certifying software *products*. An alternative (or at least complementary) and indirect approach is to *certify the processes* used to produce the artifacts. Whether certifying a process will yield the level of trust that certifying a product will depends upon how strongly a causal link can be established between process quality and product quality. Should the link be strong, process certification may prove to be an economical alternative to product certification.

Note that process certification is not entirely conjectural. Underwriters Laboratory has published its standard for certifying *software in programmable components*[15] [UL 98]. While this standard addresses product certification in the traditional sense (inspections, testing, etc.), it also includes certification of the development process. For example, the development process is examined to determine that every process step has well-defined entry and exit criteria. The tools used by the development environment are also "qualified," meaning that they must have a documented "bug" list among other things.

Consider hypothetical components or frameworks being certified as "100% Cleanroom Developed." If it is established in the public eye that cleanroom development consistently pro-

---

[15] The term *programmable component* is defined by Underwriters Laboratory as "Any microelectronic hardware that can be programmed in the design center, the factory, or in the field. Here the term 'programmable' is taken to be 'any manner in which one can alter the software wherein the behavior of the component can be altered.' "

duces "zero defect software," then such a certification regime might be effective. Similarly, although this is a bit of a stretch, the process of assembling applications from certified software components might also be certified. This might be feasible should compositional reasoning about key properties (security, for example) be sufficiently trustworthy.

Summary of Key Points

Certification involves two separate actions: 1) establishing that a (software) subject possesses particular properties, most often in *conformance* with some specification against which a component or framework is certified; and 2) having a trusted agent or organization attest to the truth of this conformance. The first of these actions has consequence on the engineering discipline, in that application developers can work with "known quantities." The second of these actions has consequence on the development of consumer trust. *Both* are necessary.

The ultimate motivation for establishing that a component or framework possesses some property is to use that knowledge to predict some property of the assembled application. Theories and techniques that can link component and framework properties to application properties are said to support *compositional reasoning*. Compositional reasoning identifies which properties of components and frameworks are material to achieving some end-system property, and how to predict the "values" of end-system properties from component and framework properties. The effectiveness of a certification regime improves as our ability to undertake compositional reasoning improves.

The NIAP approach to certifying the security properties of COTS software products is a response to the industry trend towards the use of software components. But the COTS software marketplace provides a "product first, architecture second" context for NIAP certification, leading to an unstable understanding of what properties must be certified for particular classes of products, and to weak compositional reasoning schemes. In contrast, software component technology provides an "architecture first, component second" context for certification. This shift in emphasis provides the means to overcome some of the deficiencies of the NIAP approach.

One such promising means is attribute-based architectural styles (ABAS). An ABAS couples attribute-specific compositional reasoning techniques with architectural styles. Since component models are specifications of architectural decisions, one or more ABASs can be associated with a particular component technology. This in turn would identify those properties of the framework and of components that must be certified, and define how these certified properties can be used to predict the properties of the assembled application.

Using ABASs as a foundation for a component-based certification regime requires three technical issues be addressed. First, techniques for contractual specification must be extended to include any and all properties required by an ABAS, and measurement techniques must be identified to establish whether components exhibit these properties. Second, conformance must be re-defined to allow assemblers to decide whether a component or frame-

work "passes muster." In effect, we must certify the values of component or framework properties without necessarily superimposing normative guidelines on what these values must be. Third and last, component models (and hence component technologies) must be defined *a priori* to support a certification regime. Failing this, components will not be easily certified, or the causal link between component property and system property will be weak.

10 Conclusions

Component-based systems are the result of structuring a system according to a particular design pattern. This design pattern relies upon a number of closely related technical concepts, including: *components*, component *frameworks*, component *models*, component *composition*, component *interfaces*, *contracts*, and *certification*.

There are several motivations for the component-based approach to software. Components and industry standard component models provide a basis for commerce in reusable software, something that is required if US industry is to meet current and projected demand for software. Components are replacement units in systems, thus facilitating the development of software systems that are more flexible and less likely to become prematurely obsolete. Component models express numerous design decisions, and component frameworks provide an integrated environment for component execution; together these serve to drastically reduce the time it takes to design, implement and deploy systems.

Despite these potential benefits, one critical aspect of component-based software has yet to be adequately addressed: the need for systems that will predictably exhibit the quality attributes required of them. Consumers of commercial component technologies must rely upon off-the-shelf component models and frameworks to provide the foundation for achieving system quality attributes. While component technology vendors make claims about how their products yield these attributes, the technical basis for these claims is questionable. There is, as yet, no established basis for assessing how well component models and frameworks contribute to achieving desired quality attributes; nor is there any current basis for assessing the quality of software components and how they contribute, or hinder, achieving these quality attributes.

In light of this, the SEI has adopted a "vision statement" for component-based software engineering that focuses on the question of *predictable system assembly* from *certified components* and *frameworks*. This vision statement provided the filter through which a wide array of technical concepts was examined. The technical concepts discussed in this report culminate in a discussion of component and framework certification, and how the "architecture first" approach inherent in component-based systems provides a basis for certification and predictable assembly. A detailed description of a technical agenda to build on this basis is the topic of Volume III of this report.

References

[Achermann 99]   Achermann, F.; Lumpe, M.; Schneider, J.; & Nierstrasz, O. *Piccola — A Small Composition Language* [online]. Available WWW <URL: http://www.iam.unibe.ch/~scg/ Research/Piccola/pascl.pdf.> (1999).

[Alencar 92]   Alencar, A. & Goguen, J. "OOZE," Stepney, S.; Barden, R.; & Cooper, D., ed. *Object Orientation in Z, Workshops in Computing.* Los Angeles, Ca.: Springer-Verlag, 1992.

[Allen 97]   Allen, R.; Douence, R.; & Garlan, D. "Specifying Dynamism in Software Architectures," *Proceedings of the 1st Workshop on Component-Based Systems.* Zurich, Switzerland, 1997, in conjunction with *European Software Engineering Conference (ESEC)* and *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE),* 1997 [online]. Available WWW <URL: http://www.cs.iastate.edu/~leavens/FoCBS/FoCBS.html>

[Baggiolini 97]   Baggiolini, V. & Harms, J. "Toward Automatic, Run-Time Fault Management for Component-Based Applications," *Proceedings of the 2nd International Workshop on Component-Oriented Programming (WCOP97),* in conjunction with the *European Conference on Object-Oriented Programming (ECOOP98,* Brussels, Belgium, July 1998.

[Barnes 97]   Barnes, J. *High Integrity Ada: the SPARK Approach*. Boston, Ma.: Addison-Wesley, 1997

[Bass 98]   Bass, L; Clements, P.; & Kazman R. *Software Architecture in Practice*. Boston, Ma.: Addison Wesley, March 1998.

[Bellur 98]   Bellur, U. "The Role of Components & Standards in Software Reuse," *Proceedings of OMG-DARPA-MCC Workshop on Compositional Software Architecture*. Monterey, Ca., Jan. 1998.

[Bergstra 95]   Bergstra, J. & Klint, P. *The Discrete Time Toolbus.* **(**Report p9502)
Amsterdam, the Netherlands: Programming Research Group, De-
partment of Mathematics and Computer Science, University of
Amsterdam, 1995.

[Berry 98]   Berry, S. "Programming the Middleware Machine with Finesse,"
*Proceedings of OMG-DARPA-MCC Workshop on Compositional
Software Architecture.* Monterey, Ca., January 1998.

[Beugnard 99]   Beugnard, A.; Jezequel, J.-M.; Plouzeau, N.; & Watkins, D.
"Making Components Contract Aware." *Computer 32*, 7 (July
1999): 38-45.

[Box 98]   Box, D. *Essential COM*. Boston, Ma.: Addison-Wesley, 1998.

[Canal 97]   Canal, C.; Pimentel, E.; & Troya, J. "On the Composition and
Extension of Software Components," *Proceedings of the 1st
Workshop on Component-Based Systems,* Zurich, Switzerland,
1997, in conjunction with *European Software Engineering Con-
ference (ESEC)* and *ACM SIGSOFT Symposium on the Founda-
tions of Software Engineering (FSE),* 1997. Available WWW
<URL: http://www.cs.iastate.edu/~leavens/FoCBS/FoCBS.html>

[Ciancarani 97]   Ciancarani, P. & Cimato, S. "Specifying Component-Based Soft-
ware Architectures," 60–70. *Proceedings of the ESEC/FSE-
Workshop on Foundations of Component-Based Systems
(FoCBS),* Zürich, Sep. 1997.

[Deline 99]   Deline, R. "Avoiding Packaging Mismatch with Flexible Packag-
ing," *Proceedings of the 21st International Conference on Soft-
ware Engineering.* Los Angeles, Ca., May 1999.

[Della 99]   Della, C.; Cicalese, T.; & Rotenstreich, S. "Behavioral Specifica-
tion of Distributed Software Component Interfaces." *IEEE Com-
puter* (Jul. 1998): 46-53

[Dowson 87]   Dowson, M. "ISTAR and the Contractual Approach," 287-288.
*Proceedings of the 9th International Conference on Software En-
gineering.* Monterey, Ca, March 30-April 2, 1987. Washington
DC, Baltimore, Md.: IEEE Computer Society and the Association
for Computing Machinery, April 1987.

[D'Souza 99]   D'Souza, D. & Wills, A.C. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Boston, Ma.: Addison-Wesley, 1999.

[Duke 95]   Duke, R.; Rose, G.; & Smith, G. "Object-Z: A Specification Language Advocated for the Description of Standards." *Computer Standards and Interfaces 17* (1995): 511–533

[Dürr 94]   Dürr, E. H. & Plat, N. (editors), *VDM++ Language Reference Manual.* Utrecht, The Netherlands: Cap Volmac, Mar. 1994.

[Fellner 99]   Fellner, K. & Turowki, K. "Component Framework Supporting Inter-company Cooperation," *Proceedings of the Third International IEEE Conference on Enterprise Distributed Object Computing.* Mannheim, Germany, Sept 1996. New York: IEEE Computer Society Press, 1999.

[Fischer 99]   Fischer, C. "Software Development with Object-Z, CSP and Java: A Pragmatic Link from Formal Specifications to Programs," *Proceedings of the 1st Workshop on Component-Based Systems.* Zurich, Switzerland, 1997, in conjunction with *European Software Engineering Conference (ESEC)*, *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 1997, *Proceedings of the 1st Workshop on Formal Techniques for Java Programs*, in conjunction with *the 13th European Conference on Object-Oriented Programming, ECOOP '99* [online]. Available WWW <URL: http://semantik.informatik.uni-oldenburg.de/ persons/clemens.fischer/eindex.html>

[Franch 98]   Franch, X. "Systematic Formulation of Non-Functional Characteristics of Software," *Proceedings of the 3rd IEEE International Conference on Requirements Engineering (ICRE).* Colorado Springs, Co., April 1998. New York: IEEE Computer Society Press, 1998.

[Gannon 98]   Gannon, D. "Component Architectures for High Performance, Distributed Meta-Computing," Proceedings of OMG-DARPA-*MCC Workshop on Compositional Software Architecture.* Monterey, Ca., Jan. 1998.

[Garlan 98]   Garlan, D. *"Higher-Order Connectors," Proceedings of OMG-DARPA-MCC Workshop on Compositional Software Architec-*

---

ture. Monterey, Ca., Jan. 1998.

[Goldsack 97]   Goldsack, S. J.; Lano, K.; & Dürr, E. "Invariants as Design Templates in Object-Based Systems," *Proceedings of the 1st Workshop on Component-Based System.* Zurich, Switzerland, September 1997, in conjunction with *European Software Engineering Conference (ESEC)* and *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 1997 [online]. Available WWW <URL: http://www.cs.iastate.edu/~leavens/FoCBS/FoCBS.html>(1997)

[Henderson 97]   Henderson, P. "Formal Models of Process Components," *Proceedings of the 1st Workshop on Component-Based Systems,* Zurich, Switzerland, Sep. 1997, in conjunction with *European Software Engineering Conference (ESEC)* and *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE),* 1997 [online]. Available WWW <URL: http://www.cs.iastate.edu/~leavens/FoCBS/FoCBS.html> (1997).

[Hissam 99]   Hissam, S & Carney, D. "Isolating Faults in Complex COTS-based Systems." *Journal of Software Maintenance: Research and Practice,* No. 11 (1999): 183-1999

[Holland 92]   Holland, I. "Reusable Components using Contracts," *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).* Lecture Notes in Computer Science LNCS 615. Los Angeles, Ca.: Springer-Verlag, June/July 1992.

[Johnson 97]   Johnson, D. & Kilov, H. "An Approach to an RM-ODP Toolkit in Z," *Proceedings of the 1st Workshop on Component-Based Systems.* Zurich, Switzerland, 1997, in conjunction with *European Software Engineering Conference (ESEC)* and *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE),* 1997 [online]. Available WWW <URL: http://www.cs.iastate.edu/~leavens/FoCBS/FoCBS.html> (1997).

[Kiczales 97]   Kiczales, G. "Aspect-Oriented Programming," *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP). Lecture Notes in Computer Science LNCS 1241. Los Angeles, Ca.: Springer-Verlag, June 1997.

[Klein 00]   Klein, M; Kazman, R.; & Nord, R. "A BASis (or ABASs) for Reasoning about Software Architectures," *submitted to the 22nd*

International Conference on Software Engineering (ICSE). Limerick, Ireland, 2000.

[Klein 99]   Klein, M. & Kazman, R. *Attribute-Based Architectural Styles* (CMU/SEI-99-TR-022). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1999. Available WWW <URL: http://www.sei.cmu.edu/publications/documents/ 99.reports/99tr022/99tr022title.html> (1999).

[Koutlis 98]   Koutlis, M.; Kourouniotis, P.; Kyrimis, K.; & Renieri, N. "Inter-Component Communication as a Vehicle Towards End-User Modeling," *Proceedings of the 1ˢᵗ ICSE Workshop on Component-Based Software Engineering.* Kyoto, Japan, 1998 [online]. Available WWW <URL: http://www.sei.cmu.edu/cbs/icse98/papers/ p7.html> (1998).

[Kramer 98]   Kramer, R. "iContract — The Java design by Contract Tool." *TOOLS 26: Technology of Object-Oriented Languages and Systems*, IEEE Computer Society Press, Los Alamitos, Ca. (1998): 295–307.

[Lano 97]   Lano, K.; Bicarregui, J.; Maibaum, T.;& Fiadeiro, J. "Composition of Reactive System Components," *Proceedings of the 1ˢᵗ Workshop on Component-Based Systems*. Zurich, Switzerland, 1997, in conjunction with *European Software Engineering Conference (ESEC) and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE),* 1997 [online]. Available WWW <URL: http://www.cs.iastate.edu/~leavens/FoCBS/FoCBS.html>

[Lauder 99]   Lauder, A. & Kent, S. "EventPorts: Preventing Legacy Componentware," *Proceedings of the Third International IEEE Conference on Enterprise Distributed Object Computing*. Mannheim, Germany, Sep. 1996.

[Leavens 99]   Leavens, G. T.; Baker, A. L.; & Ruby, C. *Preliminary Design of JML: A Behavioral Interface Specification Language for Java,* (Technical Report 98-06c) Ames, Ia.: Iowa State University, Department of Computer Science, Jan. 1999 [online]. Available WWW <URL: http://www.cs.iastate.edu/~leavens/JML/ prelimdesign/index.html>

[Leavens 97]   Leavens, G. & Sitaraman, M. (eds*.), Proceedings of the ESEC/FSE-Workshop on Foundations of Component-Based Systems (FoCBS).* Zürich, Switzerland, Sep. 1997. Available WWW <URL: http://www.cs.iastate.edu/~leavens/FoCBS/FoCBS.html >

[Lumpe 97]   Lumpe, M.; Schneider, J; Nierstrasz, O; & Achermann, F. "Towards a Formal Composition Language," *Proceedings on the 1ˢᵗ Workshop on Component-Based Systems*. Zürich, Switzerland, Sep. 1997. Available WWW <URL: http://www.cs.iastate.edu/~leavens/FoCBS/FoCBS.html >

[Lycett 98]   Lycett, M., & Paul, R. J. "Component-Based Development: Dealing with Operational Aspects of Architecture," 83-92. *Proceedings of the 3ʳᵈ International Workshop on Component-Oriented Programming* (WCOP '98). Brussels, Belgium, July 1998.

[Meyer 92a]   Meyer, B. "Applying 'Design by Contract.'" *Computer 25*, 10 (October 1992): 40-51.

[Meyer 92b]   Meyer, B. *Eiffel: The Language*. New York, NY: Prentice Hall, 1992.

[Meyer 97]   Meyer, B. *Object-Oriented Software Construction*, 2ⁿᵈ ed. London, UK: Prentice-Hall International, 1997.

[Mikhajlova 99]   Mikhajlova, A. "Specifying Java Frameworks Using Abstract Programs." *TOOLS 30: Technology of Object-Oriented Languages and Systems* IEEE Computer Society Press, Santa Barbara, Ca. (Aug. 1999) 136–145.

[Norman 98]   Norman, D. *The Invisible Computer: Why Good Products Can Fail, The Personal Computer is So Complex, and Information Appliances are the Solution*. Cambridge, Ma.: MIT Press, 1998.

[Olken 98]   Olken, F.; Jacobsen, H.; & McParland, C. "Middleware Requirements for Remote Monitoring and Control," *Proceedings of OMG-DARPA-MCC Workshop on Compositional Software Architecture*. Monterey, Ca., Jan. 1998.

[Parnas 71]   Parnas, D. "Information Distribution Aspects of Design Methodology." *Proceedings 1971 IFIP Congress*, North Holland Publishing Company.

[Peltz 99]	Peltz, C. "A Hierarchical Technique for Composing COM based Components," *Proceedings of the 2ⁿᵈ International Workshop on Component-Based Software Engineering (CBSE),* in conjunction with the *21ˢᵗ International Conference on Software Engineering (ICSE).* Los Angeles, Ca., May 17-18, 1999.
[Plakosh 99a]	Plakosh, D.; Smith, D.; & Wallnau, K. *Water Beans Component Builder's Guide* (CMU/SEI-99-TR-024). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1999. Available WWW <URL: http://www.sei.cmu.edu/ publications/documents/ 99.reports/99tr024/99tr024abstract.html>
[Plakosh 99b]	Plakosh, D.; Hissam, S.; & Wallnau, K. *Into the Black Box: A Case Study in Obtaining Visibility into Commercial Software* (CMU/SEI-99-TN-010). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1999. Available WWW <URL: http://www.sei.cmu.edu/publications/documents/ 99.reports/99tn010/99tn010abstract.html >
[Rice 99]	Rice, M. & Seidman, S. "Describing Software Architectures and Architectural Styles," *First Working IFIP Conference on Software Architecture (WICSA1)* San Antonio, Texas, Feb. 1999.
[Robben 98]	Robben, B.; Matthijs, F., Joosen,W.; Vanhaute, B.;& Verbaeten, P. "Components for Non-Functional Requirements," *Proceedings of the 3ʳᵈ International Workshop on Component-Oriented Programming* (WCOP '98). Brussels, Belgium, July 1998.
[Sha 95]	Sha, L.; Rajkumar, R.; & Gagliardi, M. *A Software Architecture for Dependable and Evolvable Industrial Computing Systems* (SEI-TR-95-005). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1995. Available WWW <URL: http://www.sei.cmu.edu/publications/documents/95.reports/ 95.tr.005.html>
[Shaw 99]	Shaw, M. "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.* Berlin, Germany, March 1996.
[Szyperski 98a]	Szyperski, C. & Vernik, R. "Establishing System-Wide Properties of Component-Based Systems," *Proceedings of OMG-DARPA-MCC Workshop on Compositional Software Architecture.* Monterey, Calif.,

Jan. 1998.

[Szyperski 98b]   Szyperski, C. *Component Software Beyond Object-Oriented Programming.* Boston, Ma.: Addison-Wesley and ACM Press, 1998.

[Tarr 99]   Tarr, P.; Ossher, H.; Harrison, W.; & Sutton, S. "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *Proceedings of the 21st International Conference on Software Engineering (ICSE99).* Los Angeles, Ca., 1999.

[Thompson 98]   Thompson, C.; Pazandak, P.; Vasudevan, V.; Manola, F.; Palmer, M.; Hanser, G.; & Ford, S. "Intermediary Architecture: Interposing Middleware Services And Ilities Between Web Client And Server," *Proceedings of OMG-DARPA-MCC Workshop on Compositional Software Architecture.* Monterey, Ca., Jan. 1998.

[UL 98]   Underwriters Laboratories. *UL Standard for Safety for Software in Programmable Components.* Northbrook, Il., 1998.

[Venkatasubramanian 98]   Venkatasubramanian, N. & Agha, G. "Composable QoS-Based Distributed Resource Management," *Proceedings of OMG-DARPA-MCC Workshop on Compositional Software Architecture.* Monterey, Ca., Jan. 1998.

[Weck 96]   Weck, W. "Independently Extensible Component Frameworks," *Proceedings of the 1st International Workshop on Component-Oriented Programming*, in conjunction with *the European Conference on Object-Oriented Programming (ECOOP97).* Jyväskylä, Finland, June 1997

[Wills 99]   Wills, A. "Modeling for Component Based Systems with Catalysis." Tutorial notes, *Enterprise Distributed Object Computing (EDOC).* Mannheim, Germany, 1999

[Yucel 98]   Yucel, S.; Kusano, T.; & Saydam, S. "A Component Based Distributed Software Architecture for Multimedia Services," *Proceedings of OMG-DARPA-MCC Workshop on Compositional Software Architecture.* Monterey, Ca., Jan. 1998.

REPORT DOCUMENTATION PAGE

1. **AGENCY USE ONLY** (LEAVE BLANK)	2. **REPORT DATE** May 2000	3. **REPORT TYPE AND DATES COVERED** Final

4. **TITLE AND SUBTITLE** Volume II: Technical Concepts of Components-Based Software Engineering, 2nd Edition	5. **FUNDING NUMBERS** C — F19628-95-C-0003
6. **AUTHOR(S)** Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, Kurt Wallnau	
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI –2000-TR-008
9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	10. **SPONSORING/MONITORING AGENCY REPORT NUMBER** ESC-TR-2000-007

11. **SUPPLEMENTARY NOTES**

12.A **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS	12.B **DISTRIBUTION CODE**

13. **ABSTRACT** (MAXIMUM 200 WORDS)

The Software Engineering Institute (SEI) is undertaking a feasibility study of "component-based software engineering" (CBSE). The objective of this study is to determine whether CBSE has the potential to advance the state of software engineering practice and, if so, whether the SEI can contribute to this advancement. This report is the second part of a three-part report on the study. Volume I contains a market assessment for CBSE. Volume III outlines a proposed course of action for the SEI. Volume II, this report, establishes the technical foundation for SEI work in CBSE. The paper asserts that the key technical challenge facing CBSE is to ensure that the properties of a system of components can be predicted from the properties of the components themselves. The key technical concepts of CBSE that are needed to support this vision are described: *component*, *interface, contract, component model, component framework, composition,* and *certification*.

14. **SUBJECT TERMS** component-based, component model, component framework, contract, composition, certification, CBSE	15. **NUMBER OF PAGES** 64

16. **PRICE CODE**

7. **SECURITY CLASSIFICATION OF REPORT** UNCLASSIFIED	18. **SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	19. **SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	20. **LIMITATION OF ABSTRACT** UL