WaterBeans: A Custom Component Model and Framework

Kurt C. Wallnau, Daniel Plakosh

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, 15213, U.S.A.
+1 412 268 3265
{kcw,dplakosh}@sei.cmu.edu

Introduction

The development of component-based systems is supported by commercial component technologies such as Sun's Enterprise JavaBeans™ and Microsoft's COM+. But these technologies only address the needs of a particular class of systems, one that we might refer to as *distribute, transactional enterprise systems*. Notwithstanding the significant market size for this class of system, there are many other classes of system that could benefit from a component-based approach but for which EJB and COM+ provide little support. For example, systems requiring near real-time performance or high availability have requirements that are not addressed by either EJB or COM+. For these classes of system a *custom component technology* is needed.

We encountered one such situation in work we performed for the US Environmental Protection Agency (EPA). The EPA Department of Water relies upon the use of mathematical models of pollutant "fate and transport," the chemical and hydrological processes that underlie the quality of water. A substantial amount of software has been written to use such models in simulating water quality. These simulations, in turn, are used for regulatory purposes, as well as for improving the science that produced these models in the first place. Unfortunately, the models and the software that uses these models have grown sufficiently complex that they now act as an inhibitor to the advancement of water quality science. What EPA needs is a software architecture that facilitates the substitution of new mathematical models for old, and new simulation techniques for old. In short, they need a component-based approach to water quality modeling.

Water quality simulations introduce a number of demanding requirements not addressed by commercial component technologies. First and foremost is the performance requirement. Simulations may draw upon numerous sources of data, some of which generate a large volume of data. Analysts need to visualize the results of a simulation as it progresses, because the volume of data produced makes data logging for later replay impractical. A second requirement has more to do with the intended audience for a component-based technology for water quality modeling and simulation. The intended users range from the chemists and physicists who create water quality models, to civil engineers who select and combine the models needed to, for example, design urban sewage systems, to policy makers who wish to understand the implications of regulations pertaining to allowable concentrations of pollutants in rivers. The technology needs to support all of these kinds of users, none of which may be properly thought of as "sophisticated" with respect to software technology and software engineering practice.

We first describe a reference model for software component technology, so that it is clear what we mean when we say that a custom component technology is needed. We then describe WaterBeans, a prototype component technology in terms of this reference model. We briefly describe three different applications that we developed using WaterBeans in order to demonstrate its applicability to a class of systems represented by EPA's software. Finally, we describe our interest in using WaterBeans to explore the question of component certification and prediction of application properties from certified component properties.

Software Component Technology

As part of an ongoing study of component-based software, we have conducted an extensive literature survey, studied numerous software component technologies, and examined many systems that were purported to be "component-based." From these investigations we have detected a pattern of concepts that recurs in component-based systems and technologies, as depicted in Figure 1, below.

Briefly, a *component* (①) is a software implementation that can be executed on a physical or logical device. A component implements one or more *interfaces* that are imposed upon it (②). Doing so means that the component satisfies certain contractual obligations (③). These contractual obligations ensure that independently developed components obey certain rules so that components interact (or can not interact) in predictable ways, and can be *de-*

ployed into standard build-time and run-time environments (④). A component-based system is based upon a small number of distinct component *types*, each of which plays a specialized role in a system (⑤) and is described by an interface (②). A *component model* (⑥) is the set of component types, their interfaces, and, additionally, a specification of the allowable *patterns of interaction* among component types. A component framework (⑦) provides a variety of runtime services (⑧) to support and enforce the component model. In many respects component frameworks are like special-purpose operating systems, although they operate at much higher levels of abstraction.

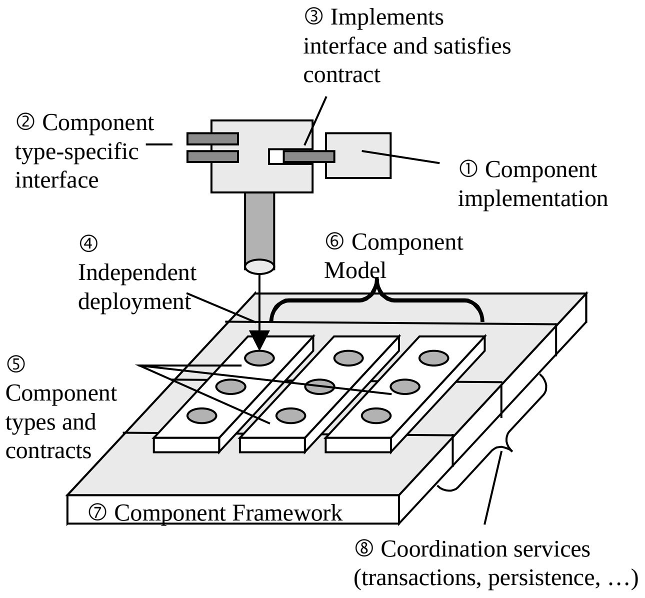

③ Implements interface and satisfies contract
② Component type-specific interface
① Component implementation
④ Independent deployment
⑥ Component Model
⑤ Component types and contracts
⑦ Component Framework
⑧ Coordination services (transactions, persistence, …)

Figure 1: Reference Model for Software Component Technology

When we say that a *custom* component model may be needed, we mean that a component framework must be constructed to support some application-specific component model. Of course, what is custom-built today may be used off-the-shelf tomorrow. But the question of how to develop industry-standard component models and frameworks is one best left to another position paper.

WaterBeans

The WaterBeans[1] component models defines only one *type* of component. WaterBeans components have *typed* input and output ports that consume and produce streams of bytes. WaterBeans components are like JavaBeans in that the component type is really a meta-type; components must conform to an interface *pattern* defined by the meta-type. Thus, a WaterBean component may have as many or as few input or output ports as needed. The *type* of the port is defined by the name given to that port by the component developer.

Components can be implemented in any programming language. The only requirement is that they implement the required interface and export this interface in the form of a Microsoft DLL. Thus, WaterBeans adopts a binary standard for components. The required interface consists of several operations that allow the framework to interrogate the component about its input and output ports (a "poor man's" introspection capability), what icon should be used in the visual composition environment, and similar bookkeeping operations. The interface also consists of operations that are used by the framework to manage the runtime execution and interaction of components. The complete API is documented in [Plakosh 99].

Applications are constructed from WaterBean components by linking the output port of one component to one or more input ports of other components. The WaterBeans framework provides a composition environment that performs primitive type-by-name matching to ensure that input and output ports "match." This approach is, depending on how you look at it, either too liberal or too restrictive. In either case it must be admitted that this typing scheme requires that component developers have some *a priori* agreement about the types of data that are used in assembling applications and what these data types are called. Applications therefore resemble a pipe-and-filter style of system, where the filters can supply different data streams to different pipes, and receive data from different pipes. An annotated snapshot of the composition environment is provided in Figure 2.

The WaterBeans framework also provides the runtime environment for applications. The most interesting aspect of the environment is the execution scheduler. The scheduler calculates, for a given topology of components, an execution ordering for components, based upon component precedence. Thus, if component B depends upon input from component A, then component A will execute *before* component B. Where there are several possible "schedules" the framework will select one non-deterministically[2]. The scheduling policy is non-preemptive. Thus the behavior of an application depends upon the components to return control to the framework in a timely way. Also, there is only a single thread of execution in a WaterBeans application that is shared by the components and the framework.

[1] We must note that the choice of the name "WaterBeans" is unfortunate in that WaterBeans are not (necessarily) written in Java.

[2] There are ways to force a particular ordering using "triggers." Refer to [Plakosh 99] for details.

WaterBeans Application Domains

The component model and framework just described has potential utility beyond EPA water quality modeling. To demonstrate this potential we applied WaterBeans to three distinct application domains. In practical terms, applying WaterBeans to an application domain means developing a family of components that share some functional scope and that agree upon the data types necessary for their integration.

Waveform Visualization and Manipulation

The first application domain comprises components that can generate, add, subtract, and display waveforms. Although this is a trivial application domain (which is not to say that the coding of all components were trivial), something simple was needed to demonstrate WaterBeans concepts to new users. The waveform components are distributed with the publicly-available WaterBeans download.

Audio Visualization and Manipulation

The second application domain comprises components that can sample audio signals from a CD, inject signals into an audio stream, combine audio streams, feed these streams to the audio player and visualize the streams that are played. Although this is also a trivial application domain it is one that has fairly severe performance requirements and thus afforded us with a simple means to demonstrate and experiment with the performance of the WaterBeans framework.

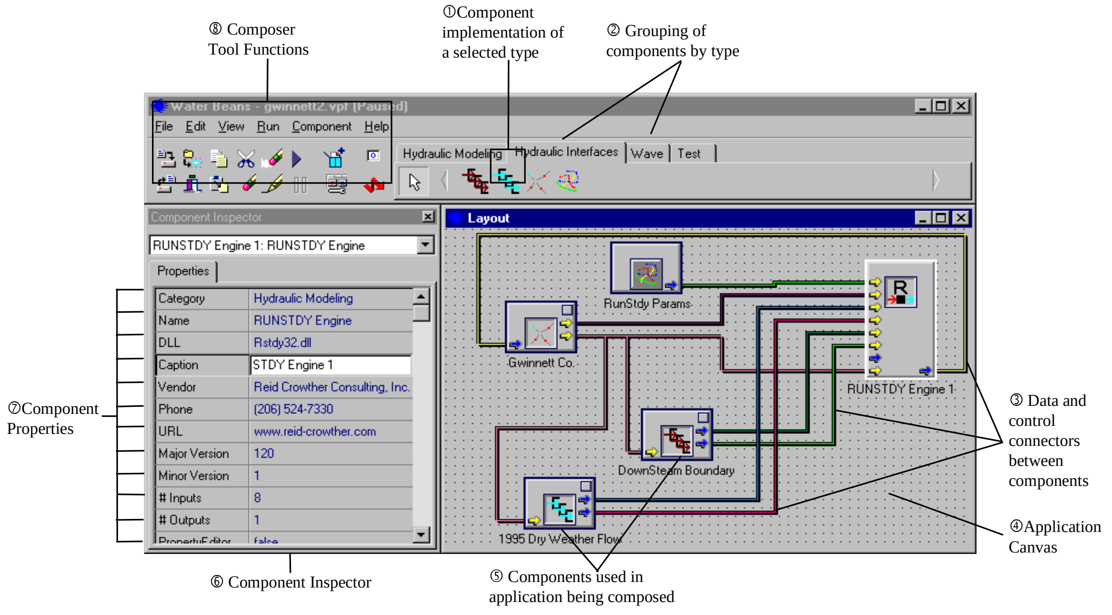
Water Quality Modeling Simulation

The third and last application domain comprises components that produce simulation data, allow modeling of urban sewage systems, strip charts for displaying water volumes and pollutant concentrations, and computational engines for solving systems of equations. The components used in this application were a mix of custom-built components and wrapped "legacy" FORTRAN components. This is a non-trivial application domain and it demonstrates that WaterBeans is scaleable to build applications at least as complex as those currently being used by EPA for water quality modeling.

Limitations and Next Steps: Predictable Assembly from Certified Components

There are, naturally enough, many limitations to the WaterBeans implementation. There is no means for hierarchical composition, and thus entire applications must be constructed on one palette. The scheduling policy is fixed; it, too, should be a component, albeit one that modifies framework behavior rather than application behavior. The component model should allow separate threads of control in components and framework. Some support for distributed composition and execution would be nice, too. All of these would be useful implementation extensions, and perhaps we will find some time to implement them (or, better still, convince someone else to implement them).

More interesting to us, however, is the question of how WaterBeans can be enhanced so that we can predict the properties of assembled applications (in particular, per-


⑧ Composer Tool Functions
①Component implementation of a selected type
② Grouping of components by type
Water Beans - gwinnett2.vpf [Paused]
File Edit View Run Component Help
Hydraulic Modeling
Hydraulic Interfaces
Wave
Test
Component Inspector
Layout
RunStdy Params
Gwinnett Co.
RUNSTDY Engine 1
DownSteam Boundary
1995 Dry Weather Flow
⑦Component Properties
③ Data and control connectors between components
④Application Canvas
⑥ Component Inspector
⑤ Components used in application being composed

RUNSTDY Engine 1: RUNSTDY Engine

Properties

Category	Hydraulic Modeling
Name	RUNSTDY Engine
DLL	Rstdy32.dll
Caption	STDY Engine 1
Vendor	Reid Crowther Consulting, Inc.
Phone	(206) 524-7330
URL	www.reid-crowther.com
Major Version	120
Minor Version	1
# Inputs	8
# Outputs	1
PropertyEditor	false

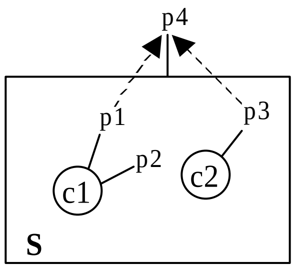
Figure 2: WaterBeans Composition Environment

formance properties) from the component model, and from the known properties of components and framework. There has been increasing interest in working towards the goal of building applications from trusted software components, perhaps even certified software components. But approaches such as those described by the Trusted Components Initiative[3] tend to take a component-centric view of certification, and, we believe, thereby miss an opportunity provided by software component technology to provide a useful foundation for component certification. We devote the remainder of this position paper to that idea.

Certification and Compositional Reasoning

The unspoken premise behind component[4] certification is that there is a causal link between those properties of a component that are certified and the properties of an end system that use that component. The more confidence we have in this link the more value will accrue from component certification. At the extreme we may achieve 100% confidence (or trust) in the causal link. At this extreme, once a property has been established for a component it is unnecessary to certify that the end system has obtained this same property, since we know this to be true *by definition*.

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



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

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

Note that the value of these theories goes beyond predicting system properties. Such theories also tell us which properties can be predicted (p4), and, just as important, which component properties we need to know about in order to make these predictions (p1 and p3). *The value of a certification regime for component-based systems is directly linked to the strength of our compositional reasoning*. Without compositional reasoning we can not know which properties of components and frameworks to certify. With weak compositional reasoning the value of certification will be questionable and therefore the economic incentives for 3rd-party certification will never be sufficient to spur industry investment.

Compositional Reasoning and Software Architecture

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

- A description of the analysis problem solved by the ABAS.
- A characterization of the stimuli to which the ABAS responds and the quality attribute measures of the response.

[3] See http://www.trusted-components.org/.

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

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

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

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

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

We have done some preliminary "thought experiments" on how this connection can be put to the test with WaterBeans. For example, we used the *concurrent pipeline* ABAS (see [Klein 99 for details]) to analyze performance attributes of a sample WaterBeans application in the audio visualization and manipulation domain. This experiment produced several interesting and encouraging results. First, the reasoning framework guided us on how to conceptualize the component topology (for example, by combining some components) in a way that made performance analysis simpler. Second, the reasoning framework clearly identified those properties that were required of components and framework in order to do performance prediction. Third, the ABAS exposed some of our WaterBeans design decisions to criticism, in particular those decisions that rendered the use of the reasoning framework less effective.

This last point in particular is worth emphasizing, because it suggests that a component model and framework be designed with ABAS's from the outset. What we are suggesting is something analogous to what motivated structured programming. If analyzing computer programs with arbitrary GOTO structures is problematic, the solution is not to improve the analysis tools (this *might* be useful), but rather to structure programs so that they are analyzable (this *will* be useful). Similarly, component frameworks can enforce a variety of architectural constraints that are expressed in a component model. Properly designed (i.e., structured), a component model can make systems more easily analyzable with respect to one or more properties of interest. We believe that the ABAS idea is one promising avenue for helping us to construct component models that facilitate analysis of system properties, and, almost incidentally, define those attributes of components and frameworks that are worth certifying.

Summary

WaterBeans is a custom software component technology that was designed for applications where near real-time performance is needed in pipe-and-filter like applications executing in a uni-processor environment. WaterBeans was demonstrated in three different application domains, ranging from trivial to industrial-strength. There are any number of extensions that can be envisioned that would extend the scope of WaterBeans to distributed applications, heterogeneous systems, and so forth.

Rather than investigate these mechanical extensions we are instead using WaterBeans to investigate how software component technology can provide a foundation for component certification and prediction of application properties from component properties. We are currently investigating how techniques such as ABAS can be used to govern the design of component models, and to identify those properties of components and frameworks that are needed to support the prediction of end-system properties.

References

[Plakosh 99] Daniel Plakosh, Dennis Smith, Kurt Wallnau, Builder's Guide for WaterBeans Components, Technical Report CMU/SEI-99-TR-024, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. http://www.sei.cmu.edu/publications/documents/99.reports/99tr024/99tr024title.html

[Klein 99] Mark Klein, Rick Kazman, Attribute-Based Architectural Styles, Technical Report CMU/SEI-99-TR-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. http://www.sei.cmu.edu/publications/documents/99.reports