

Smart Features for Compositional Wrappers

Saleh M. Al-Hatali¹ and Gwendolyn H. Walton²

School of Electrical Engineering & Computer Science
University of Central Florida
P.O. Box 162450, ENGR 407
Orlando, FL 32816-2460
¹sma@bruce.engr.ucf.edu
²gwalton@mail.ucf.edu

Abstract. Component-based software development can reduce software development cost, time, and risks. However, component integration can yield composites that provide additional, undesired functionality. To address this issue, compositional wrappers are often developed to strip off the extra functionality. Additionally, compositional wrappers can implement a number of smart features to support automated development and use of component-based systems. Features of interest include: component and system self-documentation, automated software licensing support, facilities for updating a component and for suspending usage of a particular component, usage and inheritance restrictions, automated installation and customization of components and systems, fault management, and operational support. In this paper, we provide a brief introduction to these features and to some of the issues and opportunities for implementing each feature at the level of compositional wrappers.

1 Introduction

The tremendous explosion in the size and functionality of software applications due to advances in technology, ever-growing user needs, and market trends has often resulted in large, monolithic applications that attempt to meet the needs of many different users. Many in the software engineering community have turned to component-based development to address this need. Unfortunately, during system development, component integrators often require a set of functions that cannot all be satisfied by a single component. We are currently studying the issues related to composing software components using wrappers for the purpose of satisfying the customer's functional requirements of a component. *Compositional wrappers* encase a set of components for which the union of their provided services satisfies the set of functional requirements specified by the client. Compositional wrappers can reduce software development cost, time, and risks. They can provide a comprehensive treatment of component composition, including improved possibilities for automation of component composition.

When a compositional wrapper is deployed with its constituent set of components, the components and the application environment interact with each other through the wrapper's interface. Wrapper generators could be provided to:

- Interact with the client's environment,
- Acquire for the client the set of components that need to be included in the compositional wrapper and
- Provide the “glue” code to integrate the components into a single component (“the wrapper”) that provides the required functionality on the required input domain and nothing else.

A number of “smart features” are desirable to facilitate the component marketplace and support the intelligent use of component-based systems. Smart features include: component and system self-documentation, automated software licensing support (including automated usage tracking), facilities for updating a component and for suspending usage of a particular component, usage and inheritance restrictions, automated installation and customization of components and systems, fault management, and operational support.

Previous discussions of these features in the software engineering published literature typically describe the features in the context of individual components. (See, for example, [1-6]) Orfali, Harkey, and Edwards [7] refer to a “supercomponent” as a component that adds support for: “security, licensing versioning, life-cycle management, support for visual assembly, event notification, configuration and property management, scripting, metadata and introspection, transaction control and locking, persistence, relationships, ease of use, self-testing, semantic messaging, and self-installation.” Others [for example, 8] provide similar descriptions, referring to the enhanced components as “packages”, “megablocks” and “megacomponents”. Building smart features into individual components may yield “fat” components that require very expensive and complex component development efforts and may result in components packed with unnecessary functionality. Large fat components result in slower loading and downloading of components. Systems assembled from these components may be less efficient because each individual component may perform operations that provide no added value to the user. Multiple components may implement the same features, resulting in redundant operations. Because developers often price components based on the amount of functionality provided, the result is often increased costs for systems assembled from feature-loaded components.

It will be difficult to ensure that component vendors adhere to standards supporting implementation, customization, and use of smart features. The result can be less-efficient and/or redundant code to implement each feature. Application assemblers often need to customize component features or provide additional wrappers to ensure compatibility across the component-based system.

To address these difficulties, we propose implementing the features at the level of compositional wrappers, rather than building smart features into individual components. In addition to relieving component developers from the responsibility for designing and implementing these features, this approach can provide a number of potential advantages:

- Opportunities for auto-generation of wrappers can reduce the cost for including smart features.
- Wrapper generators may be able to implement the features in an optimized manner.
- Generated code may be less prone to errors.
- If features are implemented at the wrapper level, there will be fewer instances for implementation of each feature
- It would typically be easier to switch individual features or groups of features on and off, as needed, at the wrapper level than in each individual component. (Vendors may turn features on or off for different reasons, such as when they discover bugs in the implementation of a feature or when they want to upgrade their wrappers with enhanced versions of some features. Users may turn features on or off to improve a system's performance or to protect against or respond to security breaches related to one or more features.)

2 Wrapper Properties

This section provides a brief introduction to some desirable smart features and introduces some issues and opportunities for implementing the features at the compositional wrapper level.

2.1 Self-Documentation

As emphasized by Ruttan [5], component self-documentation is needed to support the evaluation of component usability and stability. Component-based development technologies can support self-documentation services at the component level [9]. For example, COM products require a *QueryInterface* method in each COM component to display the component interfaces and describe how to connect to the interfaces. The JavaBeans *BeanDescriptor* provides a similar mechanism. CORBA provides an object query service that supports locating instances of objects and a trader service that supports locating services by description.

We recommend that compositional wrappers be self-documenting. The wrapper can generate formal or natural language self-documentation upon receiving authorized requests. Documentation may be textual or graphical, depending on the environment settings or the user needs. The documentation can serve as a syntactic and semantic contract [6] between the wrapper's vendor and its current user. At a minimum, generated static documentation should include:

- Identification of the compositional wrapper and its components: the wrapper's name, a brief description of the components it encases and their license information, and the list of interfaces (services and features) provided by the wrapper.

- Iconic and/or GUI representations of the compositional wrapper. Such representations could be made public to allow application assemblers easy handling of assembled wrappers.
- Description of the services implemented by each provided interface and the method signatures for calling the services.
- A list of pre-conditions necessary for compliant operation of the wrapper with its specification, to include environment settings (such as storage quotas and communication protocols) and availability or absence of other interfaces.
- Vendor-supplied estimates of the quality attributes (reliability, survivability, etc.) for each provided interface.
- Self-test and status reporting capability for each service, including a description of the capability and the test suite (inputs, procedures, and expected outputs) used to demonstrate the compliance of the service implementation to its specification.

It should be noted that the proposed compositional wrapper would retrieve most of this information from its individual components through their provided interfaces. Other restricted information, such as the internal data structures of components and the components' source and/or object code could be made available to authorized requesters. Depending on the system requirements, the compositional wrapper could also generate some forms of dynamic documentation such as:

- Representation of the wrapper's current state, including memory usage and active instances.
- Actual values for quality attributes of each service based on recent usage and collected statistics.
- Usage history and timing statistics for each service.

In general, dynamic information would only be disclosed to authorized requesters based on authentication by the compositional wrapper or a higher-order entity.

2.2 Licensing

A license enables an entity that owns property rights (the licensor) to grant to a third party (the licensee) the right to use those property rights. A software component licensor owns copyright, trade secrets, or patent rights in the software component. Most licensed software currently consists of stand-alone applications, and licenses typically apply only to large-grained components, consisting of many lines of code and constituting a significant part of a finished software product [3].

Software components are not intended to, and in most cases do not, function independently as stand-alone applications. Thus, a variety of licensing issues must be addressed with respect to component-based software engineering. When components are used to assemble solutions, licenses must be obtained and maintained for all but freeware components. Some licensed components are designed to actively generate reports at regular intervals to maintain a usage profile or to implement license expira-

tion policies. Component technologies such as CORBA, COM, and JavaBeans have built-in licensing capabilities [6].

There is a variety of software licensing models, each using a different software approach to impose licensing protection on a component [1,10]. However, these licensing models are not directly applicable to all environments; there is limited support for the models within programming languages; and they are applicable only to components in binary form. It can be difficult to use such licensing models with source code or templates. Users can often determine the protection scheme used and obtain unauthorized use. Enforcing these licensing models requires sophisticated techniques and technologies to track privileges and monitor use. To address these issues, we recommend that licensing capabilities be implemented at the level of the compositional wrappers, rather than implemented for individual components.

In addition to providing basic licensing capabilities, the vendor of a component or wrapper may need to be able to suspend or terminate operation, or delete all instances and uses of a component. For example, if the vendor decides to stop supporting a component (perhaps because a replacement component has been made available that uses a different technology, corrects a security risk, etc.), the vendor might want to suspend operation of the old component to force the component clients to replace the old component with the new one. On the other hand, in the case of unauthorized use, the vendor might decide to suspend or terminate the component, or to withdraw the component from all systems suspected of using it. Because this could have negative effect on the system's operations (and, hence, on vendor/client relationships), a protocol for authenticating the legitimacy of these actions must be established in advance. To support selective suspension or termination of components or wrappers from the systems of specific clients requires: the use of unique identifications of components and wrappers; a messaging system to facilitate communication between the vendor and each client; and pre-established protocols for notification, suspension, and termination of components. A compositional wrapper can act very intelligently in these situations, taking responsibility both for authenticating actions and for finding and installing components to replace suspended or withdrawn components.

2.3 State and Structure

Some languages (for example, Java) allow the developer to declare a class to be *final* to prevent further inheritance on the class. We suggest that compositional wrappers be used to manage finality. If a vendor decided to limit extensions of its components to its immediate clients and their immediate clients, the vendor would limit inheritance to level two. The vendor could also selectively enable or disable inheritance based on specific client attributes or based on types of components that can inherit from their components. Such a capability could support improved management of the potentially adverse consequences of inheritance. For example, vendors could allow the use of their components in engineering applications, but not allow their use in mission-critical systems. Also, vendors could allow the use of their components in systems that adhere to certain architectures and standards, and not allow their use in other systems.

Compositional wrappers could enforce certain restrictions on the use and operation of their components. For example, a vendor might require the existence or absence of certain features in the hosting component, system, or environment to ensure that the vendor's component can perform according to its specification. Examples of properties that must exist in the environment to allow components to function properly include: access to certain files or directories, fast Internet connection, use of specific middleware architecture, or existence of certain utilities or features in the host operating system. [2,6,11]

A compositional wrapper could address the issue of a component that functions properly only in the presence or absence of other components, or a component that requires certain operational parameters. (For example, the amount or percentage of its binary image that needs to reside in memory, or the place where the remaining of its binary image is to reside.) Compositional wrappers can also address properties that must *not* exist in the environment due to the conflicts or interferences that such properties could have on the operation of the component. Examples of such properties could include: virus scanners, operating systems that do not support concurrency, or other components that are known to the vendor to cause problems with their component.

While these properties can be implemented in individual components, it is more convenient if they are managed through a higher-order entity, such as a compositional wrapper. If implemented and managed by the compositional wrapper, a property could be turned on or off by the current owners of the encased components. The wrapper could also collect statistics about these properties for the components' owners and/or developers.

2.4 Customization

Ruttan [5] described the need for a property editor to support alterations of the component's state, code, and structure, and hence automate the installation and customization of components. Orfali, Harkey and Edwards [7] included this feature as part of the essential attributes of their "supercomponents." Batory [12] suggested encoding domain-specific intelligence into the wrapper generator so that generated wrappers could perform automatic customization and optimization. Despite the difficulty inherited in the process of encoding component properties and features that can be used in the installation and customization of components, building such features into compositional wrappers can provide greater flexibility to their users. Interactive installation and customization can be costly if implemented at the component level. The compositional wrappers should provide this capability.

If the property editor were integrated with the wrapper generation process, automatic customization and optimization could be performed during the wrapper generation. Such an implementation requires encoding domain-specific intelligence in the wrapper generation process. CORBA provides facilities to encode the domain in which multiple components can be integrated together [6]. However, these facilities do not include the ability to learn about such domains and perform customization and

integration without user intervention. Components need to communicate their installation and customization parameters to the wrapper.

The major problem to be addressed is in retrieving and categorizing the installation and customization properties or features for each component. The wrapper generator must obtain and present to the client (in an accessible form) a list of properties and features from the encased components. A mechanism will be required for internal enumeration and storing the client's responses to the various properties and features.

2.5 Fault Management

The basic process of fault management is: monitor the system's behavior to detect potential failure symptoms as early as possible; diagnose and trace the fault symptom back to the faulty components to identify the failure scenario; and make a decision to ignore the fault or to perform fault correction. An intelligent wrapper should have fault management capabilities. Baggiolini and Harms [11] discussed integrating fault management functionality into generated wrappers. Fault-injection techniques [13, 14] can be used to detect faults and isolate problematic components. These techniques work by injecting "garbage" into the interfaces between components and then observing how the garbage propagates through the system. Such techniques are more suitable to, and more easily implemented in, compositional wrappers.

Segal and Frieder [15] emphasized that any means used for fault monitoring and correction should preserve program correctness and assure no degradation in performance. Cook and Dage [16] encouraged the use of redundant or replicated components to protect against hardware and/or network faults or to provide better service in terms of speed and availability. Component replication can be easily implemented in compositional wrappers.

Execution Traces. It is important that compositional wrappers have built-in mechanisms to capture, store, and re-execute execution traces. Execution traces (also known as execution profiles, execution histories, or execution logs) are the list of states that a component passes through from its initiation to its termination. The values of each variable or parameter for each state are of interest.

There are six types of component traces that can be generated: operation, performance, error, state, GUI events, and communication traces [17]. A component may log its own activities or delegate this task to a service provided by a higher-level component. A usage history report can be generated and sent to the component's owners and/or developers. Such traces can be of great value and can easily be implemented in compositional wrappers.

A scripting architecture that supports execution traces is the Open Scripting Architecture (OSA) that forms part of OpenDoc and, therefore, the OMG's Object Management Architecture (OMA) [6]. To a lesser degree of generality, COM also supports a scripting service, ActiveX Scripting that relies on COM dispatch interfaces. Message Sequence Charts (MSC) [18] can be used to document every use of a component. MSC requires the use of some standard primitives with predefined semantics to record such component usage. The primitives map to the component's API.

Execution traces are important in detecting whether a component passes through any states that violate the list of permissible states outlined in the component's contract or specification. Execution traces are also useful in detecting bugs or failures and can also be used to measure component quality, performance, and reliability. During the component selection and integration stages, execution traces can be used to identify better-performing components to support decisions of whether an old component can be substituted with a newer version [6]. If all traces of all new objects in a component, projected to the states and operations of the old objects, are explicable in terms of the old objects, then the newer version of the component can replace the old one. Recorded execution histories, as in the Multi-Versioning Connector (MVC) [4], can also be used to select compatible components.

Execution profiles are usually very long lists of values in machine- or human-readable formats. Storing and retrieving them can be problematic because of the volume of information to be stored and the performance considerations to be considered when retrieving them, especially if retrieval is done while the system is in operation.

Simulated Execution. For fault identification and correction, the hosting system (the compositional wrapper in our case) may need to go back in history and re-execute a portion of the trace to isolate the buggy component. However, re-execution of a trace often causes problems in other parts of the systems. Thus, facilities for simulated execution may be necessary. (For example, one might simulate executing by logging the states visited without changing system state.) Simulated execution can be of particular value in debugging mission-critical systems for which live testing might be costly or even impossible.

2.6 Operation

Multiple GUI Representations. There are situations when a component needs to provide multiple GUI representations to the same host computer and allow the host computer to switch between them. For example, an application running in a Unix environment may need to provide text and GUI support to its users that run the application directly through the Unix shell or the XWindows environment. While different representations should be supported on the component level, the compositional wrapper can implement the logic necessary to check the environment and select the appropriate representation. Additionally, the wrapper can maintain consistency among supported representations of each component.

Multi-Threading Support. A component should have the ability to execute its methods into separate threads. This capability is available in modern languages such as Java. Presenting multi-threading as a feature that can be switched on or off by the user of a compositional wrapper can add consistency and flexibility to the application.

Built-In Scheduler. There are situations when a wrapper needs to schedule the tasks it performs. For example, many of the smart features discussed in this paper require the component or wrapper to send reports or interact with its owners and/or develop-

ers. Such interactions will normally take a significant amount of time from the system, due to the amount of information to be exchanged between the wrapper and its owners and/or developers. Because this can greatly degrade system throughput, it is often best to schedule such interactions during specified periods of the day and/or week. This requires a scheduler to monitor the schedule and execute tasks at appropriate times. If a scheduler were implemented at the wrapper level, it could schedule tasks for itself and all its components.

Another potential application of the scheduler is to report to other components or wrappers information concerning the periods when the smart wrapper is *active*. (We say that a wrapper is active when it can be called by or considered by other components or wrappers.) Information of interest about each period includes: active start time, active end time, initialization script (to be executed at the beginning of each period), inactivation script (to be executed at the end of each period), and inactive script (to describe how the component or wrapper responds to the outer world when it is not active).

Changing Roles. It may be important to allow a component to change roles. For example, a student taking an instructional course may also serve as a teaching assistant and need to occasionally take the role of an instructor. To support such behavior, a component might need to monitor its own state and react to state changes according to a pre-planned schedule. Implementing such behavior might require the use of Artificial Intelligence or Knowledge-Based techniques. Experts would be required to encode knowledge about the component's operational domain.

If a component is to react to changes in its state, it must be able to react when any attribute is added, modified, or deleted. To support role changing, each attribute might have scripts attached to it: *if-changed* (to be executed when the attribute's value changes); *if-added* (to be executed when the attribute is first created); and *if-removed* (to be executed when the attribute is removed). The *if-added* and *if-removed* scripts may be provided at the wrapper level.

Intelligent Restart. The component or wrapper should be able to store its current state and restore it upon subsequent invocations of the system(s) that are using it. Many technologies (for example, COM's structured storage [6]) provide some means in their architecture for storing and retrieving objects. In addition to this capability, a mechanism is needed for storing the component's state and later allowing the component to resume its execution from where it left.

Sharing Components. A single component may need to act as a stand-alone system, shared among different systems. There are many references and many interpretations for the phrase "deploy once, run anytime." Troya and Vallecillo [19] interpreted the phrase to imply that one loads the component only once and then creates instances of it thereafter. Others [4,16] proposed loading multiple, concurrent copies of the same component to address for component failures. Version management and voting schemes [16] issues must be addressed with respect to this issue.

Self-updating. As noted by Segal and Frieder [15], in order for a system to update its components, the system's logic should be preserved. This statement could also be applied to components and wrappers, requiring components to be updated in the correct order and at the correct time. A smart wrapper could assure such a mechanism by synchronizing the updates at fixed intervals. If a component lacks such a "self-updating" capability, the smart wrapper can acquire updates on the components' behalf.

2.7 Communication

Intelligent Responses. In many situations, a smart wrapper should respond more intelligently than just raising an exception. For example, when another component or wrapper communicate a requests to a smart wrapper, the smart wrapper might respond politely by informing the caller that it cannot provide the requested information. If a directory service were available, the smart wrapper might be able to suggest other components or wrappers as candidates to service the requests. The smart wrapper might also report call histories and events to the owners and/or developers of the wrapper or its components. The owners and/or developers could use this information to support decisions about maintenance and future upgrades of components or wrapper.

The smart wrapper should have some mechanism of conversing with callers and storing information passed from them. Passed information could include the wrapper and callers' domains of expertise and services provided. Other components or wrappers could use such a mechanism to contact a smart wrapper and introduce themselves to the wrapper.

2.8 Performance

Load Balancing. A smart wrapper should have the ability to record performance measures of each of its components (for example, performance per computer, operating system, method, or client.) This could be done by intercepting calls to components and storing timing and callers information. The wrapper can use such information to balance or relocate the load of its components. This can be done by cloning the same component (running multiple instances of it at the same time) [19] or by running two or more "compatible" components [16]. The first option shares the same component binary, but each of its instances runs in its own thread. The second option uses voting schemes to switch to better components according to gathered statistics. Another alternative is the use of the Multi-Versioning Connector (MVC) [4] to allow loading multiple versions of the same component. The multiple versions would be different, but presumably compatible, components, all executing at the same time. MVC monitors the performance, correctness, and reliability of each component. Compatible components can be selected based on the execution history recorded by MVC.

A wrapper may also balance load by splitting heavily used methods into multiple ones. Of course, making this practical would require new mechanisms for specifying method signatures and for implementing the methods themselves.

3. Conclusion

The smart features outlined in this paper can serve two purposes: (a) to raise the level of sophistication of the basic building blocks of future software systems, and (b) to reduce the effort, cost, and schedule required to build such intelligent systems. Compositional wrappers, especially with the help of wrapper generators, are more suitable than individual components for implementation of these smart properties. Intelligent tools and new or enhanced development processes and paradigms will be necessary to achieve the full potential for implementing smart features in compositional wrappers.

References

1. Appleman, D.: Developing COM/ActiveX Components with Visual Basic 6.0 – A Guide to the Perplexed, Sams Publishing (1998)
2. Brown, A.W.: Large-Scale Component-Based Development, Prentice Hall (2000)
3. Chavez, A., Tornabene, C., Wiederhold, G.: Software Component Licensing: A Primer, in IEEE Software, Vol. 15, No. 5, September/October 1998.
4. Rakić, M., Medvidović, N.: Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach, Technical Report USC-CSE-2000-218, University of Southern California, CA, USA (November 2000)
5. Ruttan, P: Components Should Automate their Own Usage. In Proceedings of Workshop on Compositional Software Architectures, Monterey, California, USA (January 2001). Accessed January 2002 at URL: <http://www.objs.com/workshops/ws9801/>
6. Szyperski, C.: Component Software, Addison-Wesley (1998)
7. Orfali, R., Harkey, D., and Edwards, J.: The Essential Distributed Objects Survival Guide, John Wiley & Sons, New York (1996)
8. Succi, G., Pedrycz, W., Liu, E., Yip, J.: Package-Oriented Software Engineering: A Generic Architecture. In IT Professional, Vol. 3, Issue 2 (March/April 2001) 29-36
9. Szyperski, C: Components versus Objects. In Objective View, Issue 5, (2000) 8-16 Accessed January 2002 at URL: <http://www.ratio.co.uk>
10. Simmel, S.S., Godard, I.: Metering and Licensing of Resources: Kala's General Purpose Approach. In Proceedings of the Conference on Technological Strategies for Protecting Intellectual Property in the Networked Multimedia Environment. Accessed January 2002 at URL: <http://www.cni.org/docs/ima.ip-workshop/Simmel.Godard.html>
11. Baggiolini, V., Harms, J.: Toward Automatic, Run-time Fault Management for Component-Based Applications. In Proceedings of the 2nd Workshop on Component-Oriented Programming (WCOP'97), Jyväskylä, Finland (June 1997) 5-12
12. Batory, D.: Intelligent Components and Software Generators. In Proceedings of the Software Quality Institute Symposium on Software Reliability, Austin, TX, USA (April 1997)
13. Kropp, N.P., Koopman, P.J., Siewiorek, D.P.: Automated Robustness Testing of Off-the-Shelf Software Components. In Proceedings of the Twenty-Eighth Annual International

Symposium on Fault-Tolerant Computing, Carnegie Mellon Univ., Pittsburgh, PA, USA, (June 1998) 230-239

14. Voas, J.M.: An Application-Specific Approach for Assessing the Impact of COTS Components. In Proceedings of the 1998 Workshop on Compositional Software Architectures, Monterey, CA, USA, (January 1998) Accessed January 2002 on at URL: <http://www.objs.com/workshops/ws9801/papers/paper002.ps>
15. Segal, M.E., and Frieder, O.: On-The-Fly Program Modification: Systems for Dynamic Updating. In IEEE Software, Vol. 10, Issue 2, (March 1993) 53-65
16. Cook, J.E., Dage, J.A.: Highly Upgrading of Components. In Proceedings of the 1999 International Conference on Software Engineering, Las Cruces, NM, USA (May 1999) 203-212
17. Gao, J: Component Testability and Component Testing Challenges. In Proceedings of the 2000 Workshop on Component-Based Software Engineering, Limerick, Ireland (June 2000)
18. Michiels, B., Wydaeghe, B.: Component Composition. In Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland (June 2000) 771
19. Troya, J.M., Vallecillo, A.: On the Addition of Properties to Components. In Proceedings of the 2nd Workshop on Component-Oriented Programming (WCOP'97), Jyväskylä, Finland (June 1997) 95-104