WREN—An Environment for Component-Based Development

Chris Lüer David S. Rosenblum
Institute for Software Research and
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
+1-949-824-{2703,6534}
{chl,dsr}@ics.uci.edu

ABSTRACT
Prior research in software environments focused on three important problems—tool integration, artifact management, and process guidance. The context for that research, and hence the orientation of the resulting environments, was a traditional model of development in which an application is developed completely from scratch by a single organization. A notable characteristic of component-based development is its emphasis on integrating independently developed components produced by multiple organizations. Thus, while component-based development can benefit from the capabilities of previous generations of environments, its special nature induces requirements for new capabilities not found in previous environments.

This paper is concerned with the design of *component-based development environments*, or CBDEs. We identify seven important requirements for CBDEs and discuss their rationale, and we describe a prototype environment called WREN that we are building to implement these requirements and to further evaluate and study the role of environment technology in component-based development. Important capabilities of the environment include the ability to locate potential components of interest from component distribution sites, to evaluate the identified components for suitability to an application, to incorporate selected components into application design models, and to physically integrate selected components into the application.

Categories and Subject Descriptors
D.2.6 [**Software Engineering**]: Programming Environments – *Graphical environments, Integrated environments, Interactive environments;* D.2.2 **[Software Engineering]**: Design Tools and Techniques – *Modules and interfaces, Software libraries;* D.2.11 **[Software Engineering]**: Software Architectures – *Information hiding, Languages;* D.2.9 **[Software Engineering]**: Management – *Life cycle, Software configuration management;* D.2.7 **[Software Engineering]**: Distribution, Maintenance and Enhancement; D.2.13 **[Software Engineering]**: Reusable Software – *Reusable libraries, Reuse models;* D.1.5 **[Programming Techniques]**: Object-oriented Programming.

General Terms
Design, Languages.

Keywords
Component-based software engineering, Java, Java Beans, software components, software environments.

ESEC/FSE 2001, Vienna, Austria

1. INTRODUCTION
It has been stated that component technology, while successful in industry, has not received the attention it deserves from the research community [16]. Industrial component models are still rudimentary, and the approaches of different vendors vary strongly. Research is necessary in order to define a common foundation of component technology, and to identify areas in which current standards and tools have to be extended.

Software environments are one area that can benefit especially well from further research. Software development environments (SDEs) were originally designed to integrate collections of tools and to manage locally created development artifacts. Later, process centered software engineering environments (PSEEs) were developed to facilitate the use of well-defined processes to guide development. In order to provide tool integration and process-based guidance for the special needs of component-based development, we envision a new generation of environments, *component-based development environments*, or *CBDEs*. Reusable components developed by and licensed from other organizations cannot be treated in the same way as artifacts that were developed inhouse, since it is usually not possible to change or analyze their implementations. Therefore, new approaches are needed to support identification, retrieval and integration of such components within an environment in an Internet-scalable way.

Szyperski defines a component as follows [32]:

- *A unit of independent deployment*. This means that a key goal of component technology is to facilitate code reuse [14]. A component is a piece of code that has been prepared for reuse. This is opposed to code scavenging, where code that was not explicitly intended to be reusable is being reused. Though initially more expensive, we view design-for-reuse as being the superior approach to enabling reuse.

- *A unit of third-party composition.* Reuse will pay off only when reusing a component that was developed by another organization is significantly easier than redeveloping it. In the ideal case, an application would be composable from components by domain experts without actual programming.
- *Without persistent state.* A component is a piece of code, or a set of abstract data types. In an object-oriented system, a component is a set of classes. A component is not an object or a set of objects.

A CBDE must provide its users with information about components. The users have not designed the components themselves, so they depend on the environment to learn about them. With the use of components, the focus of tools shifts from implementation to design, since the goal of component reuse is to minimize implementation effort. Users must decide which components fit best into their architecture, so the environment should be able to visualize the dependencies among the components. Because components are developed by third parties, the environment should provide the means to access components located at remote sources.

In this paper, we present requirements for CBDEs, and we describe a prototypical environment, WREN, which we are building based on these requirements. Our prototype is based on the Java language and the Java Beans component model. Components packaged as described in this paper are backwards compatible with Java Beans, although they have been extended in various ways. As described in the paper, WREN serves as an early exemplar of this new generation of environments.

2. REQUIREMENTS OF COMPONENT-BASED DEVELOPMENT ENVIRONMENTS

While a number of specialized technologies have been produced in both research and industry to facilitate particular aspects of component programming and reuse, we are unaware of any attempts to provide comprehensive, integrated environment support for the full range of lifecycle activities that must be undertaken in component-based development. In this section we identify seven requirements for CBDEs that address the needs of component-based development. Some of these requirements are addressed by industrial component models, while some of them are not yet widely adopted and are perhaps even controversial. Briefly,

- Accepted rules of *modular design* should be supported explicitly. The environment should support a separation between the private and public parts of a component.
- The environment should support and exploit component *self-description*, meta-information that is stored directly inside of the component. It is used in a limited way in industrial component models like Java Beans and COM.
- Components should be defined and accessed within a *global namespace of interfaces*, which provides a method to name interfaces in a globally (worldwide) unique way. This reduces the problem of semantics matching to one of namespace agreement.
- The environment should support a bipartite development process comprising two parts: *component development* and *application composition*. The former deals more with technical issues of individual component development, while the latter is more application-oriented.

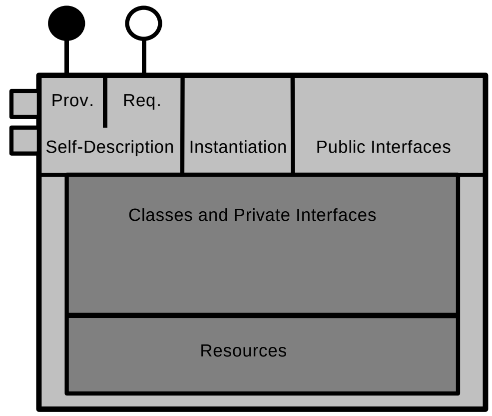

Figure 1. Structure of a Component. Public parts are light gray and private parts are dark gray.

- Application composition consists of configuration of the components and the design and implementation of additional functionality that is not available in components. The environment should support two methods of configuration: *connection* and *adaptation*.
- A CBDE should support *multiple views*, including a development view and a composition view to represent the two halves of the component-oriented process, and a type view and an instance view to show different aspects of the composition view, using an explicit architectural model to represent the overall structure of the application.
- The maintenance problems associated with component technology should be addressed by the environment through *reuse by reference,* which is supported via access to remote, searchable component repositories.

We next discuss the rationale for these requirements.

2.1 Modular Design

Figure 1 presents a generic model of a component that has been prepared for use in a CBDE. A component should be divided into a public part and a private part according to the principle of information hiding or encapsulation [26]. The private part is not accessible from the outside; it contains implementations (in the form of classes) and resources (for example, graphics or help files). The public part contains the self-description of the component, an instantiation mechanism, and optionally public interface definitions. The instantiation mechanism is necessary so that clients can retrieve instances of the data types implemented by the component. To do so, a client specifies only the interface of the data type of which it wants to retrieve a new instance. The decision of which actual class is used to provide this instance is hidden and made by the component itself. Public interface definitions are interfaces that are contained in the component and made accessible to other components, which might want to implement them. The purpose of the self-description and the *provides* and *requires* ports is described below.

The basic unit of syntactical description is the interface. An interface is a named set of operations that describes an abstract data type. Explicit interfaces make it possible to provide alternative implementations (in the form of classes) for a given data type. Thus, if we ensure that components use only interfaces for their specification, the actual implementations will be encapsulated and exchangeable. Interfaces can be specified independently from the components that implement them so that competing manufacturers can offer components that are interchangeable.

2.2 Self-Description

Self-description is a central idea of component technology. Components should be able to provide information about themselves in a systematic way to a CBDE, and to other components at runtime [25]. Description that is contained in the component itself has many advantages over externally stored description. External description, such as documentation stored in text files, can get lost, often has to be updated manually, and cannot easily be queried by development environments. On the other hand, many forms of self-description can be automatically generated and embedded within the component implementation.

The self-description of a component should contain all the information that is needed to reuse it. This is, first, information about the services that the component provides, and second, information about the services the component requires to work [7]. The information in both of these categories can include syntactic, semantic, quality-of-service, and non-technical descriptions [2].

Providing all this information in the component itself instead of in the form of documentation that is stored elsewhere makes the information available to composition tools. A composition tool can check if two components can be connected without having access to their source code, by querying the self-description. In a similar way, component repositories can leverage component self-description for searching and retrieving components. They can check a user's requirements against the self-description. A component self-description standard could reduce the need for a repository standard, because component repositories could then be very simple when all the information about components is stored where it belongs—in the components.

In a similar way, configuration management can be simplified by the use of self-describing components. Typically, configuration management tools store external information about the dependencies between components. This is necessary when arbitrary files are managed. The task becomes easier, however, when the application is built out of self-describing components. Self-description moves dependency information into the components, where it is encapsulated so that it can easily evolve with the evolution of the component implementation.

2.3 Global Namespace of Interfaces

In an ideal situation, component interfaces would be formally specified, and a CBDE would perform formal reasoning to ensure the semantic compatibility of component implementations with their interfaces. However, such reasoning tools are still not widely available or widely used by practitioners, and most commercial components do not have formally specified interfaces. A global namespace of interfaces partly solves the problem of how a CBDE will ensure consistency between the semantics of a provided component and the semantics required of the component; Zaremski and Wing have studied this problem in the context of *signature matching* [36]. While there may be different interfaces providing the same functionality, in a global namespace of interfaces, two interfaces with the same name are intended to be functionally equivalent. On a fundamental level, this greatly simplifies the problem of matching provided components to required semantics, since the problem is reduced to name equality. Only when components do not match at the interface level is human intervention required: Either they are truly incompatible (i.e., incompatible on a semantic level), or the incompatibility is only syntactic, so that they can be matched by simple manual adaptation (for example by wrapping one of them). Of course, mechanisms are still needed to ensure that a component correctly implements the semantics promised by its interfaces, but this problem already existed alongside the component matching problem.

2.4 Component Development and Application Composition Processes

A component-oriented development process looks different from a traditional one. The process is bipartite: The development of components, and the composition of an application from the components, are separated. Typically, the two process parts will be executed by different organizations, the component manufacturer and the organization that wants to license and reuse the manufactured components. We refer to these organizations as the *component developer* and the *application composer*, respectively.

Component development is a traditional development process since all the usual lifecycle phases are traversed. The main difference is that the end product is not a complete application. This means that the product is comparatively small, which may make development processes suited to small projects preferable.

A CBDE can support traditional component development, but it must excel at supporting application composition, which should focus on the business aspects of an application. In the ideal extreme, all components can be *bought* or otherwise obtained, since the goal of component reuse is to minimize the implementation phase of an application. The application composer must select the right components, connect and adapt them, and identify and build components that might be missing. In the near future, it will not be possible to completely eliminate the implementation phase except for trivial projects, but it can be minimized and simplified using appropriate components and environment capabilities.

The application composition process differs from a traditional process in the requirements phase. In requirements, and even more so in design, the component market must be taken into consideration. Finding components that match arbitrary requirements will be difficult or impossible; instead one is forced to select from prepackaged components with given architectural assumptions. The cost savings gained by component reuse will often make it feasible to adapt requirements and design to the components that are available. Thus, the availability of components must be considered during the whole process [21].

2.5 Connection and Adaptation

Once the decision to reuse a certain component is made, it will have to be configured within a CBDE. Component configuration consists of connection and adaptation. Components have to be connected to each other so that they can cooperate. In the simplest case, the connector is just a link between a given required service and a given provided service. In other words, a connector establishes how a requirement is fulfilled. But connectors can be more complex; it is useful to have them encapsulate functionality that logically belongs within a shared infrastructure (for example, communication protocols in a distributed system) rather than to either of the two components that are being connected [31] [6].

Adaptation increases the value of components [3]. The more flexible and adaptable a component is, the more often it will be reused. Ideally, a component will provide ways for application composers to adapt it. However, a component manufacturer will not be able to foresee all adaptations that might be necessary. For this reason, there should be means to adapt a component externally without having to interact with it, for example wrapping.

2.6 Multiple Views

2.6.1 Development View and Composition View

CBDEs should aide both the viewpoint of the component developer and the viewpoint of the application composer. Although a component developer will not necessarily compose any application, the application composer will have to develop some components that are specific to the application being built. So, the application composer may have to switch between both roles.

The *component development view* of a CBDE will look very much like a traditional, non-component-oriented environment. But it should provide a way to distinguish the public features of a component from its internal, private features. In many languages this is done through corresponding keywords. A specific graphic design notation that shows the outside (the specification) versus the inside (the implementation) is helpful. Further, the code for instantiation and syntactic self-description can easily be generated from a graphical representation, such as a UML class diagram.

The *application composition view* will be less traditional. Most importantly, it must abstract from the hidden internals of the components. Even if a component was written by the composer, and so its internals are accessible, the internals should be hidden. Since the purpose of component technology is to minimize implementation effort, the composition view will look very much like a design view.

2.6.2 Instance and Type View

The composition view should be divided into two subviews. The *type view* will show the components that are used and their dependencies. The *instance view* will show selected instances of some of the data types provided by the components, and how they are configured.

Instance views are known from commercial development environments (for example, Web Gain Visual Café, or IBM Visual Age). They allow the composer to visually adapt and connect certain objects (instances of classes), such as GUI elements in dialogs, menus and so on. Graphical instance views save implementation effort by providing a way to specify trivial code in a visual manner. Unfortunately, their applicability is limited. There is no way to specify dynamic behavior in them, such as instantiation. Objects that cannot be created at program initialization, but only later, cannot be represented. For this reason, instance diagrams are best suited to show objects that are singletons, such as unique GUI dialogs, or a database. They are less suited for objects that represent business logic or container data structures.

Type views are on the same logical level as UML class diagrams, but instead of classes, components are shown, and instead of associations or inheritance relations, connectors are shown. The purpose of the type view is to show how the components depend on each other, which components are used in the application, which might be exchanged, and what might be missing. The composer must be able to see what each component provides and requires, for example in order to identify requirements that are not yet met.

The type view shows the architecture of the application that is being composed, and serves as a basis for design decisions. For example, once a need is identified, the composer will have to search in a *component market* for components that fulfill this need. Typically, more than one such component will be available. The composer can use the type view to check which of them best fits into the architecture, and then this can be used as a selection criterion together with aspects like quality of service or price.

2.6.3 Explicit Architectural Diagrams

The relationship between software architecture and component-based development is not well understood yet [29]. UML can be used to some degree to model software architectures, but it currently lacks key facilities needed for this [18] [19]. For instance, UML component diagrams cannot be used to show unmet requirements since they provide no syntactic notation for entities that are required to exist but do not.

For this reason, we propose *provides* and *requires* ports as a diagrammatic notation. The concept of ports is known from, among others, the architecture description language Darwin [15], and they are also used in UML for Real-Time [30]. A port is a part of a component that is expected to be linked to another port with a connector, but is not necessarily connected at all times. Each port is either a *requires* port or a *provides* port, and connectors are directed from *requires* to *provides*, so that they can be interpreted as use-relations. A *requires* port that is not connected shows that something is missing—the component is not yet ready to be used. An application composer can keep track of the completeness of the application that is being built by watching the status of the ports.

2.7 Reuse by Reference

Component reuse exacerbates the problem of maintenance. An application that consists of a large number of independently bought components will be much harder to update than a traditional application built by a single organization, since each component will have individual updates from its manufacturer. *Reuse by reference* is a possible solution to this problem.

Reuse by reference means that a single, worldwide master copy of a component is stored in a component repository and referenced over the Internet. Copying is performed by the CBDE only in the form of caching for performance purposes. A permanent connection is established by the CBDE between the client application that uses the component and the repository on which the master copy resides, so that the component can be updated automatically.

Component repositories should adhere to a standard that makes close integration with CBDEs possible. This standard should support not only referencing of components as described above, but also allow CBDEs to search for components in a flexible and general way.

3. THE WREN ENVIRONMENT

WREN is a prototypical implementation of an integrated CBDE that we are building to realize and evaluate the requirements discussed in Section 2. WREN is integrated with Web Gain Visual Café [34], a software development environment. WREN is a client of one or more component repository servers; we have built such a server, which communicates with WREN through a simple protocol that runs on top of TCP/IP.

In the following, we describe the features of WREN[1], its use for application composition, and how it interacts with the other applications. Support for individual component development is planned, but not yet implemented except as supported in Visual Café.

3.1 Programming Language

We chose Java as the programming language for WREN because it supports component technology and addresses our requirements for CBDEs in multiple ways:

- It supports encapsulation through its access modifiers. Java provides encapsulation on two levels, class and package. Since components can contain more than one class, we use the package-level access modifiers to implement components.
- In Java, signature descriptions can be obtained at runtime through the reflection mechanism of the language. This makes it possible to automatically generate component self-descriptions and simplifies component configuration.
- Java supports interfaces as explicit entities similar to classes. This has the advantage that interfaces and classes can be treated uniformly. A component can provide both classes (i.e. implementations of interfaces) and interfaces.
- Java interfaces reside in a global, worldwide namespace, which is created through the naming convention for package names used in Java: A name should start with the reversed Internet domain name of the manufacturing organization. For example, an interface for abstract data type `foo` developed at the University of California, Irvine, could be named `EDU.uci.foo`.
- Java supports dynamic linking and late binding. This makes it possible to quickly build and evaluate different architectures of a component application.

3.2 WREN Components

WREN requires components to have a specific format and to provide a minimum of self-description. Their architecture is analogous to Figure 1.

A WREN component is a Java archive (jar) file that can contain Java classes, interfaces, and resources. It must contain a self-description class, which provides information about the component. In particular, it has methods that return descriptions of the ports of the component (i.e. which data types are required or provided). The self-description class also has to know which private classes implement the abstract data types that the component provides. Since the data types are identified through interfaces only, other components can receive instances of those data types only through the self-description class (see Section 3.5 below). All classes besides the self-description class should be declared package level private.

WREN components can also contain a diagrammatic representation of themselves, so that they can be represented in component diagrams that are created in the environment (see Section 3.4.3).

3.3 WREN Interfaces

Interfaces are central to the WREN component architecture. A component is characterized by the interfaces of the data types it provides or requires.

WREN interfaces are self-description wrappers around Java interfaces that can provide additional natural language documentation, and additional information about individual operations in the interface. Each operation can have an additional self-description wrapper to provide information about itself. In particular, this wrapper can define post-conditions to assert constraints on the execution of the operation at runtime.

Post-conditions can be a pragmatic solution to the problem of component trust in certain cases [20]. Since components are typically built by another organization and shipped without source code, the application composer needs to trust the component and its developers. Post-conditions can be used efficiently when there is a large difference between the complexity of an operation and the complexity of checking it. An example is prime factorization: while it is complex to find the prime factors of a given integer, it is comparatively easy to check if two numbers are indeed prime factors of another one.

WREN interfaces have version numbers. Versioning is linear; there are no concurrent versions. Higher versions are required to be interface-compatible with all older versions; this means that new versions may add operations, but not remove any. Interface versioning makes component versioning unnecessary, because components are fully specified by the interfaces they provide and require. Thus, two different versions of the same component implementation can simply be treated by WREN as two different components that have the same (or similar) specifications.

3.4 Application Composition Process

Figure 2 summarizes the application composition process that is facilitated by WREN. While the process is not currently enforced in any way, the environment is designed to aid each part of this process. After the requirements are identified, relevant components have to be found. As a first step, repositories should be *searched* in a top-down manner; once the most important components are identified, it will be easier to formulate search criteria for the rest. A typical search will produce far more candidates than needed, many of which will be mismatches. So, in the next step, the composer has to *select* among the found components. All levels of component self-description will be used in this activity. Components that have been selected next need to be *configured* (connected and adapted). Now, missing components, which are required by the selected components, have to be found and integrated, so the process loops back to the search step. Unlike the beginning of the process, where components can be searched for only by vague, natural language criteria, the interfaces specified by the *requires* ports can now be used to automatically search for compatible components. There will still be multiple matches, so that the composer will have to select again according to soft criteria such as quality of service. After several iterations, all components that can be reused will have been found and configured.

[1] Sir Christopher Wren (1632–1723) is remembered for his designs of 51 churches rebuilt in London after the Great Fire of 1666. Each design was unique but was a recognizable variant of an elegant new architectural style.

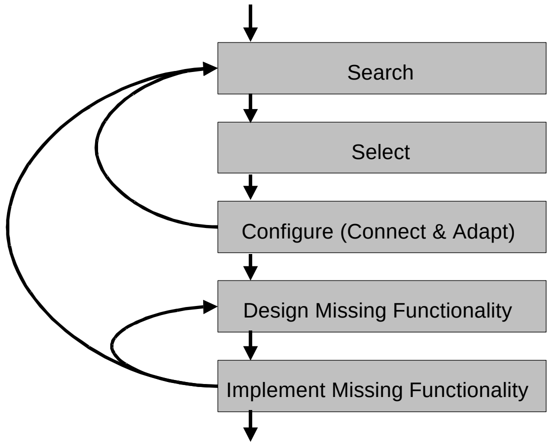

Figure 2. Application Composition Process.

Missing functionality for which no components can be found will have to be *designed* and *implemented* in a traditional manner.

In summary, application composition is an iterative process involving searching, selecting, and configuring components. Searching can be automated in part, but selection and configuration are creative tasks that require design experience.

As shown in Figure 3, WREN supports these activities through repository search and selection views. As a result of the three steps, there are three sets of components that exist during the process. First, there are *available components*, which are all components that match the current search criteria. Out of these, the composer has to select those that are to be used, the *selected components*. Given the set of selected components, the environment can identify *missing components*. These are all the implementations that are required by one of the selected components but not fulfilled by another one. Missing components can only be described in the form of incomplete requirements, since they are not found yet.

3.4.1 *Searching for Components*

Typically, the application composer will start with a broad search using natural-language keywords. The composer enters the search terms into the CBDE, which in turn sends a search command to all the repository servers it knows about.

Search commands are implemented as pieces of mobile code. A repository server executes the mobile code and allows it to search through all its stored components. The mobile code then queries the self-description of the identified components in order to check them against some associated search criteria. The default search command just checks the search terms against a list of keywords provided by the semantic self-description of a component. However, the repository architecture leaves the decision of how to search to the client CBDEs. A CBDE could easily replace this basic search strategy with a more complex one, for example one that makes use of natural language processing features. The use of mobile code for searching the repository makes the repository itself an almost trivial piece of software. All the management of meta-information, dependencies, and so on that is typically done by a reuse repository is delegated to the components themselves, or rather their self-description.

When a component is found that matches the search criteria, a part of the component is transferred to the client. This part contains the self-description information and can handle calls to the implementation part of the component. WREN adds the component to its set of available components (shown in Figure 3), and uses the component self-description to present information about the component to the composer.

3.4.2 *Component Selection*

Often, the set of available components will be very large, since it is difficult to specify search criteria in a sufficiently precise way. The application composer uses the Available Components View to browse through the available components, to look at their properties, and to select the ones that are needed (via check boxes).

As also shown in Figure 3, WREN has a window that displays a selection of relevant properties of the available components for easy comparison. Among them are name, manufacturer, size, price, and number of *provides* and *requires* ports. The numbers of ports allow an easy estimation of the architectural complexity of the component. For example, a component that has zero *requires* ports will be at the bottom of the architecture because it does not depend on any other components. An alternate view of the available components is sorted by the interfaces that the components implement, making it easy to compare all components that are possible suppliers for a given data type. However, since a component usually implements more than one interface, this view is less compact.

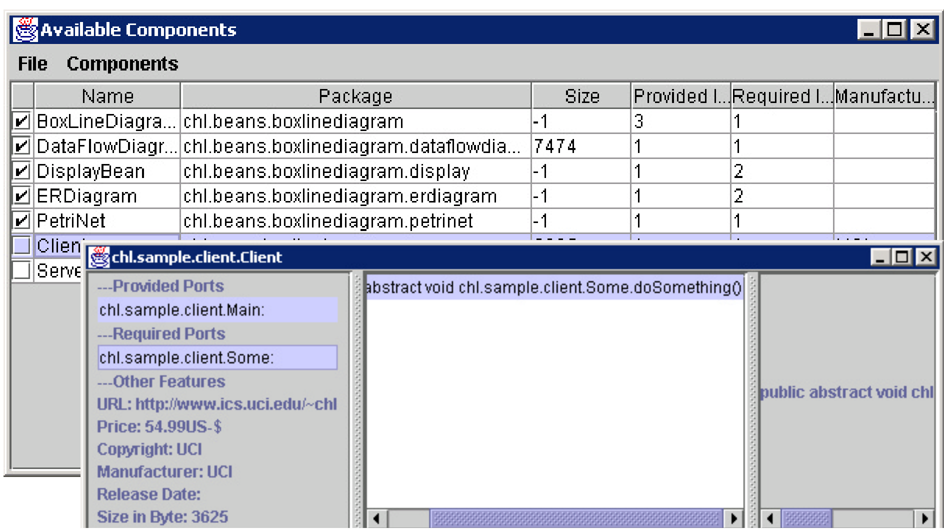

Figure 3. Available Components View and Component Information View. The Available Component View shows if a component is selected, its name, its package name, and selected properties of the component, among them the number of provides ports and of requires ports. The Component Information Window shows all essential information about one component; on the left side general information, in the middle and on the right side the information about the port that is selected.

From the requirements of the selected components, WREN identifies the set of missing components. In particular, it checks through the *requires* ports and adds an entry to the set of missing components for each required data type that is not provided by any of the selected components. It may be possible that several of the missing data types are implemented by one component, so the size of this set does not permit conclusions about the number of actual components that have to be found.

Now, the "find missing components" feature of the environment can be used to automatically search the repositories for all matching components. It is possible that more than one component matches a requirement for a "missing component", so that the composer will have to select among them. The process of searching and selecting components has to be repeated until the set of missing components is empty or the composer decides to reimplement the missing components. To do so, a missing component can be marked as "self-implemented"; this will exclude it from further searches.

3.4.3 *Type-Oriented Component Configuration*

As shown in Figure 4, WREN has a design editor that allows the composer to connect components. The editor is based on Argo/UML [28] [1], an open-source design environment, and it displays UML component diagrams that are augmented by ports as discussed in Section 2. Components selected from a repository are represented in these diagrams by icons provided in the self-description of the components. When the diagram is opened, all selected components are displayed with their respective *requires* and *provides* ports. *Requires* ports are depicted as hollow circles, *provides* ports as filled circles. Each port is labeled with the name of the interface for which an implementation is required or provided. The composer can drag the components and create directed connections in the form of UML dependencies from *requires* ports to matching *provides* ports. Each *provides* port can be used by any number of *requires* ports, but a *requires* port cannot be connected to more than one *provides* port. It is not possible to change the number or names of the ports of a component, since this would require access to its source code. It should be noted that component adaptation as described in Section 2 is not yet implemented.

A component diagram in this style gives an overview of the architecture that is being built and makes it easy to see which requirements are not yet fulfilled. Each unfulfilled requirement corresponds to a *requires* port that is not connected to any *provides* port. Figure 4 provides an example of this with `DisplayBean`'s *requires* port `Printer`. In a similar way, one can see which components may be affected when a component is exchanged for a compatible one.

To conclude type configuration, the composer must specify the main method of the application, i.e. the method with which execution is started. The environment presents a list of all public methods that could be used as main methods, and the composer chooses one of them.

3.4.4 *Instance-Oriented Configuration and Component Deployment*

WREN uses Visual Café for instance-oriented configuration. Visual Café is a commercial Java development environment that supports visual connection and adaptation of Java Beans on an instance basis. When the type-oriented configuration described above is completed, the composer can export the components to

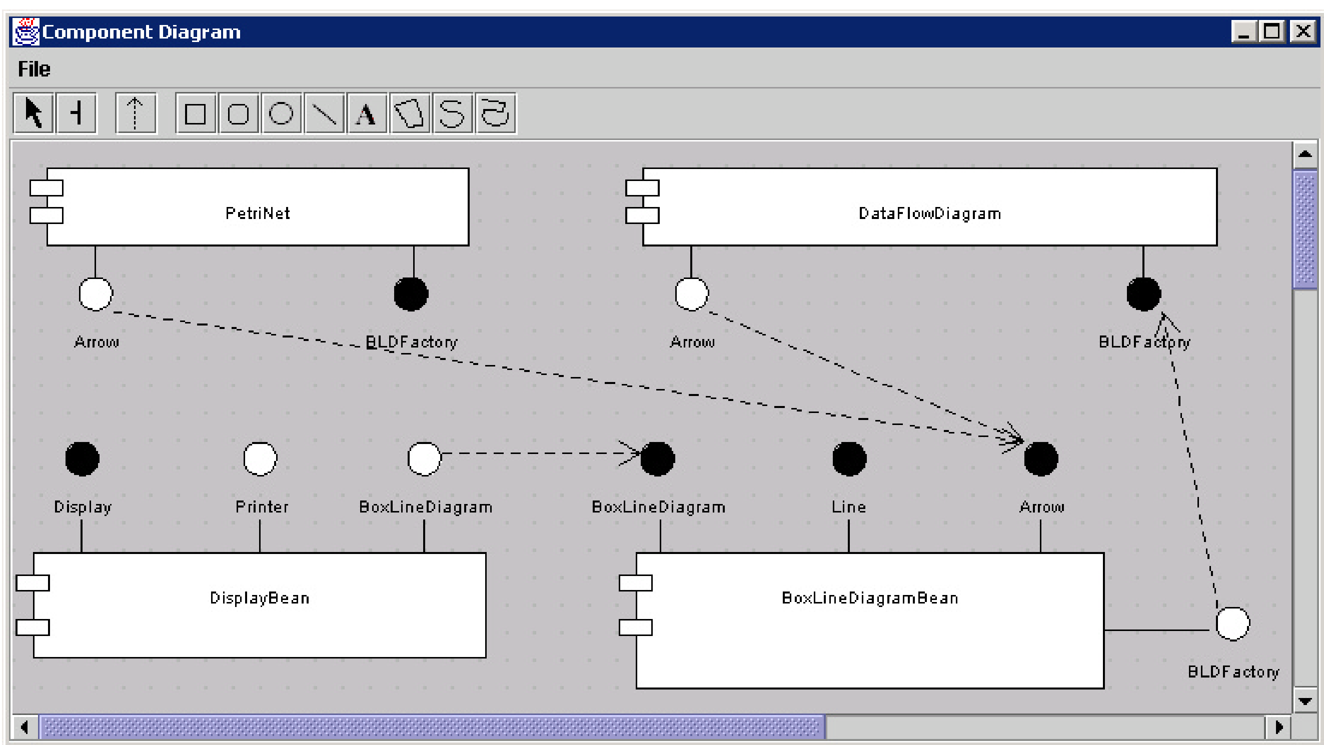

Figure 4. Type-Oriented Component Configuration.

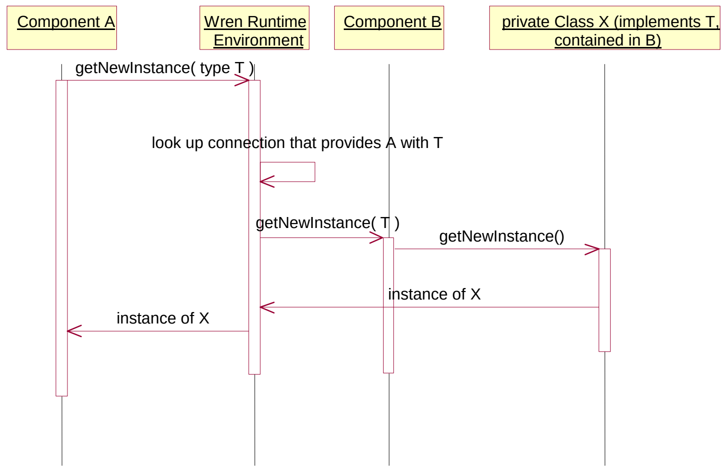

Figure 5. Instantiation across component boundaries in WREN.

Visual Café for instance-level configuration of the application.

WREN uses Remote Method Invocation (RMI) to communicate with a Visual Café plug-in, which automatically loads the components into the component library of Visual Café, from where they can be dragged into Visual Café's visual editor.

3.5 Execution

After the application has been configured, it can be executed. For testing, it can be executed in the WREN environment; this allows an iterative build process of alternating phases of configuring and executing. To make it possible to run the application outside of WREN, the environment can generate a configuration file that stores the names and URLs of the components that participate in this application, and how they are configured (their connections and the name of the main method). The WREN Runtime Environment, a jar file of 25 kilobytes, can then be used to run the application. It will first download the components if they are not yet locally available.

When an application is being executed and a component implementation needs to instantiate a data type that is specified by one of its requires ports (i.e. a data type that is implemented by another component), it will query the runtime environment for a new instance of that data type. Figure 5 depicts this process using a UML sequence diagram. The runtime environment checks the configuration of the application to determine which *provides* port is connected to this *requires* port. Then it asks the component that owns the *provides* port for a new instance, and returns it to the component that needs it. After instantiation, the component uses regular method calls to communicate with the other one; the polymorphism of the programming language makes it possible for one object to use another object without knowing the latter's precise type.

This mechanism constitutes an efficient runtime implementation of the concept of explicit connectors. The connection is explicit, because no component ever knows which of the other components provides the data type that it is using. At the same time it avoids most of the runtime overhead of message passing or similar decoupling strategies. An overhead occurs only when a data type is being instantiated, not each time its instances are used.

3.6 Component Evolution

When a component is marked as selected, the downloaded self-description can be implemented in one of two ways to provide access to the implementation of the component. In the usual case, it downloads a copy of the implementation and caches it locally. Then, it subscribes with the repository for update notifications. When an updated version of the component is published at the repository, the component is notified and can update itself.

The other possible strategy is service reuse [10]. Analogous to a client-server application architecture, the downloaded part of the component forwards requests to the master copy of the component that is located at the repository. Since the component is encapsulated, the difference between the two strategies is transparent to the user of the component, and thus to WREN. This means that the component can decide at runtime which strategy to use. For example, when the network transfer rate is high enough, the most current data can be directly accessed on the remote server. At times when the network is overloaded, the component can decide to use the locally cached version.

Both these strategies realize reuse by reference. In both cases, a logical connection between the application using a component and the original copy of the component is created in order to prevent the maintenance problems associated with reuse.

3.7 Summary

As described above, WREN implements the key requirements that we identified for a CBDE in Section 2:

- It supports modularity by accessing component implementations only through ports.
- It leverages component self-description by extending the design elements of Java Beans.
- It uses the global namespace of the Java language for its interfaces.
- It enables the process of application composition, including searching, selecting, and configuring of components. Component development is not supported, but acceptable support is provided by traditional development environments.
- It facilitates component connection with an easy-to-use graphical design editor. Adaptation is not yet supported.
- It provides multiple views to show the various stages of the application composition process.
- It realizes reuse by reference through integration with a component repository.

Thus, while WREN borrows from a number of isolated component concepts and techniques, we believe that WREN represents the first attempt to systematically support a diverse collection of concepts and techniques in an environment for lifecycle-wide component-based development.

4. RELATED WORK

While CBDEs have yet to become a focus of widespread research, there are several previous research efforts that contribute technologies, principles and insights for CBDE design.

An overview of the history and possible future of software engineering environments is given by Harrison et al. [11]. They consider *multi-view software environments* to be one of the most promising recent trends. Every complex system has many concerns that have to be considered separately. This is best done by providing different, independent views of the various aspects of a system. Type and instance view in WREN are examples of two views that show different aspects of the same system.

The ArchStudio project [17], which evolved out of the Arcadia project [13] and work on the C2 architectural style [33], defines an event-based architecture for a family of software engineering environments. The architectural style used lends itself to distribution, but it is still a subject of current research to determine whether this is possible on an Internet scale. However, integration of WREN with ArchStudio is planned. While tool integration in WREN is currently implemented on an ad-hoc basis, the principled approach of ArchStudio is clearly preferable.

Inscape [27] is an integrated development environment that uses formal module specifications to ensure component compatibility. In a similar way to our approach, it does not perform a complete semantic analysis, but instead leverages a well-designed namespace. However, rather than employing name equivalence over a namespace of interfaces, it employs partial semantic equivalence over a namespace of predicates used to specify interfaces. Inscape relies on Habermann and Perry's concept of well-formed compositions [8]. They list several desirable properties of component-based configurations, most of which are implicitly assured by WREN.

Koala [23] is a component model for embedded software in consumer electronics. It uses an explicit, visual description of architectures based on the architecture description language Darwin [15]. Like Darwin, it has *provides* and *requires* interfaces and treats interfaces as first-class entities. While Darwin was originally geared towards distributed systems, Koala demonstrates the usefulness of these features in a reuse-oriented component model.

The Application Web [22] is a strategy for sharing information between cooperating organizations that tries to minimize the problems caused by copying over organizational borders. To achieve this, connections are created to reuse data. Connections make it possible to automate caching, and to access all (not just part of) the context in which the data were originally created. Connections are comparable to the component references discussed in this paper.

The Basic Interoperability Data Model (BIDM) [4], developed by the Reuse Library Interoperability Group (RIG), is a standard for repositories of reusable artifacts that interoperate. The aim is to provide access to all artifacts offered by a network of repositories through any one repository, thus building a decentralized repository. There are two preconditions for this: There has to be a standard for meta-information about the artifacts, and a way to uniquely identify artifacts. The proposed data model covers some of the aspects we are suggesting for component self-description; however, the information is not stored in the component itself. Uniform Resource Names (URN) are the proposed solution for the identification problem; since a standard for URNs has not been adopted yet, URLs are used. In this way, the naming scheme is effectively equivalent to the naming conventions for Java packages that we rely on.

Whitehead et al. [35] point out that a well-designed architecture is an essential prerequisite for any component marketplace. They identify criteria for such an architecture, the most important of which are realized in WREN as follows:

- *Multiple component granularities* are given in WREN through the possibility to encapsulate any number of classes into a component.
- *Substitutability of components* is realized through the exclusive use of Java interfaces to specify component dependencies. Every interface can be implemented by any number of components, so that every component is substitutable.
- *Easy distribution of components* from seller to buyer is realized by the integration of development environment and component repository.

Brownsword et al. [5] share our view that new processes for developing component-based systems must be defined. Similar to Morisio et al. [21], they stress that the use of licensed components whose source code cannot be modified influences both requirements and design. Since there is a trade-off between the choice of components to license and the requirements and design of the system, these three issues have to be considered simultaneously.

Alpha Services [10] make applications available through the Internet. Instead of downloading and installing a program, services are accessed through the network when needed. This is a kind of reuse by reference; instead of components, services are reused. Candidates for Alpha Services are functionalities that are hard to develop, infrequently used, and can be modeled as transactions, such as natural language translation or large-scale optimization.

The Software Dock [9] is a system supporting the software deployment lifecycle. It integrates producer-side activities such as releasing and retiring a product with consumer-side activities such as installing, updating and uninstalling. Similar to WREN, a permanent connection is established between consumer and producer side. The Software Dock uses SRM [12] to administer the dependencies among application parts, which in WREN are administered by the components themselves. Similar to a CBDE, SRM is geared towards applications made up from independently produced parts.

5. CONCLUSIONS

In this paper we have motivated the need for a new generation of software environments to support the special needs of component-based development. We identified seven important requirements for CBDEs, and we described a prototype environment called WREN that we are building to implement these requirements and to provide a basis for further evaluation and study of the role of environment technology in component-based development.

There are several issues that remain to be resolved. Type-based adaptation does not exist yet in our prototype. Current tools provide mechanisms to adapt component instances, but not components themselves. We expect that the same methods of internal and external adaptation can be used in varied forms for type-based adaptation. Integration with development environments is another issue. It remains to be seen if tight integration of the CBDE with a commercial development environment is the optimal solution, or if an alternative solution is needed.

Updating of components still requires manual effort. While the environment can automatically retrieve updates, it cannot update components that are being used in an application. Doing so will require support for dynamic architecture modification [24]. Another important issue is contract negotiation. A component may be able to dynamically decide about trade-offs between quality of service and price, for example, so that it can negotiate with another component or a human who wants to use this component. Negotiating will require explicit environment support, so that a user can define minimum requirements, policies, and so on.

6. ACKNOWLEDGMENTS

This effort was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061; by the National Science Foundation under grant number CCR-9701973; and by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. The U. S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the Defense Advanced Research Projects Agency, the Air Force Research Laboratory, or the U. S. Government.

7. REFERENCES

[1] Argo/UML. Accessed June 2001 at http://argouml.tigris.org/.

[2] Beugnard, A., Jézéquel, J.-M., Plouzeau, N., and Watkins, D. Making Components Contract Aware. *Computer* 32, 7 (1999), 38-45.

[3] Bosch, J. Adapting Object-Oriented Components. In *Object-Oriented Technology.* Springer, Berlin, 1998, 379-383.

[4] Browne, S. V., and Moore, J. W. Reuse Library Interoperability and the World Wide Web. *Software Engineering Notes* 22, 3 (1997), 182-189.

[5] Brownsword, L., Oberndorf, T., and Sledge, C. A. Developing New Processes for COTS-Based Systems. *IEEE Software* 17, 4 (2000), 48-55.

[6] Dashofy, E. M., Medvidovic, N., and Taylor, R. N. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In *Proc. 1999 International Conference on Software Engineering.* ACM, New York, 1999, 3-12.

[7] DeRemer, F., and Kron, H. H. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Transactions on Software Engineering* 2, 2 (1976), 80-86.

[8] Habermann, A. N., and Perry, D. E. System Composition and Version Control for Ada. In *Software Engineering Environments.* North-Holland, Amsterdam, 1981, 331-343.

[9] Hall, R. S., Heimbigner, D., and Wolf, A. L. A Cooperative Approach to Support Software Deployment Using the Software Dock. In *Proc. 1999 International Conference on Software Engineering.* ACM, New York, 1999, 174-183.

[10] Harrison, C. G., and Stern, E. H. Alpha Services—An Experiment in Developing Enterprise Applications. Accessed June 2001 at http://www.isr.uci.edu/events/twist/twist2000/statements/harrison-stern.doc. 2000.

[11] Harrison, W., Ossher, H., and Tarr, P. Software Engineering Tools and Environments: A Roadmap. In *The Future of Software Engineering.* ACM, New York, 2000, 261-277.

[12] van der Hoek, A., Hall, R. S., Heimbigner, D., and Wolf, A. L. Software Release Management. In *Proc. Sixth European Software Engineering Conference.* Springer, Berlin, 1997, 159-175.

[13] Kadia, R. Issues Encountered in Building a Flexible Software Development Environment—Lessons from the Arcadia Project. *Software Engineering Notes* 17, 5 (1992), 169-180.

[14] Krueger, C. W. Software Reuse. *ACM Computing Surveys* 24, 2 (1992), 131-183.

[15] Magee, J., and Kramer, J. Dynamic Structure in Software Architectures. *Software Engineering Notes* 21, 6 (1996), 3-14.

[16] Maurer, P. M. Components: What If They Gave a Revolution and Nobody Came? *Computer* 33, 6 (2000), 28-34.

[17] Medvidovic, N., Oreizy, P., Taylor, R. N., Khare, R., and Guntersdorfer, M. An Architecture-Centered Approach to Software Environment Integration. Accessed June 2001 at ftp://www.ics.uci.edu/pub/arch/papers/TR-UCI-ICS-00-11.pdf. 2000.

[18] Medvidovic, N., and Rosenblum, D. S. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. In *Software Architecture.* Kluwer, Boston, 1999, 161-182.

[19] Medvidovic, N., Rosenblum, D. S., Robbins, J. E., and Redmiles, D. F. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology,* to appear.

[20] Meyer, B. *Object-Oriented Software Construction.* Prentice Hall, Hemel Hempstead, 1988.

[21] Morisio, M., Seaman, C. B., Parra, A. T., Basili, V. R., Kraft, S. E., and Condon, S. E. Investigating and Improving a COTS-Based Software Development Process. In *Proc. 2000 International Conference on Software Engineering.* ACM, New York, 2000, 32-41.

[22] Murer, T., and Van De Vanter, M. L. Replacing Copies with Connections: Managing Software across the Virtual Organization. In *IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'99).* IEEE Computer Society, Los Alamitos, 1999, 22-29.

[23] van Ommering, R., van der Linden, F., Kramer, J., and Magee, J. The Koala Component Model for Consumer Electronics So ftware. *Computer* 33, 3 (2000), 33-85.

[24] Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., and Wolf, A. L. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems* 14, 3 (1999), 54-62.

[25] Orso, A., Harrold, M. J., and Rosenblum, D. S. Component Metadata for Software Engineering Tasks. In *Proc. 2nd International Workshop on Engineering Distributed Objects (EDO 2000).* Springer, Berlin, 2000, 126-140.

[26] Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 12 (1972), 1053-1058.

[27] Perry, D. E. The Inscape Environment. In *Proc. 11th International Conference on Software Engineering.* IEEE, Washington, 1989, 2-12.

[28] Robbins, J. E., and Redmiles, D. F. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML. *Information and Software Technology* 42, (2000), 79-89.

[29] Rosenblum, D. S., and Natarajan, R. Supporting Architectural Concerns in Component-Interoperability Standards. *IEE Proceedings-Software* 147, 6 (2000), 215-223.

[30] Selic, B., and Rumbaugh, J. Using UML for Modeling Complex Real-Time Systems. Accessed June 2001 at http://www.rational.com/media/whitepapers/umlrt.pdf. 1998.

[31] Shaw, M., and Garlan, D. *Software Architecture.* Prentice Hall, Upper Saddle River, 1996.

[32] Szyperski, C. *Component Software.* ACM, New York, 1997.

[33] Taylor, R. N., Medvidovic, N., Anderson, K. M., Whitehead, E. J., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering* 22, 6 (1996), 390–406.

[34] Visual Café 4. Accessed June 2001 at http://www.webgain.com/products/visual_cafe/.

[35] Whitehead, E. J., Robbins, J. E., Medvidovic, N., and Taylor, R. N. Software Architecture: Foundation of a Software Component Marketplace. In *Proc. First International Workshop on Architectures for Software Systems.* ACM, New York, 1995, 276-282.

[36] Zaremski, A. M., and Wing, J. M. Signature Matching: A Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology* 4, 2 (1995), 146-170.