Reasoning About Parameterized Components with Dynamic Binding

Nigamanth Sridhar
Computer and Information Science
The Ohio State University
2015 Neil Ave
Columbus OH 43210-1277 USA
nsridhar@cis.ohio-state.edu

Bruce W. Weide
Computer and Information Science
The Ohio State University
2015 Neil Ave
Columbus OH 43210-1277 USA
weide@cis.ohio-state.edu

ABSTRACT

Parameterized components provide an effective way of building scalable, reliable, flexible software. Techniques have been developed for reasoning about parameterized components in such a way that the relevant properties of a parameterized component can be predicted based on the *restrictions* on actual parameters. These techniques assume that the parameters are bound at compile-time. But in some cases, compile-time is just not late enough to instantiate a parameterized component; we would like to push instantiation into run-time instead. Doing this is sometimes dangerous, since we can no longer depend on the type system of the language to support our reasoning methods. In this paper, we present a specification notation and associated proof obligations, which when satisfied, allow us to extend the theories of reasoning about templates with static binding to dynamically-bound templates. We present these proof obligations in the context of the Service Facility pattern, which is a way of building templates whose parameters are dynamically bound.

1. INTRODUCTION

In languages that support them, templates can be used to program parameterized components, which can be specialized to meet specific client needs at component integration time. Further, since the language recognizes templates as first-class constructs, the compiler can enforce type restrictions on them as well as it does on other parts of the language. Reasoning about templates that are instantiated[1] at compile-time is considerably helped by the fact that each of the instantiated templates defines a new type that the compiler recognizes. Further, these types remain static for the rest of the program's lifetime. In general, a compiler that does compile-time template binding (*e.g.,* the C++ compiler) requires the following of the client program that uses a template:

R1. The template is instantiated statically, and

R2. The actual template parameters result in type-correct bodies for the template's methods.

One important consideration with parameterized components that could drastically affect their usefulness is the binding time of template parameters. If parameters are bound at compile-time (as in C++), we are faced with the problem that the component is statically configured, and no changes are possible after instantiation.

Fortunately, static composition is not inherent to parameterized programming [5]. The Service Facility (Serf) design pattern [10] provides a way of building parameterized software components, particularly in languages that do not provide linguistic support for templates. In contrast with C++ templates and Ada generics, which are instantiated at compile-time, template parameters are bound to a Serf at run-time. Such run-time binding has its advantages — the client has more flexibility in pushing design decisions to later in the program's lifetime [11]. However, late binding also has a downside — reasoning about program behavior becomes harder. Since Serf templates are instantiated at run-time, the reasoning system must be strengthened using additional proof obligations that subsume compile-time type checking and related checks, which we outline in this paper.

The rest of this paper is organized as follows. Section 2 reviews the Service Facility pattern, how to build parameterized components using this pattern, and the additional obligations that are needed to reason about the correctness of such components. We conclude in Section 3.

2. THE SERVICE FACILITY PATTERN

The Service Facility (Serf) design pattern [10] is a composite design pattern [8] that combines elements of several well-known design patterns [4], *viz.* Abstract Factory, Proxy, Factory Method, Bridge, and Strategy. Here we only describe the aspects of this pattern that enable parameterized programming. We refer the reader to [10] and to [9] for more details on other aspects of the pattern.

[1] In this paper we use the word *instantiation* to mean the setting of all parameters of a template. We refer to what is often called object instantiation in the OO literature as *object creation* in order to avoid confusion.

Poster presentation at the Workshop on Specification and Verification of Component-Based Systems, co-located with ESEC/FSE 2003. Sep 1–2, 2003. Helsinki, Finland.

Listing 1: C# Stack and StackSerf interfaces

```
1  public interface Stack : Data { }
2
3  public interface StackSerf : ServiceFacility {
4    void push(Stack s, Data x);
5    void pop(Stack s, Data x);
6    int length(Stack s);
7
8    // Template parameter(s)
9    public ServiceFacility ItemSerf { get; set; }
10 }
```

Listing 2: Instantiating StackSerf

```
1 /* ... */
2 PayrollRecordSerf pSerf = new PayrollRecordSerf_R1();
3 /* ... */
4 StackSerf stkSerf = new StackSerf_R1();
5 stkSerf.ItemSerf = pSerf;
6 /* ... */
```

When using the Serf design pattern, a client program dynamically supplies template parameters to the Serf template as *strategies*. Listing 1 shows the C# interface StackSerf, a stack template. A client program using this StackSerf will "instantiate" the template by assigning to the ItemSerf property a Serf isf that will provide the type of the item in the stack (a "strategy"). For example, Listing 2 shows a client instantiating implementation StackSerf_R1 of StackSerf to create a stack of payroll records.

Since the template parameters are set at run-time, there is no way for the compiler to ensure that they are set, let alone in a type-safe way (*i.e.,* with actuals that would have allowed compile-time type-checks to succeed). At the point that the stkSerf object is declared and constructed (line 4 in Listing 2), the compiler decides that the object is ready for use. However, under the semantics of Serfs, this object has not been fully instantiated and is therefore not ready for use. The client, therefore, has proof obligations that it has to satisfy — that the Serf has been properly instantiated with appropriate parameters (line 5).

The template parameters in a Serf are represented as data members in the implementation. Each template parameter corresponds to one (or two in some cases) data member in the Serf class. In order to ensure a Serf object has, in fact, been properly instantiated, the client has to satisfy a proof obligation that all of these data members have legal values. We will see later (Section 2.4) what such legal values are. In the rest of this section, we introduce new notation for specifying template parameters for Serfs.

2.1 Specifying Template Parameters

In order to specify a template Serf, we use the RESOLVE [2] notation. As an example, we present StackContract (borrowed from [2]) specified using the RESOLVE notation in Listing 3. This module defines one type (Stack) and its interface exports three operations on this type — push, pop, and length. The type definition describes a mathematical model (string of Item, in this case), as well as the set of legal values that a new instance of this type can assume upon initialization (empty string, in the case of Stack).

Listing 3: The contract for StackContract specified using RESOLVE

```
1  contract StackContract
2    context
3      global context
4        facility StandardIntegerFacility
5      parametric context
6        type Item
7    interface
8      type Stack is modeled by string of Item
9        exemplar s
10       initialization
11         ensures |s| = 0
12     operation push (
13         alters s: Stack,
14         consumes x: Item
15       )
16       ensures  s = <#x> * #s
17     operation pop (
18         alters s: Stack,
19         produces x: Item
20       )
21       requires |s| > 0
22       ensures  #s = <x> * s
23     operation length (
24         preserves s: Stack
25       ) : Integer
26       ensures  length = |s|
27 end StackContract
```

Each module can export zero or more types. In the case of a module that exports more that one type, the types are identified using a *type identifier*[2].

The *global context* of this contract introduces other modules or facilities that this component *uses*. In this particular example, StackContract makes use of an Integer component, and therefore *imports* the standard realization of that component (StandardIntegerFacility). In programming language terms, the global context serves the same purpose as Java import statements or C# using statements.

The parameters to this template are specified in its *parametric context*. In this example, StackContract is parameterized by the type of item that is contained in a stack. In general, parameters can be of four different kinds: constants, types, facilities, and math definitions. In this paper, we only deal with type and facility parameters, although the ideas can be easily extended to the other types of parameters as well. A *type* parameter lets the client specialize the template by supplying a specific type, as in our current example. A *facility* is an instance of some template. Thus, a facility parameter allows the client set up an integration-time relationship between components. The client can provide realizations of specific contracts that the template can use. Template parameters can also be *restricted* — the actual parameter could be required to implement certain functionality in a valid binding.

2.2 Specifying Serfs in RESOLVE

The contract in Listing 3 specifies that it requires, as part of its global context, the standard integer facility. Recall that a facility is an instance of a template, all of whose

[2]For the sake of simplicity, in this paper we only deal with Serfs that export exactly one type, and so we will no longer refer to the type identifier [10].

Listing 4: The contract for StackSerf

```
contract StackSerfContract
  context
    global context
      service facility StandardIntegerSerf
    parametric context
      service facility ItemSerf
        defining type Item
  interface
    type Stack is modeled by string of Item
      exemplar s
      initialization
        ensures |s| = 0
    operation push (
        alters s: Stack,
        consumes x: Item
      )
      ensures  s = <#x> * #s
    operation pop (
        alters s: Stack,
        produces x: Item
      )
      requires |s| > 0
      ensures  #s = <x> * s
    operation length (
        preserves s: Stack
      ) : Integer
      ensures  length = |s|
end StackSerfContract
```

formal parameters have been bound to actuals. In order to accommodate run-time binding of parameters, we introduce new notation to the RESOLVE language. A *service facility* is an instance of a template that is bound to its parameters dynamically, rather than statically.

Further, we unify all the different kinds of parameters that can be part of the parametric context of a RESOLVE template to be service facilities. In the case of type parameters, for instance, we specify in the parametric context a service facility that *defines* the required type.

Substituting service facilities for facilities, we can translate the RESOLVE StackContract (Listing 3) into StackSerfContract (Listing 4). It is easy to see that the Stack and StackSerf C# interfaces (Listing 1) can be generated from StackSerfContract. All the information needed to generate these interfaces is available in the contract. In general, the type exported by a SerfContract is used to generate the type interface (Stack in the example), and the interface part of the contract, along with the parametric context is used to generate the Serf interface.

2.3 Realizing RESOLVE Contracts as Serfs

Abstract RESOLVE components are expressed in the Serf approach as *interfaces*. In the languages that we consider (Java and .NET languages[3]), interfaces are first-class constructs in the language. Interfaces in these languages are comprised of *method signatures*. There is a direct mapping from the interface in the RESOLVE specification to the programming language interface. Further, the template parameters listed in the concept's parametric context are also represented by methods in the interface. Each facility parameter in the concept corresponds to two methods — one *setter*, and one *getter*[4]. For example, Listing 1 shows the C# interface for a StackSerf component.

2.4 Reasoning About Service Facilities

Now let us see how we can augment Serf interfaces with contract checking in order to enforce the proper use of Serfs as parameterized components. We will handle the two requirements, R1 and R2, separately.

R1. Enforcing Instantiation. Before the Serf can be used to create data objects, we require that the Serf has been properly instantiated, *i.e.,* all the template parameters have been set. For each template parameter that appears in the parametric context of the component, the data members that correspond to that parameter must have been set to values other than their initial values[5].

In order to make sure that by the time we use a Serf it is properly instantiated, we include a check to make sure that all the parameters have actually been set in the pre-condition of each method, including the create method. So, in accordance with design by contract [6], clients that want to use a Serf object have to first instantiate it by supplying appropriate actual parameters.

R2. Enforcing Restrictions. In the foregoing discussion, we have presented one way of making sure that a Serf is actually instantiated before it is used. However, how do we make sure that the parameters that have been supplied are appropriate from the type-checking standpoint?

To a limited extent, we can use the compiler to do these checks for us. In Listing 1, the method used to set ItemSerf takes a parameter of type ServiceFacility. So any legal (according to the C# compiler) invocation of this method should pass in a parameter that implements the ServiceFacility interface. This works, but only as long as we can bundle up all the restrictions on a particular template parameter into a single interface. But this is not possible in most cases. The following example illustrates this further.

Consider a Sort extension to StackSerf, StackSorterSerf. This component creates stacks that can be sorted[6]. For such a Serf, we have two separate restrictions on the ItemSerf parameter. First, this parameter, as in the regular StackSerf, should implement the ServiceFacility interface. Second, data objects created by this ItemSerf must be comparable to each other. That is, we should be able order these data objects according to some policy. Such a policy can be enforced by requiring this parameter to also implement the AreInOrder interface, presented in Listing 5.

A correct ItemSerf parameter to StackSorterSerf must implement both the ServiceFacility and AreInOrder interfaces. A naive way to enforce this using the C# compiler is to make AreInOrder extend ServiceFacility. Then, we can make

[3]All languages that respect the Common Type System of the Microsoft .NET Common Language Runtime have the same set of features [7]. Henceforth, whenever we want to refer to .NET languages, we will use C# as the representative.

[4]Again, we ignore the slight complication of allowing a single component to export multiple types.

[5]In this case, we will just use the initial value conventions of Java/C# — for example, Object type variables are initialized to be null, and int variables are initialized to 0.

[6]We do not care why someone would sort a stack. The purpose of this example is to illustrate problems with checking restrictions on parameters.

Listing 5: AreInOrder interface

```
1 public interface AreInOrder
2 {
3   public bool AreInOrder(Data x1, Data x2);
4 }
```

the type of the ItemSerf property StackSorterSerf to be AreInOrder. This way, the C# compiler could make sure that the parameter ItemSerf actually implements both interfaces. However, this solution is not desirable since in introduces a spurious inheritance relationship between ServiceFacility and AreInOrder where none really exists.

A better solution would be to create a new interface that extends both ServiceFacility and AreInOrder, and change the type of the ItemSerf() property such that it implements this new interface. While this solution works, and does not create bad inheritance hierarchies, it is cumbersome, requiring the creation of too many new interfaces.

Moreover, certain kinds of restrictions are semantic restrictions that cannot be enforced by the compiler. As an example, if a template takes two parameters that have to be related in some way, thee is no way for such a relation to to be encoded syntactically in C# (or Java).

The solution we advocate is again to rely on design by contract. We embed the restrictions on parameters in the specification of the component. From these specifications, we can then create instantiation-checking wrapper components that check whether a particular Serf has, in fact, been instantiated completely. These components are similar in spirit to checking components that ensure that the behavioral contract of the component is respected [3]. The difference here is that we check template instantiation.

Each of the parameters in the parametric context may be annotated with restrictions. For instance, in the StackSorterSerf example, we impose a restriction on the ItemSerf parameter that it implement the AreInOrder interface. This requirement, however, is stated in the RESOLVE contract, but not in the corresponding C# interface for reasons cited earlier in this section. Instead, the requirement is encoded as part of the instantiation-checking component. The setItemSerf method in the instantiation-checking wrapper for StackSorterSerf will now have a precondition that the parameter it gets passed implements the AreInOrder interface.

This precondition can be checked during execution using the Reflection API in C# and Java. The setItemSerf method uses reflection to query the parameter it gets passed to see the list of interfaces that parameter object implements. If AreInOrder is not part of this list, the check fails, and the instantiation does not complete successfully. This failure in instantiation is viewed as a failure to meet the contract, and is handled in the same way as in [1].

Apart from these, there are two more things that need to be checked as well. First, the object that is passed in as a parameter to the setItemSerf method is itself a Serf, and so we need to check if that Serf object has been properly instantiated. In order to perform this check, we include another method in the instantiation-checking wrapper that can be used to query if the Serf that is wrapped in it has been fully instantiated. This method returns a boolean value, after checking (locally) all of the parameters to the Serf. Second, in each method in the Serf, the parameters passed to the method must be checked to see if their runtime types match the expected type. For example, the parameter x to push in StackSerf must be of type Item.

The kinds of assertions that we are dealing with in checking instantiation are all actually checkable at run-time. They do not include arbitrary boolean predicates, but are very constrained — of the form "object implements a given interface", or "object's dynamic type is X", etc.

3. CONCLUSION

In this paper, we have presented a framework for reasoning about parameterized components whose parameters are bound at run-time. We have illustrated the use of this framework in the context of the Service Facility pattern, which is a design pattern that supports the construction of dynamically-bound parameterized components. The components are first specified using RESOLVE, and then realized as Serfs in an implementation language (C# in this paper). Further, we have outlined wrapper components that can check whether a particular Serf component has, in fact, been properly instantiated before it is used.

4. ACKNOWLEDGMENTS

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

5. REFERENCES

[1] S. H. Edwards. Making the case for assertion checking wrappers. In *Proceedings of the RESOLVE Workshop 2002*, number Tech. Report TR-02-11, pages 28–42, Blacksburg, VA, June 2002.

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

[3] S. H. Edwards, G. Shakir, M. Sitaraman, B. W. Weide, and J. Hollingsworth. A framework for detecting interface violations in component-based software. In *Proceedings: Fifth International Conference on Software Reuse*, pages 46–55, 1998.

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

[5] J. Goguen. Parameterized programming. *IEEE TSE*, SE–10(5):528–543, September 1984.

[6] B. Meyer. *Design by contract*, chapter 1. Prentice Hall, 1992.

[7] Microsoft. *Microsoft Visual C# .NET Language Reference*. Microsoft Press, Redmond, WA, 2002.

[8] D. Riehle. Composite design patterns.

[9] N. Sridhar, S. M. Pike, and B. W. Weide. Dynamic module replacement in distributed protocols. In *Proc. ICDCS-2003*, May 2003.

[10] N. Sridhar, B. W. Weide, and P. Bucci. Service facilities: Extending abstract factories to decouple advanced dependencies. In *Proc. ICSR-7*, pages 309–326, April 2002.

[11] H. Thimbleby. Delaying commitment. *IEEE Software*, 5(3):78–86, May/June 1988.