A Paradigm for Component-Based Software Development in a Distributed Environment

Ayesha Mascarenhas
Dept. of Computer and Information Science
The Ohio State University
Columbus, Ohio, USA
Tel Ph:614-292-8234
mascaren@cis.ohio-state.edu

Paolo A.G. Sivilotti
Dept. of Computer and Information Science
The Ohio State University
Columbus, Ohio, USA
Tel Ph:614-292-5835
paolo@cis.ohio-state.edu

ABSTRACT

The component-based construction of complex software systems requires reasoning about the behavior of a system based on the behaviors of the individual components. Existential and universal properties are examples of component properties that enjoy particularly simple and elegant theories under composition. In this paper, we focus on the practical issues involved in the development of distributed systems from components exhibiting existential and universal properties. We explore how this compositional paradigm can be applied in current industrial distributed component frameworks, in particular CORBA and .NET.

Keywords

Distributed Components, Specification, CORBA, .NET

1. INTRODUCTION

Compositional development allows us to construct large software systems from smaller, simpler components. This is exciting from a software development standpoint because it promises the reusability of developed components[10, 6]. Many techniques that promote component reuse focus on this particular aspect of reuse—that is, on *implementation reuse*. Simple inheritance in object-oriented programming languages is an example of this kind of reuse. In addition to implementation reuse, however, a robust compositional methodology also requires *reasoning reuse*.

When a component is built, the developers spend time and effort convincing themselves that the implementation does what it is supposed to do. This process may involve combinations of testing, code walk-throughs, the adoption of various coding standards, and some degree of abstract reasoning, either formally or informally. Once the component is deployed, clients of this implementation should not have to start from scratch in reasoning about the aggregate behavior of this component in the larger system. This requires that component descriptions be complete and precise. Mechanisms for achieving this precision, as well as the proper balance between completeness and abstraction, have been the topics of considerable research in software engineering.

Completeness and precision, however, are not sufficient for enabling reasoning reuse. The properties used in the description of component behavior must also lend themselves nicely to composition. That is, care must be given to describe components in such a way that compositional reasoning is possible. Recently, a new taxonomy of compositional properties has been introduced [3, 2]. This taxonomy identifies two fundamental classes of properties, universal and existential properties, that enjoy particularly simple compositional theories. Existential properties are those that hold on a system whenever the system contains at least one component that has that property. Universal properties are those that hold on a system whenever all the system components have that property.

As an example of an existential property, consider a distributed component that broadcasts streaming video, generating at least 1Mb of continuous network traffic. Regardless of how this component is used in a larger system, that larger system will also generate at least 1Mb of continuous network traffic. As an example of a universal property, consider a system that uses token-passing to manage access to a shared critical resource. If every component guarantees it neither creates nor destroys tokens, the number of tokens in the system as a whole remains constant.

Not all properties of interest, however, can be expressed directly as a combination of existential and universal properties. To address this concern, *property transformers* based on weakest/strongest existential/universal properties can be used to transform properties that do not exhibit this nice compositional flavor into properties that do. For example, a strongest existential transformer for some property P, gives the strongest property that is weaker than P and that is also existential. If we know these strongest existential properties that a component exhibits, we also know something about any system that contains this component, and we can reason about the system as a whole, based on the properties of that one component.

In this paper, we examine the practical implications of these compositional properties on software development in a distributed environment. We describe how a bottom-up sys-

PDPTA 2002, Las Vegas, Nevada USA

tem integration strategy based on existential properties and on property transformers can be realized in two industrial-strength distributed component technologies: CORBA [12] and .NET [14].

2. ILLUSTRATIVE EXAMPLE

Consider the design and construction of a distributed auctioning system. One basic system property is that once the auction for an item has started, the price of the item never decreases, unless the auction is cancelled. Informally, the auction not being cancelled implies the item price never decreases. One way to achieve this system property is to identify and use a component that maintains a local integer and increases this integer based on received bids. Notice that the desired property is a *system* property, not a component property. What is needed, therefore, is a system-level guarantee that the current auction price does not decrease. This system-level guarantee follows from the observation that the monotonicity of a local variable can be phrased as an existential property. That is, the counter component would guarantee that it does not decrease its count unless the environment calls its `Clear()` operation.

Now this same component might be reused in another system to count, say, the number of visitors to a web site. The existential property of the counter guarantees the same monotonic behavior in both systems. Notice that an existential property must hold *regardless of the environment in which the component is placed*. While this requirement might seem to be a significant restriction, in practice requirements on the environment can be incorporated into the property itself. In the counter example, the existential property would, informally, be "if the `Clear()` method is never called, the count is non-decreasing". In this way, existential properties can be both highly expressive (capturing behavior that is conditional on the environment) and easily composed (capturing behavior that is unilaterally guaranteed by the component, regardless of environment).

3. BASIS FOR THE PARADIGM

Any paradigm for the compositional development of distributed systems must address certain fundamental issues. In this section, we examine each issue in turn, with particular attention to how the use of existential and universal properties influences design decisions for that issue.

Any compositional development paradigm will require:

- A method for describing component behaviors in terms of compositional properties.
- A method for publishing and searching component specifications and sharing component implementations.
- Tool support for high-confidence component integration.

3.1 Component Descriptions

In most component-based enterprise frameworks, such as CORBA and Enterprise Java Beans, a component description consists simply of the component's syntactic interface; that is, a list of method signatures implemented by the component. The inadequacies of this approach are well known and there have been several attempts to extend such interface descriptions with formal behavioral descriptions [16, 7].

Interface
Inc();
Clear();
Abstract State
integer: count
Safety
initially.$(count = 0)$
$(\forall j,k :: \#Inc = j \ \wedge \ count = l$
next $\#Inc = j+1 \ \Rightarrow \ count = l+1$ $)$
$(\forall j,k :: \#Clear = j$
next $\#Clear = j+1 \ \Rightarrow \ count = 0$ $)$
$(\forall j,k,l :: \#Inc = j \ \wedge \ \#Clear = k \ \wedge \ count = l$
next $(\#Inc = j \ \wedge \ \#Clear = k) \ \Rightarrow \ count = l$ $)$
Existential
invariant.$(\#Clear = 0)$ **guarantees**
$(\forall k :: ((count = k)$ **next** $(count \geq k))$ $)$
Informal Specification:
Inc() increases count by some positive amount
Clear() sets count to 0

Figure 1: Description of Counter Component

Formal specifications can be expressed in terms of pre and post conditions following a "design-by-contract" approach [11]. For distributed, reactive programs, it is more appropriate to use safety and liveness properties [8, 15]. Interestingly, one of the basic liveness operators, **transient**, is existential, while some of the basic safety operators, **next** and **stable** , are universal.

Many other properties, however, are neither existential nor universal. In general, then, a component description will include *both* (i) a specification of the actual component behavior, and (ii) the strongest and weakest compositional properties (existential or universal) associated with this behavior. In this way, system engineers can select and integrate existing components based on their compositional properties. A complete component specification will therefore consist of: (i) a traditional interface with method signatures, (ii) a declaration of abstract state (iii) a formal behavioral specification, (iv) compositional properties (existential and universal), and (v) an informal description.

Figure 1 contains the description for the simple counter component introduced earlier. The formal specification is in terms of safety and liveness properties. The abstract variable $\#Inc$ and $\#Clear$ represent the number of times these methods have been invoked. The first two safety properties describe the effect of calling the `Inc()` and `Clear()` methods. The last safety property indicates that the component does not spontaneously change the value of `count`.

In the existential part of the component description, the existential properties of this component are given. In this case, the **guarantees** property states that in any system where `Clear()` is never invoked, the count is nondecreasing. If this component had any strongest existential or weakest existential properties, they would also be given in this section.

3.2 Publishing and Searching Speci£cations

In order to support reuse, components and their associated descriptions of behavior must be easy to find and identify. Component providers must be able to advertise or publish these descriptions and component consumers must be able to browse or search these advertisements. In the case of distributed components, all industrial middleware technologies already support some form of publication service

or catalogue for syntactic interfaces. These services or catalogues can be naturally extended to include the augmented interfaces described in the previous section.

The CORBA standard provides the Trading Service as a common interface for component consumers to identify registered objects based on various interface criteria. We define a new service, with a backward compatible publication format, to include compositional properties. Backward compatibility is important since we do not wish to mandate that all CORBA component consumers subscribe to an enriched trading service. Similarly, the .NET framework uses the UDDI [1] registry model for publishing information. The information provided by every component must follow a standard format. The UDDI framework allows us to define the publishing format as a unique tModel. Every component that needs to publish its compositional behavior, must use this tModel data structure to indicate compliance with this publishing format.

An important design feature of any publication (or searching) service is that it be flexible. It should accomodate behavioral descriptions of various degrees of completeness. Descriptions (and searches) with more information should be correspondingly more precise and high-confidence. However, descriptions with little behavioral information should also be allowed and seamlessly incorporated under a single paradigm of component identification and binding. The amount of information in a component's behavioral description will vary, in part, based on the effort of the interface provider and also based, in part, on the nature of the component itself. For example, the counter component from Figure 1 does not have any liveness properties.

3.3 Tool Support for Integration

In a compositional development environment, ideally the major development work should lie in decomposing system requirements, and locating components that can provide these system properties. A development framework should facilitate the task of reading accompanying component specifications. Moreover, an integrated development environment should seamlessly obtain a component's specification, display it to the developer, and calculate the result of the composition of different components (based on their accompanying compositional specifications). For example, on obtaining the counter component's specification, the tool would highlight that any system containing this component will have its existential properties.

Another useful task to automate is checking whether two given specifications are a match. Searches based on matching specifications are difficult, because a match must be made between a property as it is specified by the client and the specification as it is provided by the implementer. For example, the counter component is specified in terms of a single integer variable in its abstract state. However, when searching for a monotonic counter component, it is unlikely that the client will provide a syntactically matching specification. The client's target specification will reflect the domain of interest for the client's system. Model checkers can be used to check for matches in certain cases [4]. While it would be useful to be able to automate the decomposition of a given system specification into smaller specifications, or to calculate the result of combining specifications, or to determine whether a given property is existential or universal, these tasks are, in general, incomputable.

```
interface counter {
  void Inc()
  void Clear()
  #pragma state short count
  #pragma spec initially.(count == 0)
  #pragma spec (Forall j,k: #Inc == j && count == l
    next #Inc == j+1 ==> count = l+1)
  #pragma spec (Forall j,k: #Clear == j
    next #Clear == j+1 ==> count == 0)
  #pragma spec (Forall j,k,l:
    #Inc == j && #Clear == k && count == l
    next (#Inc == j && #Clear == k) ==> count == l)
  #pragma Existential (invariant.(#Clear==0)
    guarantees (Forall k: count==k next count >= k))
}
```

Figure 2: cidl Description of Counter Component

4. APPLICATION OF OUR PARADIGM TO EXISTING FRAMEWORKS

We examined both CORBA and .NET to determine the changes and challenges involved in constructing systems compositionally following our paradigm.

4.1 CORBA

CORBA is used to build large heterogeneous systems that use the services of distributed components. These distributed objects can be viewed as components and they provide services based on calls made to the methods they implement. In this model, composition is simply the acquisition of an object reference by another object. This object reference permits method invocations and hence interaction with a component's environment. (Here we ignore the Event Service, which provides event streams for interacting with the environment.)

In order to allow developers to specify their components, we developed, in earlier work, a certificate-based extension to the CORBA IDL, *cidl* [16, 7]. Ordinary IDL captures interface names and method signatures. The CIDL extension allows the inclusion of abstract state and behavioral properties in a component's IDL. The extension makes use of pragmas to add information to the specification, in this way guaranteeing its backward compatibility with ordinary IDL. We further extend CIDL to include existential and universal properties. For example, Figure 2 shows the CIDL description for the counter component. This allows the component to store its compositional description in its idl interface itself, which is where the description belongs.

The CORBA Naming Service allows clients to retrieve references to components based on interface name. By contrast, the CORBA Trading Service allows a richer set of queries based on method signatures and simple component properties. We leverage the discipline of publishing component interfaces with the Trading Service to support the publication of compositional behavioral descriptions as well. To this end, we define a new CORBA service type, the `ComponentSpec` service. Every CORBA service type has a number of properties. The property definitions for our new `ComponentSpec` service type are given in Figure 3.

In order for a trader to be able to advertise service offerings from components meeting this `ComponentSpec` service, the trader needs to have added this new service type to its type repository. The trader must also provide the functionality to allow component providers to create service offerings

PropertyName	PropertyType	PropertyMode
AbstractState	CORBA::tcstring	Mandatory
Interface	CORBA::tcstring	Mandatory
Safety	CORBA::tcstring	Mandatory
Liveness	CORBA::tcstring	Mandatory
Exist	CORBA::tcstring	Normal
Univ	CORBA::tcstring	Normal
Informal	CORBA::tcstring	Normal

Figure 3: Properties for `ComponentSpec` Service

for this new type, allow client applications to search for service offers, perform imports, *etc.*

Now, any component that is developed to support these kind of compositional specifications, needs to export a service offer of `ComponentSpec` service type to the trader with all the fields filled in appropriately. The first four properties of the service must be provided, while the rest are optional.

4.2 .NET

The .NET framework is Microsoft's framework for building and deploying XML-based web-services. In general, web service providers deploy services that can be used by distributed client applications. At the time a web-service is being used by a client application, it fits our model of a distributed component that is dynamically composed with the client application. Thus, every web service provider must provide all the information needed by clients in order to reason about the behavior of this web service in their client application. These web-services will use the UDDI business registry to publish their interfaces and allow searches for services. Thus, the UDDI registry is the ideal place for our component specification to reside, as it will contain all the information required by a client application to understand how the service will interact with the rest of the application. A developer looking for some component to satisfy a system specification will search the UDDI registry for a service specification that matches or can provide the system requirement.

UDDI uses a data structure called the "tModel" to define industry standards. In order to standardize our component description format, we need to define a new tModel, `ComponentSpec Service`, that will be used by web services that wish to indicate compliance with this compositional specification format. Figure 4 is our ComponentSpec service tModel. This tModel will be completely described in a WSDL [5] document *cs.wsdl* that describes the service interface and protocol bindings for obtaining the component specifications. WSDL is an XML based language, that allows us to describe this interface in a uniform manner. Figure 5 illustrates the *cs.wsdl* WSDL description of our `ComponentSpec` service.

The UDDI businessService data structure is used to describe the services provided by a web service and the location where the web service can be accessed. Any web service provider (or component provider in our case) who wants to provide services that conform to our compositional specification format will use this `ComponentSpec` service tModel in the UDDI businessService data structure used to describe access to their service. In addition, the businessService will provide the accesspoint or location of the service. So, for example, the provider for the counter component will use

```
<tModel authorizedName="..." ...>
  <name> ComponentSpec Service </name>
  <description xml:lang="en">
    WSDL Description of a ComponentSpec
    service interface
  </description>
  <overviewDoc>
    ...
      <overviewURL>
        http://Component-definitions/cs.wsdl
      </overviewURL>
    ...
  <keyedReference tModelkey="uuid:C12345..." ...
    keyvalue="wsdlSpec"/>
</tModel>
```

Figure 4: tModel for the ComponentSpec Service

```
<? xml version="1.0">
...
<definitions name="ComponentSpec"...
  xmlns : xsd1=".../componentspec.xsd"...>
  <types>
    <schema
      targetNamespace=".../componentspec.xsd"
      Service xmlns=".../XMLSchema">
      <element name="AbstractState">
        <complexType>
          <all>
            <element name="abstractState"
            type="string"/>
          </all>
        </complexType>
      </element>
      <element name="FormalSpecification">
        <complexType>
          <all>
            <element name="formalSpec"
            type="string"/>
          </all>
        </complexType>
      </element>
      ...
    </schema>
  </types>
  ...
  <message name="GetAbstractState">
    <part name="body"
      element="xsd1:AbstractState"/>
  </message>
  ...
  <message name="GetExistentialProperties">
    <part name="body"
      element="xsd1:ExistentialProperty"/>
  </message>
</definitions>
```

Figure 5: WSDL Description for the ComponentSpec Service

```
<businessService businessKey="..." serviceKey="...">
  <name> CounterService </name>
    ...
    <bindingTemplates>
      <bindingTemplate>
        <accessPoint urlType="http">
          "http://www.sample.com/counter"
        </accessPoint>
      <tModelInstanceDetails>
        <tModelInstanceInfo tModelkey=
          "ComponentSpecification key">
        </tModelInstanceInfo>
      </tModelInstanceDetails>
    </bindingTemplate>
  </bindingTemplates>
</businessService>
```

Figure 6: businessService structure for counter component

our `ComponentSpec` service as shown in Figure 6.

Thus, incorporating the compositional behavior of a component into its specification and make use of it in the .NET framework is just a matter of conforming to one extra tModel.

5. CONCLUSION

In his paper on composition [9], Lamport states that if a component is likely to be used in multiple systems, an open system specification needs to be written for it. He goes on to say that although the composition of open system specifications is an attractive idea, it is unlikely that the thinking will change within the development community in the next 15 years so that engineers would make the extra effort of proving open system properties on the components they build. However, with the advent of web-services and the UDDI registry, designed with the intention of allowing service providers to provide services dynamically to clients, it is only natural that a description of any service should contain information about how the service will interact with the client application. Thus, it has become imperative for developers to provide this compositional information in order to increase the likelihood of the component being used.

The component description we defined earlier provides a systematic way for developers to provide what amounts to an open system specification. These compositional specifications can be incorporated into the standard interface descriptions of both CORBA and .NET in a natural manner. If component-based system development is to become the norm, it is necessary that component developers provide component specifications that are complete and informative enough for clients to be able to reason about the interaction of these components in their systems. The paradigm we discuss in this paper allows us to reap the full benefit of compositional development.

6. REFERENCES

[1] Uddi executive white paper. available at www.uddi.org/pubs/UDDI_Executive_White_Paper.pdf.

[2] Charpentier, M., and Chandy, K. M. Reasoning about composition using property transformers and their conjugates. In *IFIP TCS* (2000), pp. 580–595.

[3] Charpentier, M., and Chandy, K. M. Theorems about composition. In *Mathematics of Program Construction* (2000), pp. 167–186.

[4] Clarke, E. M., Grumberg, O., and Peled, D. A. *Model Checking*. The MIT Press, 1999.

[5] Curbera, F., Ehnebuske, D., and Rogers, D. Using wsdl in a uddi registry 1.05. available at http://www.uddi.org/pubs/wsdlbestpractices-V1.05-Open-20010625.pdf.

[6] Hopkins, J. Component primer. *Communications of the ACM 43*, 10 (October 2000), 27–30.

[7] Krishnamurthy, P., and Sivilotti, P. A. G. The specification and testing of quantified progress properties in distributed systems. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)* (Toronto, Canada, May 2001), IEEE and ACM SIGSOFT.

[8] Lamport, L. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering SE-3*, 2 (March 1977), 125–143.

[9] Lamport, L. Composition: A way to make proofs harder. In *International Symposium on Compositionality – The Significant Difference* (September 1997).

[10] Larsen, G. Component based enterprise frameworks. *Communications of the ACM 43*, 10 (October 2000), 25–26.

[11] Meyer, B. *Object-Oriented Software Construction*, second ed. Prentice-Hall, Upper Saddle River, New Jersey 07458, 1997.

[12] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, February 2001. Revision 2.4.2.

[13] Pree, W. Component based software development-a new paradigm in software engineering? *Springer-Verlag Software-Concepts and Tools*, 18 (1997), 169–172.

[14] Richter, J. Microsoft .net framework delivers the platform for and integrated service oriented web.

[15] Sivilotti, P. A. G. *A Method for the Specification, Composition, and Testing of Distributed Object Systems*. PhD thesis, California Institute of Technology, 256-80 Caltech, Pasadena, California 91125, December 1997. Available as CS-TR-97-31.

[16] Sivilotti, P. A. G., and Giles, C. P. The specification of distributed objects: Liveness and locality. In *Proceedings of CASCON '99* (Toronto, Ontario, Canada, December 1999), S. A. MacKay and J. H. Johnson, Eds., pp. 150–160.