

A uniform component modeling space

Duane Hybertson
 The MITRE Corp., McLean, VA, USA
 Phone: 703-983-7079, Fax: 703-983-1339
 dhyberts@mitre.org

Keywords: component, composition, generalization, interaction, model, modeling space, representation, specification

Received: June 5, 2001

This paper presents a component modeling space as a context for supporting component-based software development and accumulating component-related knowledge. The modeling space is structured in three dimensions: A representation dimension that ranges from the languages of problem domains to computer processor languages; a composition dimension that supports a repeating pattern of the whole-part, or system-component, hierarchy; and a generalization dimension that supports reuse of models and components. Also integrated into the modeling space are an interaction model of components and connectors, an approach to component specification, and a provision for relating models via mappings. Each of these elements is characterized as applying in a uniform way throughout the modeling space.

1 Introduction

The goal of component-based software development (CBSD) is to build software systems by integrating pre-existing software components. It is understood that reaching this goal requires reusable components that interact with each other and fit into system architectures. This sounds relatively straightforward but has proved difficult to achieve.

This paper briefly discusses some of the difficulties, and then presents elements of a modeling space intended to facilitate the resolution of these difficulties. The contribution of this paper is not based on the individual elements of the modeling space, because most of them have been described elsewhere. Rather, it is based on the selection and organization of these elements into an integrated structure, and on the uniform modeling approach emphasized in this structure.

Significant issues in CBSD include:

- **Integration:** How can components developed independently be integrated into a workable system?
- **Scope:** Are components restricted to a certain scope or size? Are “objects” or “procedures” too small? Are “subsystems” too large? How can we market and use integrated collections of components?
- **Problem set:** A component is intended to be used in multiple systems. How can we develop a component to support solving multiple problems, instead of just one problem? This is the basic reuse issue.
- **Shared understanding:** How do we know what a component does, and whether it will fit into our architecture? This is the basic specification problem.
- **Semantic gap:** Software development, including CBSD, must cover the spectrum from a problem domain representation to a machine language solution.

How can models bridge this gap, and what are necessary constraints on component representations?

Some of the issues listed above are not specific to CBSD. The discussion of these issues in this paper will focus on how the proposed modeling space benefits CBSD, but will also indicate broader software engineering benefits.

Goals and building blocks of the modeling space are presented in Section 2. Modeling space elements are then described in Section 3. Section 4 reviews related work, and concluding remarks are presented in Section 5.

2 Goals and building blocks

The modeling space definition is based on four goals or principles: (1) *Separation*: isolate elements important for successful long-term CBSD and define each element separately. (2) *Integration*: define the elements in a compatible way so they add up to a unified whole. (3) *Simplicity*: keep each definition as simple and uniform throughout the modeling space as possible. (4) *Universality*: find elements and definitions that apply to the full range of the component paradigm, rather than restricting the scope to any specific problem domain, life cycle phase, framework, or component model.

Several building blocks support the modeling space elements described in Section 3. These building blocks consist of three types of entities: problem domain entities, software entities, and description entities.

Problem domain entities: Elements or objects of interest in a problem domain, such as a bank account in the financial domain

Software entities:

- **Data:** Values interpreted as the state or properties of an entity or set of entities
- **Component:** Computational entity, i.e., performs operations on data
- **Port:** Point of interaction of a component with its environment, and through which a component provides or receives a service; structural part of component interface
- **Service:** Data operation(s) that may be performed by one component on behalf of another component; behavioral part of component interface
- **Connector:** Interaction entity, i.e., mediates communication and coordination among components; examples: remote procedure call, pipe, event broadcast
- **Role:** Name of behavior pattern that may be performed by a component in an interaction context; structural part of connector interface; examples: client, server
- **Protocol:** Specification of behavior pattern that may be performed by a component in an interaction context; behavioral part of connector interface
- **System:** Configuration of software entities

Description entities:

- **Model:** Explicit description of an entity or set of entities; may include entity properties
- **Specification:** Precise shared understanding of an entity or set of entities and entity properties; includes semantics
- **View:** Useful subset of an entity or set of entities

Software and description entities exist in the modeling space; problem domain entities do not. Services are defined separately from ports because services can be specified in the form of APIs that are defined separately from the components that may provide (implement) them or use them. Whether a description entity is a specification depends on the parties involved. If they share a common understanding, it is a specification.

3 Modeling space elements

The foregoing issues, goals, and building blocks led to these modeling space elements:

- An **interaction model** of components and connectors that addresses component interaction, coordination, and integration in a uniform way throughout the modeling space. This element addresses the CBSD integration issue.
- A **composition** spectrum that represents a whole-part hierarchy ranging from the most inclusive system of systems to the lowest level indivisible unit. It is recursive in that a given whole can be part of a larger whole. This element is related to the scope issue.
- A **generalization** spectrum that represents a “kind-of” or “is-a” hierarchy ranging from universal models to

instance models. It is recursive in that a model that is a generalization can in turn be further generalized. This element addresses the problem set issue.

- A **specification** approach that emphasizes contracts, precision, and semantics, and has two primary specification types or views for each component and connector: *external* and *internal*. The same kinds of specification information apply throughout the modeling space. This element addresses the shared understanding and integration issues.
- A **representation** spectrum that ranges from problem domain languages to computer processor languages. This spectrum covers not only a range of representations but also a range of conceptualizations. This element is related to the semantic gap issue.
- **Mappings** that capture knowledge about the relations among models, specifications, and views throughout the modeling space. This element is related to the semantic gap issue.

Composition, generalization, and representation collectively structure the modeling space into three dimensions as shown in Figure 1. They are separate dimensions because two entities can be at the same point on any two dimensions but differ on the third. This structure is proposed in place of the traditional temporal life cycle.

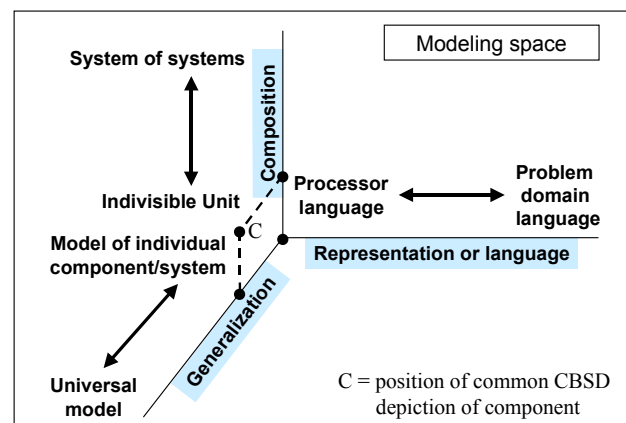


Figure 1. Modeling space dimensions

The definition of component is still a matter of debate in the CBSD community. A common view is that a component is a deployable (physical) entity that is larger than a class or object but smaller than a subsystem, and provides a specifically defined set of services but is reusable in multiple systems. This depiction corresponds to an area around point C in Figure 1, at or near the processor language end of the representation dimension and at intermediate levels of the composition and generalization dimensions. In contrast, the definition of component in this paper allows it to be anywhere in the modeling space.

Abstraction and levels of abstraction are important concepts in the modeling space, but the terms are rarely used in this paper. The reason is that four kinds of abstraction are part of the modeling space, corresponding

to the three dimensions plus views. Instead of using the general ‘abstraction’ term, each kind is discussed in its own context. Each dimension has a range of levels of its kind of abstraction.

A brief example will illustrate the modeling space separation of concerns. Suppose we design and build a financial system for First National Bank (see Figure 2). The system interacts with customers in setting up and using accounts, and sends certain reports to the Federal Reserve Bank (FRB) on a periodic basis. The system consists of hardware, software, and manual operations performed by customer service representatives. The software portion of the system consists of customer management and account management. Account management is composed of two parts: accounts and account transfers. In the course of developing this system, we specialize the party management facility defined by OMG for our customer management need, and then as we move into implementation, we find and incorporate a customer management component that satisfies our requirements for that part of the system. We develop the account management part, but when we are done, we decide this is a general capability that multiple banking systems could use. We generalize account management and make it available as a component with accompanying specification.

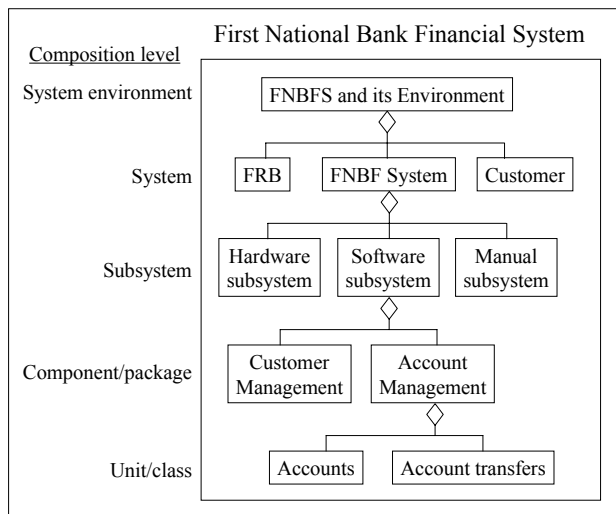


Figure 2. Example system

We will now briefly map the example to the modeling space dimensions. Figure 2 shows the *Composition* perspective. Using familiar terms for each level, the composition levels range from the system environment to the unit or class level. In the *Representation* dimension, models range from use cases expressed in English (on the problem domain end) to machine instructions expressed in binary on the machine end. Between these end points are design and implementation models in UML and Java. The Java bytecode is very close to the machine code in this spectrum. In the *Generalization* dimension, the focus of the example is on a single system, which places it at the specific end of the dimension. However, there are a few generalized elements. The party management facility

was a general model that we specialized for customer management, and the more specialized customer management was general enough to find an existing component to satisfy the need. We also generalized account management into a reusable component.

Each modeling space element in the list above will now be described in additional detail.

3.1 Interaction model

This section describes an interaction model of components and connectors that addresses the CBSD integration issue. At its most basic level, the component paradigm is about developing components and integrating them into systems in which the components interact. The interaction model supports the modeling of component interaction with two entity types: **components**, which serve as a locus of computation and decision-making, and **connectors**, which serve as a locus of interaction between components. Both entity types exist throughout the modeling space in all dimensions. Every box shown in the hierarchy in Figure 2 can be a component. Correspondingly, connectors define and facilitate interactions ranging from a procedure call or message passing to UNIX pipe-filter interactions to distributed system interactions. As a locus of interaction, a connector provides not just an exchange medium, but also specification of interaction roles and protocols.

Components and connectors have respective interface points called ports and roles, as shown in Figure 3. The left side of the figure shows a basic interaction of components A and B via a connector. The center is a visualization of the component-port-role-connector model. The right side shows specification elements in this structure (see Section 3.4). A key point is that *roles that a component plays in an environment are defined not by the component, but by the connector-specified interactions in which the component participates.*

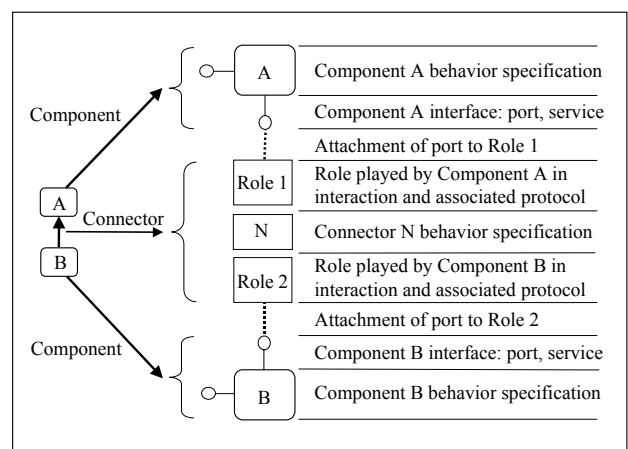


Figure 3. Anatomy of an interaction

The interaction model supports CBSD in two ways. First, the explicit treatment of interaction, connectors, and

coordination provides a basis for integrating components into systems by clearly defining the integration context. Second, the uniform nature of this model throughout the modeling space facilitates component modeling and the use of the component paradigm throughout the full spectrum from problem definition to deployment. Both of these benefits will become clearer in the ensuing discussion of the remaining modeling space elements.

3.2 Composition spectrum

This section describes a composition spectrum that addresses the CBSD scope issue. Software systems and components typically exhibit a whole-part hierarchy. The end points of this spectrum are the smallest component or unit that is not further divided and the most inclusive component or system of systems. Formally, system and component are synonyms. Informally, they can be used as relative terms. A system at one level may be a component of another system at the next higher level, and the same relations repeat at each level. Figure 4 illustrates the repeating pattern. The Figure 2 example shows the pattern repeated four times.

‘A’ represents a system of components B, C, D, and E interacting via connectors K, L, M, and N. ‘E’ in turn represents a system of its interacting components and connectors F, P, etc.. The figure shows a component as a composition of components and connectors. A connector can also be a composition of connectors and components. One mapping of the applicable parts of Figure 4 to the example in Figure 2 could be: E = FNBS system, D = FRB, N = asynchronous interprocess connector, C = customer, M = human-computer interaction, F = software subsystem, G = hardware subsystem, H = manual subsystem. Another mapping could be: E = account management, D = customer management, F = accounts, G = account transfers, P = message passing connector.

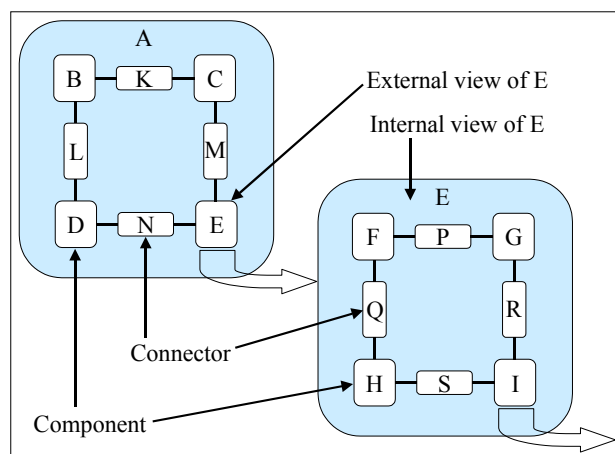


Figure 4. Recursive composition pattern

This spectrum provides a clear context for internal and external specifications (discussed in Section 3.4), and offers a uniform way to treat components at multiple levels. From the Figure 2 example, account management

could be a CBSD component in other systems in addition to the FNBF software system. The FNBF software system, since both of its components are generalized, could itself be a CBSD component in other financial systems. At each level, the internal view of each component is seen by the component developer, but is hidden from the system composer. The person producing E is a system composer when integrating I into E, but is a component developer when preparing E for users such as the system composer of A.

3.3 Generalization spectrum

This section describes a generalization spectrum that addresses the CBSD problem set issue. A key potential benefit of the component paradigm is reuse—a component is intended to be usable in multiple systems. The generalization spectrum supports this goal with the idea of general models and specifications. Generalization is a form of abstraction in which information is removed to make a more general component or model that is useful in multiple environments or that allows multiple implementations. General models include abstract data types, classes in class hierarchies, generics, templates, component and connector types, frameworks, reference models, domain specific architectures, product line architectures, analysis patterns, architecture/design patterns, architecture styles, and programming idioms.

Each of these is aimed at a goal that is difficult to achieve: solve a *group of problems* rather than a single problem. If we characterize a problem as a set of features or aspects, then the union of problem sets yields a problem space, and the intersection of the problem sets defines the features that are common among the problems in that space. The difference between the union and the intersection represents variation among the problems. If the intersection is small, the problem space is heterogeneous. The class of problem domains supported by software engineering is an important example of a heterogeneous problem space. If the intersection is large, the problem space is homogeneous. The class of hardware processors is an important example of a relatively homogeneous problem space.

In any problem space, we can identify subspaces that are more homogeneous than the complete space, and increase reuse in that subspace. Domain specific engineering is targeted to a homogeneous subspace of the overall problem space. There is a general tradeoff. We can achieve limited reuse across the whole set of problems, or we can achieve greater reuse within a more homogeneous subset of problems.

The modeling space approach to this tradeoff is a principle we will call maximum *leverage*. Leverage of a solution (e.g., a model or component) is defined as the degree to which it satisfies these two conflicting criteria: (1) number of problem situations to which it applies; and (2) proportion of solution it provides—i.e., extent to

which it provides the complete solution needed for the applicable problem set. Leverage as a metric is the product of these two criteria. This makes the tradeoff explicit. The concept is illustrated in Figure 5.

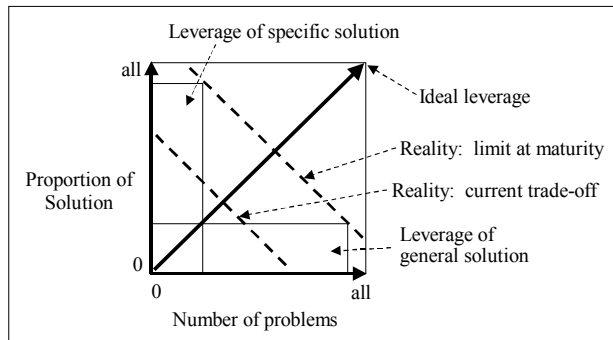


Figure 5. Leverage

Ideally, one solution would completely solve all problems (upper right corner of box). However, there is a limit even when a discipline reaches maturity—the tradeoff still exists. In an immature problem domain, the limit is not yet reached, and leverage is even more restricted. The current and ultimate tradeoffs are represented by the diagonal dashed lines in Figure 5. Two models of equal leverage may differ in that one may provide a small part of the solution for a large number of problems (shown as the box labeled “Leverage of general solution” in Figure 5), while the second may provide most of the solution for a small number of problems (shown as “Leverage of specific solution”). The first criterion reflects the perspective of the person with the problem—the consumer or client: I want a solution that completely solves my specific problem. The second criterion reflects the perspective of the person with the solution—the producer or provider: I have a solution that will help solve everyone’s problems.

Achieving leverage is critical to CBSD in terms of market viability. A component developer must produce a component that simultaneously satisfies enough of the specific needs (‘proportion of solution’) of a sufficient number (‘number of problems’) of individual system composers, to establish a market.

Generalization and the techniques for increasing leverage support CBSD and component reuse. Leverage will increase as CBSD and software engineering in general mature. But how can we increase leverage in the meantime? Within a given problem space, one can go beyond what is common and also capture some of the variability of a set of problems. We might call this predefined variability. A simple example of this is parameterization. Adding a parameter to a component interface increases the variability it can accommodate. Another example is a general model that defines a “component product line” from which multiple component variants can be instantiated. A third example is the interaction model described in Section 3.1, which has been specialized into architecture styles such as

object-oriented, pipe-and-filter, event-based, and blackboard systems [12]. Each style defines component and connector types for a class of systems.

3.4 Specification

This section describes a specification approach that addresses the CBSD shared understanding and integration issues. A specification is a precise shared understanding of an entity or set of entities, as defined earlier. This means that a specification involves an entity such as a model and at least two parties communicating about the model. Typically one party writes the model (e.g., a programmer), and the other party reads the model (e.g., a compiler). The two parties must understand the language used to represent the model. If the two parties share the underlying concepts or semantics of the model, much of the specification can be implicit. If the parties do not share these concepts, more of the specification must be explicit. In a mature discipline, small models are sufficient to represent specifications, because most of the shared information is implicit.

The modeling space approach to specification emphasizes the basic principles of modularity, encapsulation, and precision. A specification consists of a set of rules, where ‘rule’ is used in a very general sense that includes everything from system requirements to code. Examples of types of rules: required data types; required functions; performance properties; provided services; dependencies; policies; types of permitted components in a system; specific components and connectors in a system; attachment of components to connectors (ports to roles); required properties or attribute values; invariants, preconditions, and postconditions; exception handling; state transitions.

An important element of component specification in the modeling space is design by contract, as defined by Meyer [9] but extended to include non-functional (e.g., performance, quality of service/QoS) information.

Specification types are derived from the interaction model and the composition dimension—specifically, the intertwining of internal and external views. An *internal specification* of a composite component or connector entity is a set of rules—including policies—about the data, components, and connectors that are within the composite, and their structure and interaction. An *external specification* of a component or connector entity is a set of rules about the external view of that entity. For a component, that includes observable data, behavior, ports, and services. The relation between the two specification types (shown in Figures 3 and 4) is that an internal specification of a composite includes the external specifications of its components and connectors. The external view corresponds to what we typically call requirements, and the internal view corresponds to what we typically call architecture, design, or implementation.

An external component specification has two contract types. The first is a user specification, which specifies what the component provides to users (services offered). The second is a provider specification, which specifies what it requires of providers (dependencies). Note that this pattern can set up a dependency chain of indefinite length, in which a component can be a provider or server to one component and a user-of or client-to another component.

The interaction model in Figure 3 will now be described further, in terms of informal specification examples. Suppose connector N is a function call, A is a procedure, and B is a square root function. Role 1 is caller, and its associated protocol is as follows. It decides to initiate a call, which involves transferring data and control, and then it waits for a return, which involves receiving data and control. Role 2 is “callee” or server function, and its associated protocol is: It waits for a call, which involves receiving data and control, and then it initiates a return, which involves transferring data and control. The connector N behavior specification is: It receives a call at the caller role and initiates a transfer of this call to the server function role; then it receives a return at the server function role and initiates a transfer of this return to the caller role. Each receipt and transfer of a call or return includes a transfer of data and control. Thus, connector N blocks control at the caller role from the time it receives a call to the time it transfers a return. The Component B behavior specification is: Precondition: Received data $X \geq 0$. Postcondition: Returned data $Y = \sqrt{X}$ within tolerance T and time delta D. B receives control and a data value X. It then returns control and a data value Y.

Most of these details of a function call interaction specification are usually implicit, because function call is a mature connector type and we have a shared understanding of it. However, more details of complex or higher-level interaction specifications need to be explicit to avoid component mismatch.

The specification pattern of relating external specifications to a larger internal specification addresses the CBSD problem of fitting a component into a system. The inclusion of connectors, along with the modeling space approach to component and connector specifications, defines this problem in a precise way and helps determine if a component matches a system or will interoperate with other components. Specifically, suppose that C is an available component, and S is a composite component or system that could potentially use C. That is, the internal specification of S includes an external specification of a needed component we will call SC, and we want to determine if available component C satisfies the needed SC. (In Figure 2, S = FNBS software subsystem and C = customer management component.) The determination is based on a comparison of the external specifications of C and SC. From a contract perspective, we can say that C satisfies SC if and only if these two conditions hold: C provides *at least* all the

services that SC provides, and C requires *at most* all the services that SC requires ($C_{\text{prov}} \geq SC_{\text{prov}} \wedge C_{\text{req}} \leq SC_{\text{req}}$).

To be able to determine this, however, both the system and the component need to be adequately specified. Most current programming languages lack support for this specification approach in the areas of semantics and external specification of required services.

3.5 Representation spectrum

This section describes a representation spectrum that addresses the CBSD semantic gap issue. The artifacts of software engineering have traditional names such as requirements specification, architecture description, design description, and code. In the modeling space, all these are regarded as models of one or more software entities such as system or component. Each model is represented in some notation or language, or combination of languages. The general categories of languages are textual, graphical, and mathematical. The representation spectrum ranges from problem domain models (such as banking or geospatial information) to computer processor models. Corresponding to the language differences are differences in concepts, terms, and domain ontologies. It is really the latter set of differences that establishes the large conceptual gap between problem domains and computer processors, and defines the range of this spectrum. Example: In the banking domain used in the earlier example, key concepts are account, withdraw, deposit, balance, and transfer. In the geospatial domain, key concepts are map, contour, elevation, feature, thematic layer, and projection. In the computer processor domain, key concepts are load, store, add, branch, memory address, and register (actually 01011000, 0101000, etc. but we will use translated terms). The computer processor uses this basic set of concepts to solve problems in banking, geospatial information, and all other problem domains. Note that we listed the concepts in all these domains using English, but the conceptual distance between them remains large.

The relation between models in the representation dimension is translation from one representation to another—for example, problem domain notation to formal specification to UML to Java to machine language. Note that a translation may be combined with relations in other dimensions. In Figure 4, assume that the internal view of A and external view of E are represented in UML, while the next composition level—the internal view of E—is represented in Java. In this example, the respective models of the external and internal views of E have two relations: translation in the representation dimension, and composition in the composition dimension.

The representation spectrum brings into focus several CBSD issues related to language, notation, terminology, and semantics. One issue is sufficiency. Is a specific language sufficient to express the necessary specification

information (described in Section 3.4)? As indicated earlier, most current programming languages are not sufficient in this regard. Further investigation of the use of declarative languages for external specifications may be useful.

Another representation issue is how to determine whether the specification of an available component satisfies the specification of a needed component if the two specifications are in different languages. Do we need to try to adopt a common external specification language—e.g., a formal language, or UML, or IDL, or XML, or natural language? How do we deal with differing ontologies or paradigms, such as procedural versus object-oriented versus functional? An example of recent research is an approach to component search in the context of differing ontologies [4]. The representation spectrum does not resolve the issues, but it does provide a focal point for addressing representation and the semantic gap issue separately from other CBSD issues.

3.6 Mappings

This section briefly describes mappings that capture knowledge about the relations among entities throughout the modeling space. Mappings address the CBSD semantic gap issue. The knowledge to be captured in the modeling space includes not only a large number of reusable models, but also reusable mappings among the models. Mappings are commonly used relations that tie together existing models throughout the modeling space, and help navigate the space when solving a specific problem. Relations include composition, decomposition, generalization, specialization, translation, optimization, and view. For example, suppose we start with a given problem that matches a general model near the problem end of the representation spectrum. A translation mapping might lead us to a model represented as a formal external system specification. A decomposition and translation mapping might lead us to a model representing interacting components of the system in UML. We may then go to our component catalog and match our needs (the specified components of our system) with the specifications of available components, and pick a set that matches. The catalog may exist in the modeling space in the form of external component specifications that provide purchasing or leasing information for associated deployable components.

4 Related work

The generalized view of components and connectors is consistent with software architecture literature, which has promoted connectors as first-class entities [12, 1]. The Real-Time Object-Oriented Modeling approach [11] shares some of these features, and its composition approach is recursive and hence more compatible with the modeling space composition dimension than are most object-oriented treatments.

The connector, as a locus of interaction and coordination, is consistent with literature on coordination models and languages. This literature recognizes coordination as distinct from computation and as a subject of study in its own right [6, 10], and it also addresses the issues of heterogeneous systems. A preliminary taxonomy of connectors is proposed in [8]. Taxonomies and classification schemes are important steps toward reducing artificial variability and accumulating a body of knowledge.

The product-line approach and domain engineering [2, 15] exploit extensive commonality within a homogeneous problem class, which positions both in the generalization dimension. The Kobra approach [3] is an example of the product line approach. Kobra also has other similarities with the modeling space elements presented here. The Kobra framework captures what is common and also captures “concrete variants” (predetermined variability). The dimensions that embody separation of concerns are in partial agreement with the modeling space dimensions. The primary differences between the two approaches are (1) greater emphasis on interaction and connectors in the modeling space, and (2) the modeling space representation dimension as opposed to the development process emphasis in Kobra.

Szyperski’s approach to component specification [14] is consistent with the approach in this paper. Szyperski also discusses the specific “wiring standards” defined in three primary approaches to component software: CORBA, JavaBeans, and Microsoft’s COM/DCOM. However, his treatment does not provide a general approach to connectors or interaction. The new CORBA component model (CCM), described in [13], is a generalized and extended form of Enterprise JavaBeans or Java 2 Enterprise Edition. The CCM is consistent with a number of elements in the modeling space, including specification contracts and modularity. The concept of container has some of the mediation features of a connector, but is more specialized for the CCM environment. CCM appears to be focused on the programming region of the representation spectrum rather than the full spectrum.

Catalysis [5] is an approach to objects, components, and frameworks that emphasizes connectors as well as components and covers a significant part of the modeling space. Catalysis uses the concept of *object* as the locus of static functionality and data, and *action* as the locus of dynamic activity. It supports composition of both objects and actions. Generalization is supported via model frameworks.

RM-ODP [7], an ISO standard for distributed processing systems, has a number of similarities with the modeling space. Many of the foundation concepts, such as encapsulation, interface, and contract, are compatible. The RM-ODP architecture concepts include a list of distribution transparencies, which maps to the

generalization dimension. RM-ODP presents five viewpoints of a distributed system: enterprise, information, computational, engineering, and technology. The information view maps to data specification in the modeling space. The other four viewpoints all map to some degree to an internal component specification, at different levels of composition and generalization.

What the modeling space adds to this related work is a broad context in which these various approaches can be positioned and compared. The modeling space also adds a structure for compiling and organizing models that describe components, their interactions, and the larger configurations into which they can be integrated.

5 Conclusion

Benefits. The modeling space described in this paper supports CBSD and component modeling, both in the near term and the long term. In the near term, it provides a uniform structure for modeling components and modeling systems in which the components may be integrated. Modeling the systems supports the system composers. Modeling the components supports both the component developers and the system composers.

In the long term, the uniform structure can serve as the basis for an organized repository of knowledge of components and systems in which they can be used. This knowledge will be in the form of a large number of well-understood models that will exist throughout all dimensions of the modeling space, and relations or mappings among the models.

In addition, the modeling space elements apply to software engineering in general, not just to CBSD. Many large systems require a combination of the component paradigm and other approaches such as custom development. The modeling space defined in this paper can reconcile these approaches.

Validation. The modeling space approach described in this paper has not yet been directly validated in CBSD practice. However, the approach represents a consolidation of elements with a solid foundation in software and systems engineering practice. Conceptualizing software engineering as modeling is fairly well established. Generalization and composition are well established in software engineering and also have a long tradition in other disciplines such as ontology, biology, and mathematics. Composition and representation have long been the primary elements of the software life cycle. Thus the argument for the validity of the modeling space at this point is based on the pedigree of its elements. Further work in direct CBSD validation is anticipated.

Acknowledgments

The MITRE Corporation provided support for the research reported in this paper.

6 References

- [1] Allen R. & Garlan D. (1997) A Formal Basis for Architectural Connection. *ACM Trans. on Software Engineering & Methodology*, 6, 3, p. 213-249.
- [2] Arango G. & Prieto-Diaz R. (1991) *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press.
- [3] Atkinson C., Bayer J., Laitenberger O. & Zettel J. (2000) Component-Based Software Engineering: The KobrA Approach. 2000 Int. Workshop on CBSE, Limerick, Ireland. Available at <http://www.sei.cmu.edu/cbs/cbse2000/papers/21/21.html>
- [4] Braga R., Mattoso M. & Werner C. (2001) The Use of Mediation and Ontology Technologies for Software Component Information Retrieval. 2001 Symposium on Software Reusability, Toronto, Canada, May 18-20, 2001. Published in *Software Engineering Notes*, 26, 3, p. 19-28.
- [5] D'Souza D. & Wills A. (1998) *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley.
- [6] Gelernter D. & Carriero N. (1992) Coordination languages and their significance. *Communications of the ACM*, 35, 2, p. 97-107.
- [7] International Organization for Standardization (1995) *Basic Reference Model of Open Distributed Processing*. ITU-T Recommendation X.902 | ISO/IEC 10746-2: *Foundations* and ITU-T Recommendation X.903 | ISO/IEC 10746-3: *Architecture*.
- [8] Mehta N., Medvidovic N. & Phadke S. (2000) Towards a taxonomy of software connectors. *Proceedings of the 22nd Int. Conference on Software Engineering*, Limerick, Ireland, p. 178-187.
- [9] Meyer B. (1997) *Object-Oriented Software Construction* (2nd ed.) Prentice Hall.
- [10] Papadopoulos G. & Arbab F. (1998) Coordination Models and Languages. CWI Report SEN-R9834. Available at: <http://citeseer.nj.nec.com/papadopoulos98coordination.html>
- [11] Selic B., Gullekson G. & Ward P. (1994) *Real-Time Object-Oriented Modeling*. John Wiley.
- [12] Shaw M. & Garlan D. (1996) *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall.
- [13] Siegel J. (2001) *Quick CORBA 3*. John Wiley.
- [14] Szyperski C. (1998) *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley.
- [15] Weiss D. & Lai C. T. R. (1999) *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley.