A Cottage Industry of Software Publishing: Implications for Theories of Composition *

K. Mani Chandy, Paulo A.G. Sivilotti, Joseph R. Kiniry

Department of Computer Science,
California Institute of Technology, m/c 256-80,
Pasadena, CA 91125
[mani,paolo,kiniry]@cs.caltech.edu
http://www.infospheres.caltech.edu/

Abstract. This note explores the use of UNITY-based theories to facilitate a cottage industry of software publishing. The requirements for such an industry are discussed, the appropriateness of UNITY specification and compositional theories for these requirements are analyzed, and further research opportunities in this area are identified. This work is based on joint work with Beverly Sanders, and the ideas discussed here have been explored jointly with Paul Sivilotti and Joseph Kiniry.

1 Introduction: A Cottage Industry of Software Publishing

Our vision of a cottage industry of software publishing is outlined in this section. The discussion is couched in terms of object-oriented systems, but it applies equally well to other systems. In particular, much of our experience that prompts the following discussion is from the design, construction, and use of Infospheres 2.0.[CKRZ98]

The formative years of a cottage industry of software components is taking place today in the form of the publication and distribution of code, primarily in the form of Java applets and Beans and ActiveX controls. The term "cottage industry" derives from the analogy to the cottage industry in manufacturing just prior to the Industrial Revolution, a time when lone experts painstakingly manufactured hand-tooled, customized items. This time ended in the Industrial Revolution, resulting in a new world of manufacturing where standards, interchangeable parts, and low-priced, high-reliability items were the norm.[Cox96]

We expect a similar change in the software component industry, and believe that such a shift will be motivated more by the publication and dissemination of software by individuals than by companies. We expect this propagation of new code will take place in a decentralized fashion, similar to publication of documents on the Web today.

The discussion is presented in terms of publication on the World Wide Web, though the ideas are generally applicable. The Web is widely used for publishing and accessing documents. We treat programs as special documents, and we use the Web for publishing and accessing programs.

* Supported in part by NSF Grant CCR-9527130

We consider publications of (i) classes, (ii) objects which are instances of classes and (iii) relationships between classes and relationships between objects. Most of this short note is about publications of classes and objects; we have insufficient space to discuss relationships at length.

The primary difference between publishing classes and objects is in their life-cycles. An object usually has a shorter lifetime than a class, but this is not always the case. An additional distinction is that there are usually many objects of a given class, but an object is related to few classes.

We take a broad view of "publishing" classes. You publish a class by giving a specification of the class (and possibly other information as well) and indicating where the implementation of the class may be found. We are not concerned with the implementation (i.e., the code for the class) other than that it can be retrieved. The implementation may be in terms of object code, and the implementation could be encrypted. A class can be composed from other classes, in which case the implementation has references — such as URLs (Universal Resource Locators) on the Web — to the other classes.

In our view, a class has three parts: (i) a *specification*, possibly additional description of the class, (ii) an *implementation* of the class, and (iii) *evidence* demonstrating that the implementation matches the specification. We call this evidence "validity evidence".

Examples of validity evidence include a formal proof of correctness carried out by a mechanical theorem prover or an expert formalist, a comprehensive test suite, or references to other people who have used the class successfully. Certain kinds of evidence give users more confidence than others. Publishing a class means putting the description and validity evidence in URLs and indicating where the implementation of the class can be found. Objects and people can search the Web for appropriate classes and then use them.

Similarly, publishing an object means: (i) putting a description of the object in a URL and thus allowing other objects to discover your object, (ii) indicating where the object can be found so that other objects can call methods on it, and (iii) referencing a document containing evidence that the implementation of the object matches its description. Your object's implementation may not (and usually will not) be available to others, but you provide ways for other objects to interact with the objects you publish. Collections of objects may organize themselves into larger objects, and these self-organized composed objects, in turn, can publish themselves.

There are many aspects to facilitating a cottage industry of software and many issues in publishing, including:

1. **Discovery:** deals with finding classes and objects that are published on the Web.
2. **Composition:** deals with (i) composing classes to form new classes and then publishing these new classes, in turn, on the Web, and (ii) likewise discovering, composing, and publishing objects.
3. **Scalability:** deals with ensuring that the world-wide pool of classes and objects can contain millions of items.
4. **Commerce:** deals with payment, licensing and legal issues. We do not discuss commerce in this short note.
5. **Security and other "ilities":** Security deals with ensuring that your resources (software, data, and hardware) are not misused by another person or object. There

are many other attributes of these systems including performance and reliability. We do not consider security issues here.

Next, we discuss discovery and composition, and then carry out a requirements analysis for these activities.

2 Discovery

Discovery is the process by which an object discovers other objects and classes from a large pool of objects and classes. In our case, this pool is implemented by the Web. An object searches for classes and objects that implement a specification and it then composes these classes and objects. So, one of the key questions is how are objects described and specified so that other objects can find them?

We first consider discovery for classes and later, for objects.

2.1 Discovery of Classes

There are two ways of thinking about classes: one is in terms of its syntactic interface and the other is in terms of semantics. Of course, all of us have to be concerned about *both* the syntax and semantics of a class interface. In practice (e.g., CORBA IDL) most object systems carry out compile-time or run-time checks of syntax, but expect semantic compatibility checking to be the exclusive responsibility of the programmer. A cottage industry of publishing is facilitated by pushing compositionality to its limits: The compatibility of interfaces are determined by semantics and the syntax of method calls is negotiable. Next, we follow this line of reasoning.

Classes as Abstract Data Types and Theorems. Think of a class as definitions in two notations and an assertion (with supporting documents such as proofs or test suites) connecting the two definitions. One notation is the programming language and the other is the specification language. The fundamental issues do not depend on precisely how a specification is given or how the assertion relating specification and implementation is demonstrated.

When you publish a class you (in effect) publish three things: its specification, its implementation, and an assertion relating the two. The form in which these three components are published can vary widely. You can, for instance, give URLs for the description, implementation, and assertion coupled with proof.

Next, we discuss the specification and assertion. (There is not a great deal to be said about the implementation other than whether the source code and associated documents are available to the user.)

Relationships. A relationship is an assertion about a set of classes, or a set of objects, with evidence demonstrating the validity of the relationship. For instance, a relationship could be that two classes are equivalent. Relationships are different from classes and objects. Objects searching for classes or other objects can find relationships and use these relationships in finding appropriate objects.

Ontologies. We use software classes and types to reflect classes of "real world" objects. At some basic level, the relationship between a software class and the class of real-world object that it reflects cannot be proved: it has to be treated as a given. For instance, we may accept without proof an assertion that a dollar class truly reflects a real dollar. Classes that are accepted without evidence to be reflections of the corresponding real-world classes are called *ground* classes of a component ontology.[Gru93]

People can publish relationships between classes including relationships between ground classes. For instance, the Web may contain a relationship between a U.S. dollar and a Canadian dollar. Here too, we can accept or discount the published relationship based on several factors.

We can prove properties about relationships between classes. For instance, we can prove that one class is a refinement of another. Such proofs are relevant to our infrastructure, but demonstrations of the "equivalence" between classes and reality are not.

We use markup languages like XML[BPS97] to describe object ontologies. An example of an evolving, ontology-focused specialization of XML is the Ontology Markup Language (OML).[Ken98]

Permanence of Specifications and Theorems. People can publish theorems about class specifications. For instance, you may be publish a "theorem" that ticket type `T` of airline `X` is a refinement of ticket type `T'` of airline `X'`. (For the purposes of this note we ignore important issues such as version numbers of classes; therefore we assume that ticket type `T` of airline `X` refers to a unique class, whereas in reality the type `T` is likely to have a version number or some other unique id.)

To the extent that specifications and theorems are correct, the collection of specifications and theorems can only change by addition. Once a theorem is added to the collection it remains there until proved incorrect. A specification of version `V` of class `C` remains unchanged for ever (unless it is proved wrong). So, we can think of publications of ground classes, specifications, and theorems relating classes, in terms of a growing world-wide pool. [SC97]

Search for Classes. Imagine that you have been given the task of developing a class for a given specification. If you can find a class with this specification on the Web, and it has the appropriate attributes such as cost and reputation of the vendor, then you use this class directly. If you cannot find this class then you may want to search for classes that you can compose to construct your class. How are you going to search for potential component classes?

The search problem is as follows: Given a specification and a large pool of specifications of components, mechanically select a set of components that can be composed to obtain a system with the given specification. This problem is intractable. So, let us consider a much simpler problem. You come up with a specification for a desired component and you now search the Web for a component with this specification or a stronger specification. (How you come up with this specification is not our concern here.) Since there is no standard normal form for specifications, you need a theorem-prover to prove that one specification is equivalent to, or is a refinement of, another specification. Since theorem-provers may require interaction from people, and they often require a lot of time, we want to facilitate search by some other means.

One approach is to associate with each class a description that includes a set of attribute-value pairs, where both the attribute and value are strings. Typical attributes include the class's owner, cost, complexity, size, etc. This description is used to narrow the search, and only then do we attempt to prove theorems about the equivalence of specifications. Narrowing the search in this way has a disadvantage: The use of attribute-value pairs may incorrectly exclude some classes merely because the attributes and values were slightly different. We have are still exploring designs of search engines that find components that satisfy a given specification.

2.2 Discovery of Objects

The following scenario is one that we use in evaluating our model of a cottage industry of software publishing.

You are appointed the chair of a program committee and you want to determine the time for a video-conference of the committee. You instruct your appointment-scheduler object to make a tentative appointment in the calendars of all committee members. You give your appointment-scheduler object the URLs of the home pages of committee members. Can your object carry out its task? (Note: Many research groups use the term "agent" for our "objects.")

In our solution, there is a directory of objects associated with a home page. This directory contains a specification (or at least a description) of the objects belonging to the owner of the home page. The directory in your home page is searched to find the appropriate appointment-scheduler objects that schedule your appointments.

The attribute-value pairs describing an object may extend the attribute-value pairs of its class. For instance, you may have several objects, from the same class, that place bids for you in on-line auctions: for example, one object to buy books and another to buy CDs. The book-buying object will have some attribute-value pairs that are different from the CD-buying object. If I want to start an auction of rare books on Mogul miniature paintings, my auction-initiation object will have to find appropriate book-buying objects and then ensure that the semantics of the interfaces of the objects are compatible.

The research problem we discuss next is the following: *We may need to compose objects that were not designed to be composed with each other*. The next paragraph discusses the motivation for this problem.

Imagine that there are a million people world-wide creating classes and publishing them on the web, possibly by referencing them in home pages of individuals or small businesses. Different members of your program committee are likely to have classes written by different people, and it is possible that these classes were not derived from a common specification, and were not designed to be composed with each other. If we want to facilitate a cottage industry of software publishing we have to take compositionality (or "plug-and-play" in the vernacular) to its limit, and allow objects that were not designed to be composed with each other to negotiate protocols with each other so that composition is possible.

Consider the case where the interface between appointment-scheduling objects is very simple; one object calls a method on another object to make an appointment for a specific start time and duration. The specification of the method can be defined in

several ways; for instance in terms of its pre and post conditions. For the same specification there can be several ways of implementing the method: We need to give the method name, the names and order of the arguments, and the protocol used to "serialize" an argument of the method and pack it into a message at the caller and unpack it at the receiver. Given compatible specifications, the objects must be able to negotiate the rest. Publishing specifications is therefore critical to facilitating a cottage industry of software publishing.

In summary, our approach defines an interface in terms of the semantics of method calls. Interfaces are compatible if the semantics of the method calls required by an object are compatible with the semantics provided by the other object. Everything, other than the semantics of the method calls, is negotiable. As with classes, we use attribute-value pairs to narrow down a search for objects, and then use specifications to determine whether interfaces are compatible.

3 Composition

3.1 Composition of Classes

A class publication consists of (i) its specification, (ii) its implementation, and (iii) an assertion relating the specification and implementation, coupled with documentation demonstrating the validity of the assertion. When you create a class by composing other classes, your assertion that your composed class' implementation satisfies its specification is valid provided that your proof of composition is correct and the assertions about correctness are also valid for the component classes. Thus, the validity of your assertion depends on the validity of other assertions about components which, in turn, depend on the validity of assertions of subcomponents, all the way down to the ground classes.

Your publication of your composed class will include the implementation of its compositional structure. The implementations of the components may be obtained from the references (e.g., URLs) that you provide, or you may have some way to package and license the components directly. The publication must also contain the class specification and the proof of correctness.

One of the most important characteristics of an infrastructure that supports a cottage industry of software publication is that it facilitates one person creating and publishing a new component by finding and composing existing components. Reasoning about systems from the specifications, but not the implementations, of the components is central to this effort.

3.2 Composition of Objects

When we think about composing classes we usually think of an intelligent human being creatively putting classes together and proving the necessary theorems. By contrast, when we think about composition of objects, we should also consider mechanical composition. Consider, once again, the example of you as chair of a program committee asking one of your objects to determine a time for a video-conference meeting of your committee. One scenario is that you yourself, search the Web, find the appropriate objects belonging to your program committee members, modify and perhaps recompile

your object so that composition is possible, ask others in your committee to do the same, and only then carry out the collaboration. The other scenario is that your object does the searching, negotiation of protocols for interaction, and then sets up the collaborative structure. We are exploring the latter scenario.

4 Using UNITY to Facilitate the Software Cottage Industry

The central theoretical issue in this domain is that of demonstrating that a composed object satisfies its specification given its compositional structure and the specifications of its components. Next we show how UNITY and its derivatives are well suited to deal with this issue.

The most important characteristic of UNITY is that its theory, including its theory of parallel composition, is based on universal and existential composition in the predicate calculus. This characteristic is relevant to our vision of a cottage industry of software components. We need to be able to reason about object composition in the easiest way possible. For convenience, we summarize the relevant part of [CS95] which introduces program properties and shows how $\forall$ and $\exists$ forms the basis for the logic.

A *program property*, or just property for short, is a predicate on programs. Here, we use programs and objects interchangeably. A property is an *all-component* or universal property means that the property holds for a composed object if and only if the property holds for each of its components. Therefore p is an all-component property if and only if for any system with components G_k, where $k \in K$:

$$p.(\|k : k \in K : G_k) \equiv (\forall k : k \in K : p.G_k)$$

Here, $\|$ is the parallel composition operator, and so $(\|k : k \in K : G_k)$ is the system obtained by composing G_k in parallel for all $k \in K$. We use p, q and r for properties and G, H, and K for objects.

The safety properties *next* and *stable* are all-component properties. Invariant is weaker:

$$(\forall k : k \in K : \text{invariant}.X.G_k) \Rightarrow \text{invariant}.X.(\|k : k \in K : G_k)$$

where $\text{invariant}.X.G$ means that state-predicate X is an invariant of G.

Similarly, for *always* properties:

$$(\forall k : k \in K : \text{always}.X.G_k) \Rightarrow \text{always}.X.(\|k : k \in K : G_k)$$

A property is an *exists-component* or existential property means that the property holds for a composed system if it holds for any component. Therefore p is an exists-component property if and only if for any system with components G_k, where $k \in K$:

$$p.(\|k : k \in K : G_k) \Leftarrow (\exists k : k \in K : p.G_k)$$

The progress property *transient* is an exists-component property. In fact:

$$\text{transient}.X.(\|k : k \in K : G_k) \equiv (\exists k : k \in K : \text{transient}.X.G_k)$$

Other progress properties such as $\rightsquigarrow$ (leads to) are proved by judicious combinations of all-component and exists-component properties.

Some of the questions that arise in facilitating a software component industry include:

1. How much of the design of a component should be exposed? A specification in terms of a large number of next, transient and initial condition properties may expose too much of a component's design. If we give specifications in terms of always and leads-to properties then the specifications may not be compositional: We may be unable to prove the specifications for the composed object from its compositional structure and the specifications of its components.
2. Should the specification of a component be tailored to a particular set of environments for that component? Or, should the specification be totally general, and work for all environments? What, if any, are the negative consequences of giving specifications for totally general environments?
3. What are good structures for components? For instance, if components are objects, is composition simplified if arguments of methods are passed by value rather than by reference?
4. What is the appropriate degree of atomicity that can be implemented with reasonable efficiency in a large world-wide pool of interacting objects? How do we define transactions in this context, and how do we define interference between competing transactions?

We are still exploring these issues. Next, we suggest one set of answers to these questions. These answers follow from our exploration of using Infospheres 2.0 as an infrastructure for a pool of objects.

We have to be able to use always and leads-to properties, and we have to find some way to make these non-compositional properties appear to be compositional. Restricting reasoning about composition to next, transient and initial condition properties exposes too much of a component's design and is too tedious. Therefore, a specification for a component is given with some idea of its intended environments. We use objects where method calls pass arguments by value and not by reference because that is easier to implement and also doesn't make reasoning harder. And, we assume the existence of monitors to manage atomicity of operations.

We obtain compositionality by using *guarantees* properties from [CS95]. For program properties p and q, the program property $(p \ \mathrm{guarantees} \ \mathrm{q})$ holds for an object G means that for any object H in which G is a component, if p holds for H then q holds for H. Therefore, a guarantees property is an exists-only property. Further, the p and q in a guarantees property can be always and leads-to properties. When you publish a component you identify the systems in which expect that component to be used, and you then specify your object using guarantees properties where the left-hand side of the guarantees property (i.e., the p) is a property that you expect the composed system to have. This approach simplifies proofs of object composition, but it has a price: Your object may be useful in an application that you hadn't designed it for, and the guarantees properties that you used in the specification may not help in proving properties for that application.

See the references in [CS95] and [CM88,CM88,MS96,SC97] for further discussion and past work on the material of this section.

5 Conclusion

This paper is an exploration of issues that stem from attempting to facilitate a cottage industry of software components. We are exploring the use of UNITY because it has a simple compositional structure based on universal and existential quantification that gives rise to all-component and exists-component properties (respectively). There are many other issues including the ones we listed above. This is an ongoing effort, and we have just begun.

References

[BPS97] Tim Bray, Jean Paoli, and C.M. Sperberg-McQueen. Extensible Markup Language (XML) Proposed W3C Recommendation PR-xml-971208, Dec 1997. http://www.w3.org/TR/PR-xml.

[CKRZ98] K. Mani Chandy, Joseph R. Kiniry, Adam Rifkin, and Dan Zimmerman *Infospheres 2 Users Manual*, Mar 1998. http://www.infospheres.caltech.edu/.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[CS95] K. Mani Chandy and Beverly A. Sanders. Predicate transformers for reasoning about concurrent computation. *Science of Computer Programming*, 24:129–148, 1995.

[Cox96] Brad Cox. *Superdistribution : Objects As Property on the Electronic Frontier* Addison-Wesley, 1996

[Gru93] T. R. Gruber. A translation approach to portable ontologies *Knowledge Acquisition*, 5(2):199-220, 1993.

[Ken98] Robert Kent et al. Ontology Markup Language (OML) http://asimov.eecs.wsu.edu/WAVE/Ontologies/OML/OML-DTD.html.

[MS96] R. Manohar and P. Sivilotti. Composing processes using modified rely-guarantee specifications. Caltech technical report CS-TR-96-22, 1996.

[SC97] Paolo A. G. Sivilotti and K. Mani Chandy, A Distributed Infrastructure for Software Component Technology. Technical Report CS-TR-97-32, Department of Computer Science, California Institute of Technology, September 1997. ftp://ftp.cs.caltech.edu/tr/cs-tr-97-32.ps.Z.