Generating Configurable Containers for Component-Based Software

Nigamanth Sridhar
Computer and Information Science
The Ohio State University
2015 Neil Ave
Columbus OH 43210, USA
nsridhar@cis.ohio-state.edu

Jason O. Hallstrom
Computer and Information Science
The Ohio State University
2015 Neil Ave
Columbus OH 43210, USA
hallstro@cis.ohio-state.edu

ABSTRACT

Existing container-based development strategies provide solutions to the problem of encapsulating cross-cutting concerns in component-based software systems. These approaches fall short, however, in enabling tractable reasoning. To extend existing work in reasoning about parameterized components to container-based approaches, we view containers as parameterized components. We present a model of component containers based on Service Facilities (Serfs) [18] — a design pattern framework that supports the construction of parameterized components that supports dynamic binding. To ease the transition to this new approach, we present the design of a tool that automatically generates Serf containers for existing component libraries.

Keywords

Automated Software Engineering, Containers, Component Software, Templates, Design Patterns

1. INTRODUCTION

The size of modern software systems is increasing rapidly, with studies suggesting an order of magnitude growth every five to ten years [6, 8]. Development disciplines which accommodate the size of these modern architectures must address a number of issues. Of particular importance, and the focus of this paper, is the adoption of techniques which enable tractable reasoning.

Our inability to do much is well recognized [3]. The techniques used to reason about small programs do not scale in the large. In an effort to keep the *logical size* of our programs as small as possible, software should be built from small, well-understood components that can be assembled predictably. The requirement of predictable assembly imposes an additional constraint on the constituent components: they must be locally certifiable, requiring *all* system dependencies to be at an abstract level [16]. In the absence of this modularity requirement, predictable assembly, and consequently tractable reasoning, is just a pipe dream.

Unfortunately, developing component-based architectures is not an easy task. System design is governed by a (potentially large) number of concerns that should be realized as separate software components [12]. Oftentimes, however, these concerns cannot be encapsulated using traditional modularization techniques. The underlying programming language may lack support for encapsulating particular concerns as reusable modules, or the set of concerns may be conflicting. That is, the component boundaries dictated by one concern may be inconsistent with the boundaries dictated by other concerns that must be accommodated by the same design. Several approaches for overcoming the inadequacies of existing modularization techniques have been proposed:

- **Meta-Level Architectures.** Meta-level architectures like those discussed in [9, 1] expose a meta-level interface for altering the execution of systems built on top of the architecture. These architectures enable the development of meta-level components that encapsulate concerns that are thought of as *cross-cutting*. That is, the meta-level components encapsulate concerns that do not respect the boundaries of the basic decomposition.
- **Aspect-Oriented Programming.** Aspect-oriented programming (AOP) [10] provides another technique for encapsulating cross-cutting concerns. The programming units used to encapsulate these concerns, termed *aspects*, are developed independently of the basic system decomposition. The code contained in these modules is automatically *woven* throughout the implementation of the core system components. AOP has been used to effectively encapsulate a number of cross-cutting concerns, including invocation logging, authentication,

Proceedings of the 6^{th} Component-Based Software Engineering workshop (CBSE6) at ICSE 2003. Portland OR. May 3–4 2003.

synchronization, etc. [2]

- **Multidimensional Separation of Concerns.** Recent efforts investigating multidimensional separation of concerns (MDSOC) [20] have made progress in handling the case of conflicting modularizations. MDSOC deals with the multi-dimensional "hyperspace" of concerns that software must accommodate. Approaches like [11] which afford MDSOC enable software to be developed under multiple system decompositions that can exist simultaneously. Similar to the concept of code weaving, these independently developed system slices are automatically bound into a consistent whole.

- **Component Containers.** Another approach enabling advanced separation of concerns is to include the appropriate implementations as part of the architecture itself. This is the approach taken by *component containers*, like those used to host Enterprise Java Beans [19]. A component container hosts a number of component instances, and provides a logical environment for their execution. Instances hosted by the container are imbued with support for distribution, security, transaction processing, etc., just by virtue of executing within the container. In the commercial software world, the component technology vendor implements the container, and the application developer implements the components to be hosted. This saves substantial development effort on the part of application developers, who are freed from having to implement the services common to all components.

While these approaches have been shown to provide improvements over existing modularization techniques, they fall short in enabling tractable reasoning. In this paper, we focus on component containers as the foundation of a scalable, component-based development discipline that accommodates cross-cutting concerns (Section 3). We present a brief mental model of Service Facilities (Section 2), and provide a way of implementing containers as parameterized components using this approach, and discuss why it enables tractable reasoning (Section 3.1). We then describe a tool that automatically generates service facility container wrappers for existing components (Section 4).

2. SERVICE FACILITIES

The Service Facility (Serf) approach [18] provides a framework for constructing component-based software. The approach includes elements of several popular design patterns [5] such as Abstract Factory, Proxy, Strategy, etc. For a full description of service facilities, their implementation and their advantages, we refer the reader to [18]. Here, we present the key aspects of the approach for the purposes of understanding the rest of this paper.

As with the abstract factory pattern, a service facility provides an abstract interface to clients of a component that can be used for creating instances of the type(s) the component exports. However, while the abstract factory pattern is based on a manufacturing industry metaphor (whereby the factory is out of the picture once the object is created), the service facility approach is based on a service industry metaphor (the service facility always remains as an intermediary between the client and the object instances). Through the lifetime of the program, when the client wants to modify a data object, it invokes the appropriate method on the Serf, with the data object as parameter, rather than invoking the method on the data object directly.

As such, when using the Serf approach, the client is dealing with two kinds of objects at run-time: service facility (Serf) objects and data objects. Each Serf object is responsible for creating certain data objects and then "keeping track of" and "protecting" all those data objects. All methods for manipulating the data objects are supplied by the Serf objects that created them. Data objects have no methods of their own[1].

A Serf object can be viewed as a run-time incarnation of templates in C++ or generics in Ada. They provide an effective way to create parameterized components in languages that do not support generic programming with first class language constructs (e.g., Java and C#). The advantage here is that the binding of template parameters is done at run-time, rather than at compile-time. Such run-time parameter binding incurs a performance penalty when compared to binding at compile-time. However, this is a one-time cost that gives us the benefit of flexibility for the duration of the application's deployment. In long-running systems, this benefit quickly outweighs the cost of instantiation.

"Template" parameters to Serfs are set using the Strategy [5] pattern. The parameters to the template are represented as *strategies*. Each parameter corresponds to a data member in the Serf object, which is set using a method invocation. For every parameter, the Serf exports one method of the form set*Parameter*. The client invokes this method with the appropriate actual parameter. All the parameters for the various Serfs in a hierarchically composed system can be chosen at the highest (client) level, putting the client in complete control of which components it wants to use in the entire system.

This, however, does not mean that the client necessarily controls all the component selection decisions in the whole system. Service facilities also allow for distribution of control as the system designer sees fit. Client programs may choose to use a "partially instantiated" Serf — one in which some, but not all of the parameters have already been fixed. Such partial instantiation decreases the flexibility for a client, but also reduces

[1]In languages such as Java and C#, all object types derive from a root class (Object), these data objects export all methods that they inherit from this root class, but do not define any new methods.

the complexity and "workload" involved in putting the components together. In fact, in languages that support serialization of object instances (such as Java and C#), Serf objects can be created and instantiated with some (or all) parameters, and then serialized, thus creating pre-compiled, partially instantiated templates. These templates can then be used in other programs as is.

A common folk theorem in computer science is that any problem can be solved by introducing an additional level of indirection. Serfs do exactly this. Serfs create a level of indirection between the client and the objects it uses. The creation of objects is decoupled from the client code and placed in the Serf. Further, this decoupling is maintained throughout the lifetime of the objects. This extra level of indirection is what gives the Serf approach its flexibility.

3. CONTAINERS

Containers provide a model of component-based software engineering that supports a clean separation of concerns between core component functionality, and system-level (peripheral) services. The component designer is not concerned with anything other than the core component functionality; all of the peripheral services are encapsulated within the container. Once the base component code has been implemented, the component is registered with the container, and when viewed from outside by a client, is seen as providing the core functionality augmented by system-level services. That is, the client's abstract view of a component P which resides in a container C, includes the effects of the services that C provides to P. So when reasoning about compositional behavior, it is important to consider each component in conjunction with its hosting container.

Problems arise, however, when trying to reason *formally* about component-container compositionality. The idea that the abstract interface of a component changes as a result of inserting the component in a container cannot be accounted for under standard theories of composition. Under various circumstances, we may not be able to predict how the components in a system interact, or how a particular container influences the behavior of the components it hosts.

3.1 Containers as Parameterized Components

Parameterized components provide another way to solve the problem of cross-cutting concerns in component-based software. With this approach, cross-cutting services such as logging, load balancing, etc., are implemented in their own modules, and these modules are passed in as parameters to the component template. The component template can take a variable number of parameters, each of a different type, thereby providing varying services in different instantiations.

One important consideration with such parameterized components that could drastically affect their usefulness is the binding time of template parameters. If the implementation uses compile-time binding, like templates in C++, we are faced with the problem that services are statically bound, and no changes are possible after instantiation. Fortunately, static composition is not inherent to parameterized programming. It is only an artifact of the language mechanisms provided by languages like C++ and Ada. As we have seen in Section 2, Serfs offer a way of programming parameterized components that support dynamic binding.

Several standard proof techniques are available for reasoning about parameterized components [15, 14]. Further, tools exist to automatically generate reasoning tables for parameterized component models that use templates as the parameter binding mechanism [13]. One of the key reasons for the development of Serfs was to make use of these reasoning techniques and automated reasoning methods. Thus, sound modular reasoning and verification of Serfs can be achieved using established techniques. For more details on reasoning about parameterized components constructed using the Serf approach, we refer the reader to [17].

In our model, we view containers as parameterized components. Therefore, containers and components are really the same kind of entities. Removing this distinction allows us to talk about all the different parts of a system using a uniform model. A component that needs to be augmented with peripheral services is encapsulated in a template. The peripheral services are each encapsulated in their own module and supplied to this template as parameters. The completely instantiated template can be seen as a container that contains the component, and extends its functionality by supplying the peripheral services. The primary advantage of adopting this approach is that we can now reason about our components using known techniques, rather than informal appeals to our intuition. Thus, Serf containers provide a means of enabling tractable reasoning, while also providing the flexibility of dynamic composition.

4. AUTOMATED GENERATION OF CONTAINERS

We believe that the Serf approach offers great promise in providing advanced separation of concerns in commercial software systems. We are aware, however, that the software industry has been remarkably slow in integrating the results of academic research, with results taking an average of eighteen years to find their way into standard programming practice [6]. This trend is in large part due to the inability of new development paradigms to coexist with existing technologies. Given the preponderance of legacy systems and the recent flood of commercially available COTS components, systems are increasingly composed, at least in part, of existing technologies. The industry is rightfully unwilling to adopt disciplines that preclude leveraging this substantial investment.

In an effort to hasten the incorporation of Serfs into

the repertoire of standard programming practice, we present a tool to enable the incremental adoption of the Serf approach. The basic approach is to extend the benefits of Service Facilities to existing class libraries by automatically generating Serf wrappers for arbitrary class implementations. While the tool is still under development, we have completed a prototype that illustrates the approach, and serves as a proof of concept for ongoing research. This section describes the use and implementation of the current prototype, and provides pointers to future enhancements.

Given the gaining popularity of Microsoft's .NET Framework, providing an implementation that adapts .NET class libraries seems to be a natural choice. Moreover, the design of the Service Facility Adapter Tool (SFAT) relies heavily on reflection and code generation services, powerful implementations of which are provided as part of the standard .NET class library. As SFAT operates on .NET assemblies, the tool supports the adaptation of any .NET class implementation, regardless of the implementation language. The destination language for the generated wrapper can also, at least conceptually, be any supported .NET language[2]. To illustrate the use and implementation of SFAT, we consider adapting a simple stack implementation, Stack, provided as part of the System.Collections namespace.

SFAT provides a wizard interface, structured as a series of prompts that gather information about the adaptation to be performed. The result of this process is the generation of four primary artifacts that provide the Serf adaptation. SFAT generates an interface corresponding to the Serf wrapper, an interface corresponding to the associated data object, and one class implementation for each of these interfaces. Following standard discipline, the interfaces and implementations are compiled under two separate namespaces. The first prompt collects the names to be used in generating each class, interface, and namespace. In the Stack example, we might choose to name the Serf interface StackSerF, the data object interface Stack, and the class implementations StackSerF_R1 and Stack_R1, respectively. StackSerF_R1 and Stack_R1 will be generated under the ServiceFacility.Stack and ServiceFacility.Stack.R1 namespaces, respectively. Note that both namespaces import the ServiceFacility namespace, which provides interface definitions for ServiceFacility, and Data. The generated Stack.R1 namespace additionally imports the ServiceFacility.Stack namespace.

The second prompt queries the user for the names of the formal parameters to the Serf wrapper. For each of the parameter names provided, a private field will be generated in the Serf wrapper class to maintain a reference to the corresponding Serf for that parameter.

[2]SFAT relies on System.CodeDom.Compiler.ICodeGenerator to generate the source code for the Serf wrapper. Therefore, SFAT supports any destination language for which an implementation of this interface is available.

Public methods will also be generated, both in the interface and implementation, to set a reference to the Serf for each parameter. In the Stack example, we are assuming the contained elements will be of like types, we therefore only provide a single parameter, Item. The result is the generation of a private field within StackSerF_R1 named m_ItemSerf of type ServiceFacility. A public method, setItemSerf will also be generated in both StackSerF and StackSerF_R1, providing a mechanism for setting the value of m_ItemSerf. (Note that more complex classes, like System.Collections.Hashtable, would define two Item types. One item type would correspond to the domain of the hashtable, and the other to the range.). Further, the kinds of parameters are not restricted to just type parameters to specialize collection components. Parameters to templates can encapsulate arbitrarily complex behavior [4].

SFAT next prompts the user to provide the .NET assembly that contains the class to be adapted. Once the assembly has been loaded, SFAT provides a list of the types defined within the specified assembly. The user must then select the class to be adapted from the list of contained types. In the stack example, we would have selected the mscorlib.dll assembly, and selected the Stack type.

In an effort to accommodate a large number of diverse applications, the classes contained in commercial class libraries often contain a large number of methods. Oftentimes these methods are provided for convenience, not completeness. Rather than exposing every method of the class to be adapted through the Serf wrapper, SFAT provides a list of the methods defined in the class to be adapted. Using a simple selection list, the user selects the methods to be exposed through the Serf. In the Stack example, we might choose to select Push, Pop, and Length. Other methods contained in Stack, like Peek, ToArray, etc., will not be adapted, simplifying the interface of the generated wrapper.

Recall that the user previously specified the parameters used by the Serf wrapper. The Serfs corresponding to these parameters are used by the wrapper to operate on method arguments both before and after invoking methods on the underlying adaptee. The methods of the adaptee, however, are not specified in terms of these parameters. A wrapper generated for Hashtable, for example, would have no way of knowing which Serf to use for the domain item or the range item. To address this problem, we ask the user to pick which of the parameters to each method need to be used via Serfs, and which particular Serf to use for each such parameter. The generated code for the method would then use the specified Serf to prepare each of the parameters for the operation. This mapping has to be strict subtype mapping, similar to the one used in the Formal Adapter pattern [7]. In addition, the user also marks the parameter mode of each parameter (consumes, produces, or alters) [4].

The extra level of indirection provided by the Serf wrapper provides a natural point of interception for adding additional peripheral aspects. Like other interception-based approaches, this is achieved by injecting additional code both before and after the invocation on the underlying adaptee. Although quite simple, the approach has been shown to provide a powerful technique for adding peripheral services to existing components in a way that is transparent to the component, as well as the component's clients. Interception has been used to add fault recovery, load-balancing, and a host of other enterprise-critical services.

The final step in the adaptation wizard allows the user to select a number of services to be added to the generated Serf wrapper. SFAT currently supports logging and persistence as services that can woven into the generated wrapper. By selecting logging, for example, the generated wrapper will log all invocations on the Serf to a file specified by the user. Future work aims at developing support for other services that can be automatically integrated in a predictable way. We are currently experimenting with transparent distribution and replication.

5. CONCLUSION

In this paper, we have presented a model of component containers using the Serf approach. The model is based on existing work in parameterized components, allowing us to leverage existing tools when reasoning about container-based software. To ease the transition to a Serf-based development discipline, we have presented the design of a tool used to automatically generate Serf wrappers. In addition to aiding in this transition, the wrappers directly support dynamic binding of services to components, increasing the flexibility of existing container models.

6. ACKNOWLEDGMENTS

This work has been supported by the National Science Foundation under grant CCR-0081596, by Lucent Technologies, and by Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not reflect the views of the National Science Foundation, Lucent Technologies, or Microsoft Research.

We would also like to thank Anish Arora, Paul Sivilotti, and Bruce Weide for their help in writing this paper, as well as the anonymous reviewers for their valuable feedback.

7. REFERENCES

[1] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Proceedings of the IFIP Conference on Dependable Computing for Critical Applications*, Sicily, 1992.

[2] S. Clarke and R. J. Walker. Composition patterns: an approach to designing reusable aspects. In *Proceedings of the 23rd international conference on Software engineering*, pages 5–14. IEEE Computer Society, 2001.

[3] E. W. Dijkstra. Notes on structured programming. In O. Dahl, E. Dijkstra, and C. Hoare, editors, *Structured Programming*, number 8 in A.P.I.C. Studies in Data Processing, chapter 1, pages 1–82. Academic Press, 1971.

[4] S. H. Edwards, W. D. Heym, T. J. Long, M. Sitaraman, and B. W. Weide. Specifying Components in RESOLVE. *Software Engineering Notes*, 19(4):29–39, 1994.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[6] W. W. Gibbs. Software's chronic crisis. *Scientific American (International Edition)*, pages 72–81, Sept. 1994.

[7] J. O. Hallstrom. Formal patterns for design and reasoning in OO systems. URL: http://www.cis.ohio-state.edu/~hallstro/formalpatterns.pdf, February 2002.

[8] W. S. Humphrey. The future of software engineering I. URL: http://interactive.sei.cmu.edu/newssei/columns/watts_-new/2001/1q01/watts-new-1q01.htm, 2001.

[9] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[10] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[11] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, 2001.

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

[13] RSRG. RESOLVE reasoning table generator. URL: http://www.cs.clemson.edu/~resolve/scw/download/rrt-doc.html.

[14] M. Sitaraman, S. Atkinson, G. Kulczycki, B. W. Weide, T. J. Long, P. Bucci, S. M. Pike, W. Heym, and J. E. Hollingsworth. Reasoning about software-component behavior. In W. B. Frakes, editor, *Software reuse: advances in software reusability: 6th international conference,*

ICSR-6, Vienna, Austria, June 27–29, 2000: proceedings, volume 1844 of *Lecture Notes in Computer Science*, pages 266–283, New York, NY, USA, 2000. Springer-Verlag Inc.

[15] M. Sitaraman and B. Weide. Component-based software using RESOLVE. *Software Engineering Notes*, 19(4):21–22, October 1994.

[16] M. Sitaraman and B. W. Weide. Special feature: Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):21–67, 1994.

[17] N. Sridhar. Understanding service facilities. URL: http://www.cis.ohio-state.edu/~nsridhar/Research/usf.pdf, March 2003.

[18] N. Sridhar, B. W. Weide, and P. Bucci. Service facilities: Extending abstract factories to decouple advanced dependencies. In *Proceedings of the 7th International Conference on Software Reuse*, pages 309–326, April 2002.

[19] SunMicrosystems. J2EE 1.3 specification. URL: http://java.sun.com/j2ee/download.html, July 2001.

[20] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. IEEE Computer Society Press, 1999.