

A Formal Ontology for Re-Use of Architecture-Level Software Documents

Christopher Welty* and David Ferrucci

IBM Watson Research Center
Yorktown Heights, NY 10598
{welty,ferrucci}@us.ibm.com

Software Architecture has been established as a viable level of representation for re-use in practical software engineering efforts. The main reason for this is that an architectural view of software is sufficiently abstract to have many instantiations. Even with technologies such as CORBA and JavaBeans, which emphasize re-use of components, the realization of widespread re-use has been severely limited. While architectural re-use has been successful, it has thus far suffered from an ad-hoc semantics, and even savvy architecture practitioners are unsure precisely what is being re-used. We have been engaged in research into re-use of software *documents*, such as design documents, statements of work, contracts, etc., that capture and re-use architectural level knowledge of software solutions. We have found that, given a sufficiently robust knowledge-based tool for maintaining documents, a formal ontology or meta-model for software architectures is required to achieve re-use of these architecture-level documents. We present such an ontology here.

Introduction

It is entirely unclear how much software reuse has actually been realized over the past twenty years. While software synthesis has succeeded on a small scale, it has thus far been limited to domains that are inherently mathematical, such as seismic modeling (Baxter and Kant, 1991), scheduling (Smith, 1991), and spacecraft simulation (Lowry, et al. 1994). Larger scale projects such as the C++ standard template library (Musser, et al., 1996) have certainly been responsible for widespread reuse of generic algorithms and data structures, however the cost or time savings here is minor, since any programmer advanced enough to plod through the syntax of STL could write a container class or sort program in their sleep anyway.

The software architecture community has been touting the advantages of a higher level approach to reuse for the past few years (Shaw and Garlan, 1996). Where synthesis systems require a complete, formal (and correct) axiomatization of a domain in order to realize reuse (Welty, 1998), architectures can be reused fairly easily. The architecture exists at such an abstract level that most details have been removed. The connection to software is sometimes unclear, and precisely what gets reused is also quite vague: it may be pictures, documents, parts of documents, software components, parts of software components, off-the-shelf software packages, etc.

In all cases, from synthesis to architecture, it is clear that knowledge does get reused. The difference is the amount to which that reuse can be reflected in resource savings, and the degree to which the knowledge can be transferred to others. One would assume that in a formal system the knowledge is in a highly reusable form, and that in an informal system it is less so.

While architecture has been sweeping the commercial software world, along with enabling technologies such as CORBA (Ben-Natan, 1995), *knowledge management* has become the current buzzword at the executive level, and most large corporations even have V.P.s assigned to this rather vague and unspecified task.

We have been studying the problem of capturing and reusing architectural level knowledge used by software groups within IBM. We chose the architectural level for the very reason that it is abstract

* Also at Vassar College, Poughkeepsie, NY. Contact email: weltyc@cs.vassar.edu.

enough to imagine real knowledge reuse working from project to project. A complete axiomatization of a domain would be finished and ready to generate software several years after the software would be needed, and even then the ability to reuse the domain model for another project would be severely limited, because the knowledge is so specific.

At the architectural level, this is far less true. We concentrated on formally representing architectures in such a way that the knowledge could be captured and formally reused in *documents*. While generating software is the ultimate goal of any software group, many documents also get generated. In fact, these documents (such as statements of work, proposals, contracts, etc.) typically go through far more revisions than the software itself during their (comparatively brief) lifetimes.

Architecture Document Re-Use

To begin with, we felt reuse through documents had a better chance of succeeding because documents are interpreted by people, not by computers. As a result, slight variations in terminology can be easily resolved. Even simple issues like using synonyms to specify the same concept are not easily resolved by computers, but do not present problems to people. Architectures, again, are at a sufficiently high level of abstraction that a lot of different software systems can be created that still conform to the architecture.

Architectural knowledge is difficult to nail down formally, however, and opinions vary on what exactly makes a good architecture (Bass, et al., 1998). Some important things for architectures to get correct are what the major functional components in the system are, who will be responsible for them, and how they will interact. The assignment of resources at this stage is also central.

In reusing architectural knowledge, in our experience the things that change at this level are the specifics of *how the components connect* to each other, however we found this to be a weak point in many architecture specifications. To reuse the architecture effectively, a thorough understanding of the various ways the components can be connected is required. In addition to being able to represent an architecture in general, this is the very knowledge we intend to capture and reuse.

For example, a critical enabling technology that has pushed software architectures forward has been middleware such as Object Request Brokers (ORBs, see Ben-Natan, 1995). ORBs provide precisely the kind of support that large software system designers need to make distributed development (across multiple teams, perhaps across multiple companies) and distributed execution possible in a fairly transparent way. It is typically the case in such distributed development that individual teams understand their own components quite well, and the real effort at the architectural level is understanding how the components integrate. This makes the connections between the components, i.e. the interfaces and the middleware, a central focus of the architecture. Furthermore, when any of the components to be integrated are legacy systems (which is usually the case), the connections become the primary part of the effort from architecture and system design straight through to development and maintenance.

We have focused our attention on representing architectures in way that fosters reuse of knowledge about integrating, and therefore connecting, different types of components.

Software Architecture Meta-Model

Meta-models for software architectures abound. In our research we have found that the high level of abstraction of most architectures has led to a very weak, and at times ambiguous, semantics for these models. There have been attempts at formalizing the semantics further (see for example Penix, et al., 1997), however none stressed connections between components to the degree we required. Most meta-models realize Architecture Description Languages (ADLs) which have an intuitive interpretation that makes sense to designers who have experience with the language (Kirova, et al., 1997). Most software architecture *specifications* have a similar informal interpretation, and only make sense to people with experience in the domain of the architecture. (Bass, et al, 1998)

This lack of semantics is not particularly significant given the high level at which architectures reside, and given that the only real purpose of any architecture is to provide a reference for designers and implementers (Kirova, et al., 1998). It does become a problem when any kind of reasoning is applied to a specification. Our goal was specifically to tie an architectural description to a reasoning engine for the purpose of maintaining a document, so a more complete semantics than that we found in the literature was required.

We should note, however, that while we did seek to formalize the semantics of architectural models for the purposes of automated reasoning, we stopped well short of a complete specification. The vague and potentially ambiguous semantics of certain architecture concepts are actually their strength in many ways. A complete formal semantics seemed an unreasonable goal considering that we only wished to generate documents. These documents, again, would be interpreted by people who would understand the subtlety of the language or notation used. This is precisely why architectural description has been viewed as successful. We provide several examples throughout the paper of where terms were left ambiguous.

Standard Definitions

Most ADLs provide two basic modeling primitives: *components* and *connectors*. Components define some functional part of the software system (such as “user-interface”); generally speaking they are the modules or packages in the implementation. Connectors provide communication between components; this may be as simple as passing parameters with normal function invocation, or as complex as an object request broker (ORB).

Given the possibility for a connector to be a piece of software, the first question we had when examining the architectural approach was whether a connector is a component or not. In some respects, the two are entirely different; where a component contributes something to the overall software functionality, connectors are normally considered “middleware,” and perform no externally manifested task. They have numerous similarities as well; complex connectors such as ORBs do provide *location transparency*, which allows components to be seamlessly distributed across a network. This service can be viewed as a “feature” of the overall software system, which is something we intuitively attribute to a component. In addition, in large software systems, particularly those that involve the integration of legacy software components, it is normal to assign a team to implement the connectors. The association of personnel to a part of a software system is usually sufficient for considering that part a component. Finally, complex connectors such as ORBs occasionally need to be connected (through other middleware) to components themselves.

Meta-Model Additions

In our work, we found the notion of an *interface* to be fundamental, yet not supported in most architecture meta-models. Software interfaces are the most essential ingredient for component-based design. CORBA, JavaBeans, and all distributed object support systems rely on clearly defined interfaces to enable the integration they provide. Most architecture models view the interface as a part of each component, at a level below that which is considered architectural.

New technologies such as CORBA and Java, however, stress the notion that a single interface can be implemented by a variety of components. These technologies intentionally decouple interfaces from their implementations, and make both important to be able to represent and manipulate independently. Note that we do not wish to claim that the notion of decoupling an interface from its implementation was introduced by Java, however Java and CORBA in particular have made the implications of this separate treatment more apparent to a wider range of developers.

Most descriptions of distributed systems also refer vaguely to the notion of *services* that interfaces publish or provide, and that components implement. It is difficult to determine whether a service corresponds precisely to a method (i.e. member function) defined in an interface, or whether it is something more abstract. For example, a database component may provide a “query” service, which is a method listed in

the component's interface. The same database component may provide "object persistence," which is a service that does not correspond to any particular method.

From an architectural standpoint, or at least from the standpoint of our project goals, the specifics of how a service is manifested in software was not important. Our goal was not to generate software but to generate documents, and this is an example of where ambiguity in a term was intentionally left in our model. There was no doubt that the people reading the documents would be able to understand how "persistence" and "query" services map to implementation.

Patterns

In addition to components and connectors, many ADLs employ standard software patterns, either as part of the meta-model, or more commonly as templates that can be instantiated for a particular architecture (Mowbray and Malveau, 1997). The notion of design patterns fits well into our methodology, although it suffers far more than architectures from loose semantics and ambiguity of terms.

Several efforts to standardize design pattern terminology have been made, but as with CASE tools, most practitioners choose a particular approach and it's accompanying terminology. We found this lack of cohesion a major barrier in understanding the subtleties of certain patterns that were relevant to our work. Even worse, these terms caused confusion in communicating between group members, and near disaster communicating with external members. One particular pattern, the "Bridge", repeatedly set the project back. These problems were quite disturbing, and forced us to re-think our belief that, since we were dealing with people a little ambiguity was acceptable.

Examining the sources of these problems led us to two conclusions:

Many design patterns are too closely tied to implementations to be useful at the architectural level. For example, most pattern books list "iterator" as a standard design pattern, yet all C++ programmers have a very clear idea of what an iterator is. The C++ idea of an iterator does not subsume all iterator definitions, in fact an iterator design pattern is merely something that provides for iteration over a container, it doesn't need to behave like a pointer (as it does in C++).

Many of these pattern names are heavily overloaded. This is probably a consequence of the previous point. Many people are familiar with only one implementation of a pattern and find it difficult to abstract away the details. When different people who have experience with different versions of a pattern get together, confusion reigns.

These points bring about an interesting analogy to physical architecture. Architects are very rarely involved in actual construction, and typically are not conversant in the vernacular of construction workers, and the reverse is true of construction workers. Many of the problems we had in describing these terms came from the fact that all our software architects were also experienced programmers.

Rather than remove programmers from the room during meetings, we found terms, in particular design pattern names, that caused confusion needed to be crisply defined, and in these cases our ontology (discussed below) was extremely helpful. For example, although we did end up using the term "bridge," we defined it as "a connector that provides location transparency." Our definition only matched a subset of the definitions of Bridge we encountered, but it was all we needed for the purposes of our modeling efforts. The ability to provide such a concise definition for the term made it easy to wave our hands at all questions and say, "it doesn't matter what that book says, in this system a bridge provides location transparency and that's it."

Formal Ontology

Ontology literally means "the study of the state of being." In a formal sense, it is a description of *what things are* in a particular model. Formal Ontologies are too often associated simply with taxonomies of classes, but in more expressive formal systems they also include general structural descriptions, constraints, and axioms.

A meta-model is a type of formal ontology. We choose to employ the term formal ontology here, however, because its association with knowledge representation and reasoning helps differentiate our efforts from the more traditional ADLs which, to our understanding, are not usually intended to support automated reasoning.

Basic Elements

Our ontology identifies three basic types of architectural elements: *component*, *interface*, and *service*. Based on our observations, we chose to make *connector* a type of *component*, and differentiate *functional-component* from *connector* below the *component* concept. These types are shown in Figure 1. We further formalize the differences between a functional component and a connector below, however there is some ambiguity left, and it is up to the architects to determine when a component is a functional component and when it is a connector.

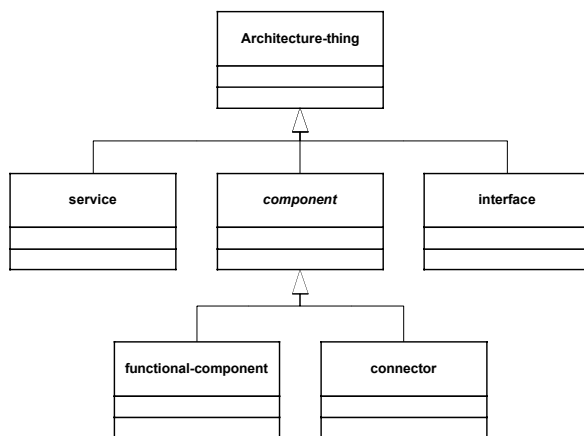


Figure 1 The taxonomy of architectural elements.

The primary focus we have, as mentioned above, is on reusing knowledge about integrating functional components. The notions of services, interfaces, and connectors are therefore central to our ontology.

In addition to the architectural elements, we also define *group* and *project* as types of objects that we will need to associate with particular instances of the architectures we create. An instance of *project* will represent a single instantiation of an architecture. In other words, each time an architecture is used, there will be a single project object associated with it. The project object is needed by our document generation system to provide a single point of reference into any architectural model.

We take the position that formal ontology should not be done without specific goals in mind. Our ontology goals are to develop a basic ontology for representing architectures, equivalent to ADLs with the addition of automated reasoning. Architects then specify architectures using the elements of our formal ontology. Completed architectural models are then *instantiated* for individual projects. This instantiation is characterized primarily by the creation of several documents that describe the architecture and specify how it will be used to solve the particular problem the project is concerned with.

Relations

In an architecture, instances of the three basic elements (component, interface, and service) are specified and related to each other through various structural relations. The UML class diagram for the architectural elements is shown in Figure 2. Note that while many of these diagrams use UML notation, we are not actually using UML to represent our formal ontology. We are using a Prolog-like language that provides simple relations, attributes, and inference. The subset of UML notation that we use here is sufficient for diagrammatic representation of our ontology, but we found the lack of semantics in UML to be far too crippling to use it as a true modeling language.

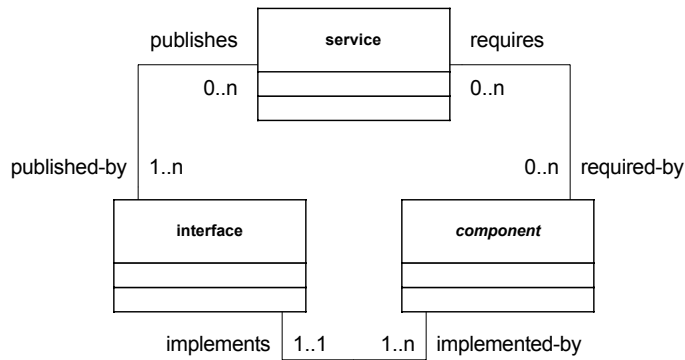


Figure 2 Relationships among architectural elements.

A component *implements* one and only one interface. A component also *requires* any number of services.

An interface *publishes* any number of services. A zero cardinality is permitted here because, at the architectural level, the services published by some interfaces may not be relevant. An interface can be *implemented-by* (inverse of *implements*) one or more components.

A service may be *published-by* (inverse of *publishes*) one or more interfaces, and may be *required-by* (inverse of *requires*) any number of components.

Normally, only the relations *publishes*, *requires*, and *implements* are represented explicitly. The inverse relations are propagated automatically by the reasoning system.

In addition to these basic relations for the architectural elements, we also represent relations for the group and project concepts, as shown in Figure 3.

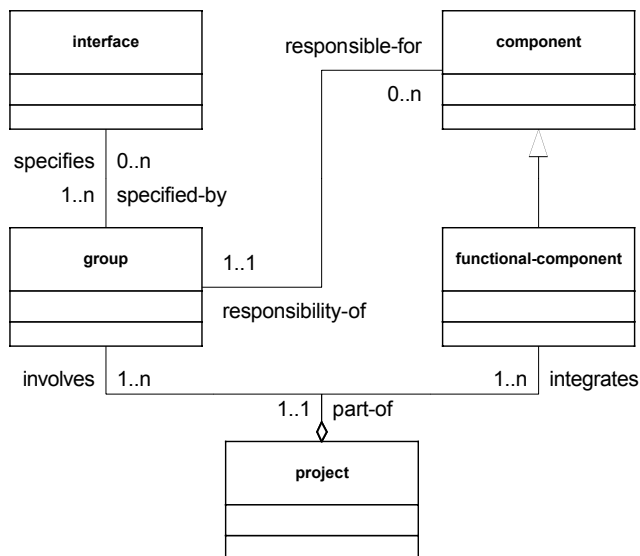


Figure 3 Relationships for group and project objects.

A project is an aggregate object that *involves* one or more groups, and *integrates* one or more functional-components. The *integrates* and *involve* relations are mereological (i.e. a form of *has-parts*). Note that at the level of a project, we do not need to directly specify the connector components that will be used. These will be accessible through relations the system provides, and furthermore, as described later, are also automatically derived. Note that our underlying system does not deal with more than one project at a time.

A *group* encapsulates a group of people, and could be a company, a division, or just a collection of people assigned to a specific task. The group is *responsible-for* implementing any number of components, and *specifies* any number of interfaces. Note that multiple groups can specify an interface, but only one group can be responsible for a component. Given that division of labor and allocation of resources is a primary purpose of architectural specification, this simple representational detail shows an important feature of separating the notion of an interface from a component. Interfaces are frequently devised through collaboration between multiple groups, however components are typically implemented by a single group.

Reasoning

The most important aspect of the formal ontology is the axiomatization of the domain of possible connectors. For various intellectual property reasons, we are not at liberty to share the specifics of this axiomatization, because it represents the very knowledge we seek to re-use. We include some of the general framework from our ontology here.

To begin with, we add another relation between components and interfaces: *calls*. A component *calls* an interface that publishes a service the component requires. Our choice of representation may seem *prima facie* unnatural, because the *calls* relationship is not explicit. Most people think of architecture specification as a process of specifically connecting components. For example, a typical and familiar architecture for a bank ATM system is shown in Figure 4. This architecture specifies that there are two components, and that they need to be connected.

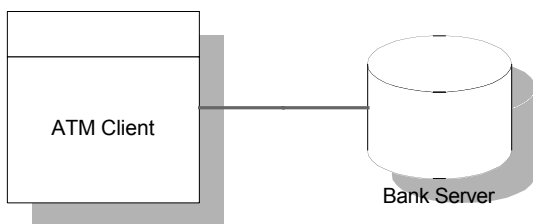


Figure 4 A simple architecture.

Of course there is a lot this architecture does not specify, and in many respects it is a terrible architectural specification, but our point is to define some of the differences with our approach, not to exemplify good architectures. Rather than specifically represent the fact that the *ATM Client* component is connected to the *Bank Server* component, we prefer to represent that the *ATM Client* requires a service that is provided by the *Bank Server*. Furthermore, as we discussed above, the notion of an interface is critical to understanding how a system of components integrate. We have chosen to represent the fact that a component “presents an interface” to other components, and this interface “publishes services.” The result, for the same architecture, would be as shown in Figure 5. The dotted line indicates a relationship that is (or can be) derived. The *published-by* relation appears simply as the inverse of *publishes*.

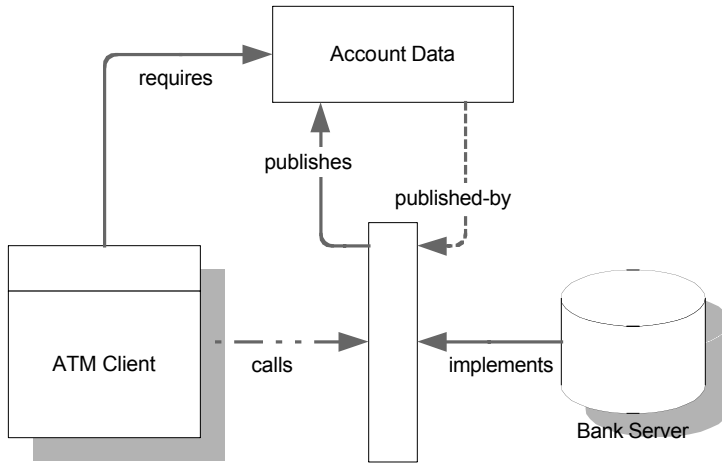


Figure 5 The same architecture using our ontology.

The *calls* relationship, displayed as a mixed dash and dotted line, is not explicitly derived. In our modeling language, we differentiate between axioms that provide forward reasoning (such as relation inverses and transitivity), and axioms that specify constraints. Where forward reasoning simply derives values automatically, constraints are processed into sets of possible values, which are presented to the user. The user then selects from these to set a value.

For the *calls* relation, we have the constraint:

$$\forall c \in \text{component } s \in \text{service } i \in \text{interface} \\ \text{calls}(c,i) \text{ IFF } \text{requires}(c,s) \wedge \text{published-by}(s,i)$$

In other words, a component calls an interface that publishes a service the component requires.

Our system also collects all constraints on a particular relation and processes them together. Multiple constraints are therefore conjunctive, and the intersection of the values that satisfy all the constraints are what the user must choose from. This is important because the *calls* relationship has several constraints on it; in fact, as will become clear below, this relationship lies at the core of our technique, since we have focused on reusing knowledge of how to connect the various components in an architecture. We present only one of the simpler constraints here, to give the flavor of the kinds of knowledge we have captured.

We have developed a taxonomy of general technologies, such as C++, Java, CORBA, RPC, OLE, etc., and specific instances of these technologies such as ORBIX, Persistence, etc. Each instance of *group* in a model has associated with it a set of *skills-possessed*, indicating the technologies (general and specific) the group has experience with. Each interface has associated with it the technologies required to call it. We define the constraint:

$$\forall c \in \text{component } i \in \text{interface } g \in \text{group} \\ \text{calls}(c,i) \text{ IFF } \text{responsibility-of}(c,g) \wedge \text{skills-required}(i) \subseteq \text{skills-possessed}(g)$$

In other words, a component can only call an interface if the group responsible for it possesses the skills the interface requires. This constraint may *prima facie* seem overly restrictive; it is certainly possible for a group to acquire new skills. The point here, and it is a crucial one, is that it makes the need to acquire these skills apparent at the architectural stage. If an architect wants to connect one component to another and finds that he is prevented from doing so, our system has an explanation facility that will divulge the reasons. The knowledge that a group will need to acquire new skills impacts the design in numerous ways:

1. It may increase the cost. An inexperienced group will take time to learn a technology, and once learned will not be able to develop their software as quickly as an experienced group.

2. It increases the risk. An inexperienced group may not know all the problems a technology has.
3. It may require a new set of connections to be used. It is possible, for example, to put layers of middleware between two components that map from one technology to another, and allow a component developer to connect to the system without acquiring new skills. This obviously requires that another group be responsible for the new middleware components.

This third point exposes another interesting issue in our ontology: the treatment of connector components.

Connectors

The final important idea in the formal ontology to discuss is how we have represented connectors. Connector details are not normally explicit at the architectural level. We hope we have made the point that in our experience we found this to be a limitation; the details of connecting the components are quite important at this stage. In order to facilitate using our approach, we tried to make specifying the details of connectors as seamless as possible. In particular, we developed (as part of the taxonomy of technologies) a taxonomy of connector technologies and their appropriate attributes.

These connectors are integrated into an architectural model (of functional components, interfaces, and services) in two ways:

1. Many connectors provide some service in addition to providing a connection. For example, a CORBA connector provides location transparency. Some middleware provide object persistence. These services are represented in the model as required by various components.
2. Connectors *project the services* of the component they connect to, to the component they connect from.

An example of the ATM architecture expanded to include knowledge of connectors is shown in Figure 6.

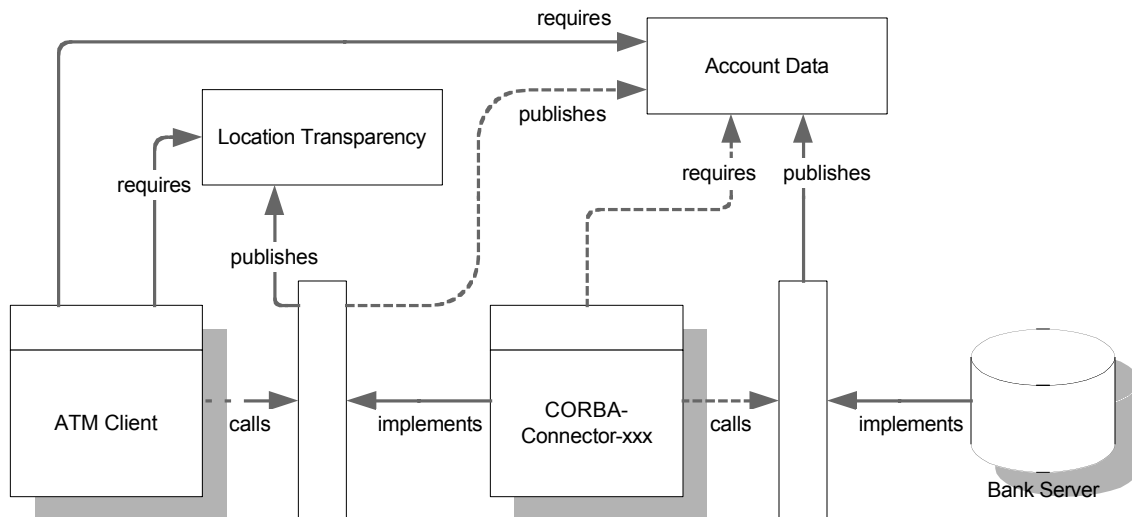


Figure 6 ATM architecture with connectors

In this example, as previously, the dotted lines indicate derived relationships. Our connectors find through inference the places where they “fit.” A connector derives the *publishes* relation from the interface it *calls*:

$$\forall i1, i2 \in \text{interface } s \in \text{service } c \in \text{connector} \\ \text{publishes}(i1, s) \leftarrow \text{implemented-by}(i1, c) \wedge \text{calls}(c, i2) \wedge \text{publishes}(i2, s)$$

A connector also derives the *requires* relation from the component it is *called-by*:

$$\forall c1 \in \text{connector } s \in \text{service } i \in \text{interface } c2 \in \text{component} \\ \text{requires}(c1, s) \leftarrow \text{implements}(c1, i) \wedge \text{called-by}(i, c2) \wedge \text{requires}(c2, s)$$

Note that both of these rules allow the automatic propagation of information so that the constraints on the *calls* relation hold. Note also that these rules allow for “chains” of connectors to be between two functional components, with all the information propagated along the chain. One of the main tasks in the automated reasoning we have defined for our document configuration system is a search for the *minimal* set of connectors that will satisfy all the constraints.

Architectural Documents

The main purpose of all this work is to enable reuse of architectural knowledge through documents. For each architecture an organization wishes to reuse, a model is created using the ontology described in the previous section. In addition, for each architecture a set of document models are created. For each project in which the architecture is used, the architecture and document models are instantiated, and specific knowledge about the project is added.

A single project will typically have multiple documents associated with it. Document types include proposals, statements of work, legal contracts, etc. These documents will be generated repeatedly as new information is gained, yet the basic structure of most will remain unchanged. In addition, from project to project a large part of the documents will also be the same. The purpose of the architectural and document models will be to identify the things that will be the same each time, and the things that may change. These “variants” are then analyzed, and as much as can be modeled about the possibilities are represented. The process of instantiating an architecture, then, is simply filling in values for all the variants.

Our document configuration system should not be confused with a simple template system. The document models are specified using the same sorts of axiomatizations we described for the architecture ontology. This allows the documents to maintain their internal consistency across their lifetime. In fact, the majority of the knowledge in the documents is maintained in the knowledge representation system, not in the documents themselves. The axioms governing the architecture and the document are therefore always kept consistent.

Conclusion

We have presented a formal ontology for representing software systems at an architectural level. This ontology allows us to represent an architecture from the perspective of the middleware: the connections between components. We found this to be an important aspect to focus on because it is the place where the most potential variation, and thus the most potential misunderstanding, takes place.

Our ontology facilitates automated reasoning for determining good choices for connectors. These choices are based on information such as the skills possessed by the groups, the interoperability of certain software with others, technologies used, etc.

The result of modeling an architecture and instantiating it for a project is the generation and automatic maintenance of documents that describe important aspects of the architecture, such as work assignments, levels of risk, cost, etc. These documents are configured automatically by a proprietary system developed jointly by IBM and Legal Knowledge Systems (LKS Inc.). The important feature of this tool for our purposes is that changes in the architecture are made *to the knowledge* not the document. New versions of the document are generated from the updated model.

References

- Baxter, Ira, and Elaine Kant. 1991. Domain Modeling in SINAPSE for Synthesizing Mathematical Modeling Programs. *Proceedings of the ICSE 1991 Domain Modeling Workshop*. Austin Laboratory for Software Engineering and Computer Science. Pp. 23-25. May, 1991.
- Bass, Len, Paul Clements and Rick Kazman. 1998. *Software Architecture in Practice*. Addison-Wesley.
- Ben-Natan, Ron. 1995. *CORBA: A Guide to Common Object Request Broker Architecture*. McGraw-Hill.
- Kirova V., L. Jololian, H. Lawson, and T. Zemel. 1997. A Generic Model for Software Architectures. *IEEE Software*. 14(4). July/August 1997, IEEE Computer Society Press, Los Alamitos CA, pp. 84-92.

- Kirova, V., Howard Kradjel, and Willhelm Rossak. 1998. DirSA Case Study: An Introduction to Software Architecture Technology. *Bell Labs Technical Journal*. 3(3). September 1998, pp.125-139.
- Lowry, M., A. Philpot, T. Pressburger and I. Underwood.1994. A Formal Approach to Domain-Oriented Software Design Environments. *Proceedings of the Ninth Knowledge-Based Software Engineering Conference*. IEEE Computer Society Press.
- Mowbray, Thomas and Raphael Malveau. 1997. *CORBA Design Patterns*. J. Wiley and Sons.
- Musser, David R., Atul Saini, and Alexander Stepanov. 1996. *STL Tutorial & Reference Guide : C++ Programming With the Standard Template Library*. Addison-Wesley.
- Penix, John, Perry Alexander, and Klaus Havelund. 1997. Declarative Specification of Software Architectures. In Lowry and Ledru, eds., *Proceedings of ASE-97: The Twelfth Automated Software Engineering Conference*. IEEE Computer Society Press, Lake Tahoe, Nevada.
- Shaw, Mary, and David Garlan. 1996. *Software Architecture : Perspectives on an Emerging Discipline*. Prentice Hall.
- Smith, Douglas. 1991. KIDS: A knowledge-based software development system. In *Automating Software Design*, M. Lowry and R. McCartney, Eds. MIT Press, Menlo Park.
- Welty, Chris. 1998. Desert Island. *J. Automated Software Engineering*. 5(4). Oct. 1998. Pp. 465-469. Kluwer.