Designing Dynamic Adaptations of CCM Component-based Applications with Roles

Renato Maia and Renato Cerqueira

Pontifícia Universidade Católica do Rio de Janeiro,
Departamento de Informática,
Rua Marquês de São Vicente, 225 RDC,
Gávea, Rio de Janeiro, RJ, Brasil.
{maia,rcerq}@inf.puc-rio.br

Abstract. Roles are been used as abstractions for the design of modularized applications. Some works claim that this abstraction is especially useful in the design of dynamic adaptations. In this work, we put this idea into the test by extending a standardized industrial component model to support the concept of dynamically applied roles. Then, we used this role model to design unanticipated support for communication synchronization, distributed debugging and passive replication that is dynamically introduced into a component-based application. As result, we show that roles have proven an efficient abstraction for proper separation of concerns between the application and all different adaptations applied.

1 Introduction

The necessity for applying changes on an application without interrupting its services leaded to the development of mechanisms for construction of dynamically adaptable software [1, 2]. However, until quite recently, the research on this subject lacked attention on respect to methodologies and abstractions suited for the design of such applications as well as the changes applied to them.

Our research group has been investigating the use of a scripting language as a mechanism to add more flexibility into software systems [3–5], mainly due to the dynamic features and high-level abstraction provided by such languages.

Recent works propose the use of the role concept as a suited abstraction for the design of changes introduced in running applications [6, 7]. In our research, we experimented this idea by introducing the role concept in the CORBA Component Model (CCM). In this paper, we present our implementation of the role concept on the LuaCCM infrastructure, a CCM implementation in the Lua scripting language that provides computational reflection features suited for the implementation of programming abstractions. Additionally, we present three use cases where we successfully used the role concept on a component-based application to design dynamic changes that add support for communication synchronization, distributed debugging and passive replication for fault-tolerance.
2 LuaOrb Adaptation Framework

LuaOrb [3] is a binding of CORBA into the Lua language. It is implemented using the Dynamic Invocation Interface (DII) of some off-the-shelf C++ ORB. Using the flexibility provided by LuaOrb we developed a CCM implementation extended with reflection features as the underpinnings of LOAF (LuaOrb Adaptation Framework), a framework of tools and abstractions for dynamic adaptation of component-based systems. In the remaining of this section we briefly present the LuaCCM infrastructure and the implementation of the role abstraction on top of it.

2.1 LuaCCM

LuaCCM is an implementation of the CCM model using the features provided by LuaOrb. It also extends the CCM model with the concept of dynamic containers that adds support for interception of component invocations.

In the CCM model, a container is a protected execution environment for a component. All interactions of the component with its environment are done through the container. In conventional CCM implementations, part of the container implementation is generated by an IDL compiler accordingly to the component definition and interfaces. In the LuaCCM infrastructure, we provide a dynamic container that is able to adapt itself dynamically in order to comply with some component definition or interface provided at run-time. Additionally, the LuaCCM dynamic container also offers support for changing the component by the addition of new ports, along with its implementation as new component segments, as well as the introduction of interceptors that may change the existing behavior of current component ports.

LuaCCM defines an adaptation interface that is provided by all components as a built-in facet called adaptation. This interface is also supported by all component homes in order to support adaptations over all instances of that home. Therefore, adaptations applied using the facet of a component will enclose only that single component instance. However, if the adaptation is applied using the adaptation interface of a component home, then all the instances of that particular implementation are adapted. Moreover, the LuaCCM container provides an additional operation for retrieval of adaptors for component types. Each component adaptor supports the adaptation interface and it is used to perform adaptation on every single instance of that given component type that resides in the container.

2.2 Roles

Using the reflective features provided by LuaCCM and the extension mechanisms of Lua we implemented the role abstraction in the LOAF based on the work presented in [6]. A LOAF role defines a set of changes on a component by specifying new ports and interceptors on existing ports and can be applied by means of an adaptor object that implements the LuaCCM adaptation interface.
This way, a role may introduce changes in a component instance (through the component adaptor facet), implementation (through the home) or type (through the component type adaptor obtained through the container).

In order to define a new port, the role must specify the kind of the port and the interface associated with the facet or receptacle, or the event transmitted through the event source or sink. Additionally, the role also defines the implementation of facets and event sinks as a chunk of Lua code that creates a new implementation segment for the new port.

Interceptors are objects that are invoked before and after a request is performed on a component port. A LOAF role defines interceptors by providing methods to intercept requests before and after it is dispatched to the component implementation. The interceptor may change the parameters of the invocation or its results. Furthermore, the interceptor may cancel a request.

3 Use Cases

In this section, we present a simple event-based application made of CCM components and then we design three different adaptations using the role abstraction of the LOAF that are applied to this application on-the-fly. This application as well as the adaptations described in this section are based on the examples presented in [6]. The three adaptations described in this section illustrate the use of the main features provided by the LOAF and its underlining infrastructure, the LuaCCM component model.

The application is made of three components: one producer that sends raw data through an event channel and two processors connected to this channel that process the data received. Each time a processor finishes the data processing it sends a request for the next event. These requests contain the number of the event so the producer ignores repeated requests for the same event. This application is essentially abstract; however its model matches many applications, like multimedia streaming, parallel computing, and industrial process control.

Listing 1 shows the IDL definition of this example application. The producer sends data through the produced event source and receives requests through the request event sink. The processors receive data through the raw event sink and send requests through the done event source. Figure 1 illustrates the application using the graphical notation of CCM.

Listing 1. IDL definition for the event-based application.
3.1 Flow Synchronization

This adaptation example introduces a protocol to synchronize the flow of events between producer and processors. The purpose of such protocol is to avoid that some processor get stuck with an amount of event bigger than its processing capabilities. For such, we defined mechanisms for event flow analysis and regulation to be added dynamically to the system by means of dynamic adaptations designed as component roles. The criterion used to define the throughput of each processor is the amount of time taken to process each event. When this throughput goes over some given limit then the necessary event flow regulation mechanism is activated.

Listing 2 illustrates the definition of a role to add throughput analysis support to the processor component. This is done by means of an interceptor on the raw event sink that measures the time between the reception of data events. Additionally, the role also defines a new receptacle to which the interface of the flow regulator mechanism must be attached. This interface is used to set the proper event emission rate. Every time a new event is received the current time is captured so when the event reception finishes, which signals the end of the data processing, the total processing time is calculated. If this time is larger than an upper bound that is set by the additional limit facet then the regulation mechanism provided by the regulator receptacle is used to slow down
the producer. This way, the following events will be emitted at a compatible rate with the processing capabilities of processor.

Listing 2. Role for adding support for flow analysis.

```plaintext
FlowWatcher = l o a f . R o l e { 
  provides = { 
    limit = { 
      interface = "eventflow::flowsync::Limited", 
      code = [[ { value = 0.05 } ]], }, }, 
    uses = { 
      regulator = { 
        interface = "eventflow::flowsync::Rateable", }, 
      }, 
  }, 
  before = { 
    raw = { 
      code = [[function(self, request) 
               request.start_time = get_time() 
               end]], }, }, 
  }, 
}
```

In order to provide the regulator support we define another role as depicted in listing 3. This role adds an interceptor to the produced event source in order to introduce delays in event emission accordingly to a rate defined by the rater facet that is also added by the role. The proper delay is calculated taking in consideration the time of the last emitted event.

Listing 3. Role for adding support for flow regulation.

```plaintext
FlowRegulator = l o a f . R o l e { 
  before = { 
    produced = { 
      code = [[function(self, request) 
                local now = get_time() 
                local last = self.last or now 
                self.last = now 
                local time_spent = now - last 
                if time_spent > limit.value then 
                  local regulator = request.context:get_connection_regulator() 
                  if regulator then 
                    regulator:set_rate(time_spent) 
                    end 
                  end 
                end 
              raw = { 
                code = [[function(self, request) 
                           local now = get_time() 
                           local last = self.last or now 
                           self.last = now 
                           local time_spent = now - last 
                           if request.context.rate then 
                             sleep(request.context.rate - time_spent) 
                             end 
                           end 
             end], }, }, 
  provides = { 
    rater = { 
      interface = "eventflow::flowsync::Rateable", 
      code = [[{self.session_context = function(self, context) 
                     self.context = context 
                    end,
```
Note that besides the assigning roles to components, we also need to combine these changed components in order to achieve the new functionality, in this case, the flow synchronization. This is done by a protocol object that is responsible for implementing the way the roles are applied to components and how they are combined after the changes. Listing 4 shows the code of the protocol object that provides the operation `sync` that implements how the roles are applied and components are combined in order to enable flow synchronization.

### Listing 4. Protocol object used to apply roles and re-combine component.

```lua
FlowSyncProtocol = {
    sync = function(self, server, client)
        FlowRegulator:assign(server)
        FlowWatcher:assign(client)
        client:connect_regulator(server:provide_rater())
    end,
}
```

### 3.2 Distributed Debugging

Besides the addition of new features, a role can also access the component implementation in order to inspect or even change its internal state. This is done by the additional operations of LuaCCM context object `get_executor` and `get_interceptor` that return the implementation or the interceptor of a given component port. For example, consider the definition of role `Inspector` of listing 5. This role adds support for inspecting the internal state of the component by the operation `get_field` provided by the `evaluator` facet. This operation returns the value of a field on the object that implements a particular port specified by the second parameter.

### Listing 5. Role for adding support for inspection of component fields.

```lua
Inspectable = loaf.Role {
    provides = {
        evaluator = {
            interface = "eventflow::distdebug::Inspectable",
            code = [['{
                set_session_context = function(self, context)
                    self.context = context
                end,
                get_field = function(self, field, port)
                    local obj = self.context:get_executor(port)
                    return tostring(obj[field])
                end,
            }]]
        },
    },
}
```

Other important feature for debugging is the insertion of breakpoints to temporarily suspend the execution at specific places. This feature may be included by a role using interception to suspend the execution of components. Listing 6 illustrates a role that is used to add support for a breakpoint on the requests over a port named `port` of a component. The role also defines a receptacle that is used to attach an interface used to check if the breakpoint is still active.
Listing 6. Role for adding support for breakpoint on port port.

```plaintext
1 BreakPoint = loaf.Role {
2   uses = { pauser = { interface = "eventflow::distdebug::Pauser" } },
3   before = {
4     port = {
5       code = [[function (self, request)
6         local pauser = request.context:get_connection_pauser()
7         if pauser then
8           while pauser:locked() do
9           sleep(1)
10         end
11       end
12     end]], }, },
9
Since this role is specifically designed for a given port, we have to define different roles for adding breakpoints on other ports of a component. However, since the roles can be dynamically created, this is not a limitation. Consider the code on listing 7 that illustrates the operation of the distributed debugging protocol object that inserts breakpoints into a CCM component-based application.

Listing 7. Protocol object used to insert breakpoints into component.

```plaintext
1 DebugProtocol = {
2   add_break = function (self, component, port, moment)
3     local break_point = loaf.Role {
4       uses = { pauser = { interface = "eventflow::distdebug::Pauser" } },
5       [moment] = {
6         [port] = {
7           code = [[function (self, request)
8             local pauser = request.context:get_connection_pauser()
9             if pauser then
10               while pauser:locked() do
11               sleep(1)
12             end
13           end
14         end]], }, },
15
9
3.3 Passive Replication

LOAF roles can also be applied to an entire collection of components at once rather than over single component instances. This is done using the adaptation interfaces provided by LuaCCM component homes, used to adapt all instances of a home, and the component type adaptors of LuaCCM containers, used to adapt of instances of a component type. For example, suppose that in an application we have many processor components that receive events from a producer component. Now if we want to add a mechanism for replication of the processing components, we can introduce support for retransmission of received events to a replica. In our example application, this is done with the role defined in listing
```
8 that uses an interceptor to forward replicated events through the additional event source `copied` that must be connected to the component replica.

**Listing 8.** Role for adding support for passive replicator.

```lua
1 Replica = loaf.Role {
2    publishes = { copied = { event = "eventflow::SerialEvent" } },
3    before = {
4        raw = {
5            code = [=[function(self, request)
6                request.context:push_copied(request.params[1])
7            end]]}, }, }, }
```

In order to apply this change over all the component instances of a given type that reside inside a container, we use the adaptor object for that component type that can be retrieved from the LuaCCM container. Listing 9 depicts a script that applies the `Replicator` role over the adaptor object for the `eventflow::Processor` component type. As a result, all the instances of this component type are dynamically changed to support the replication feature and therefore they can be connected to proper replicas that will receive copies of subsequent events.

**Listing 9.** Script to apply `Replicator` role over all `Processor` instances.

```lua
1  −− creating application
2  producer = producerHome:create()
3  processor1 = processorHome:create()
4  processor2 = processorHome:create()
5  producer:subscribe_produced(processor1:get_consumer_raw())
6  producer:subscribe_produced(processor2:get_consumer_raw())
7
8  −− adapting all processor instances
9  adaptor = container:get_component_adaptor("eventflow::Processor")
10 Replica:assign(adaptor)
11
12  −− creating replicas and binding them to changed components
13  replica1 = replicaHome:create()
14  replica2 = replicaHome:create()
15  processor1:subscribe_copied(replica1:get_consumer_raw())
16  processor2:subscribe_copied(replica2:get_consumer_raw())
```

4 Related Work

The notion of roles used in this work is strongly based on the adaptation abstractions developed for the Comet middleware as presented in [6]. Comet is an event-based middleware for component-based distributed applications implemented in the Java language. In this work, we extended the Comet role model to cope with the additional features of the CCM model like interface-oriented ports, component homes, containers, etc. This work also works as a validation of the role concept as a feasible abstraction for the design of dynamic adaptations on complex component models like CCM.

OpenORB [8] is an object request broker implemented on top of the efficient component model OpenCOM that provides reflective mechanisms that may be used to introduce dynamic changes on component-based applications. However,
OpenCOM does not provide proper adaptation abstraction for design of the system changes. Instead, OpenORB adaptations rely on the concept of component frameworks as a design policy to properly isolate different concerns and then make changes over the system more localized and controlled.

DYROBA [7] (Dynamic Role-based Aspect-oriented) approach proposes the use of AOP techniques for the implementation of roles in object-oriented applications. Differently, we use reflection mechanisms as basis for the implementation of dynamic role support. Aside from that, we believe that the use of adaptation abstraction combined with the modularization and organization provided by the software component model provides a good foundation for the design of manageable and maintainable applications.

5 Final Remarks

Development of dynamically adaptable applications raises new challenges when compared to conventional software development. This is especially true when we deal with the design of such application and the changes we intend to apply, which main reason is the tight coupling of all the parts of the system that are merged together at run-time. The use of proper adaptation abstractions provide means to design the application and their adaptations in modularized and organized way.

Generally, the software component model is claimed as a good alternative for the design of modular and maintainable software. Some works [6, 9] extend such model with abstractions for the design of dynamic adaptations that support proper organization and separation of concerns. In this work, we experimented with an extension of the CCM model that provides the role concept as an adaptation abstraction for the design of system changes that are dynamically merged into component-based systems. This experimentation was realized as the design of three unanticipated adaptations totally modularized into different roles that add support for event flow synchronization, distributed debugging and passive replication for fault-tolerance. Even though the role code are very intrusive and depended of the code of the components, the roles provided a feasible mechanism to separate the adaptation from the application.

5.1 Future Work

As future work, we intend to evaluate the role-based approach for designing dynamic adaptation of more complex applications. A future study case for this research is the dynamic introduction of fault-tolerance support by passive replication of components of highly dependable systems, like industrial process controlling.

However, the role abstraction may not be the most appropriate abstraction for every situation. So it is important to evaluate other abstractions for the design of adaptations and compare them with the role-based approach. Therefore, we intend to add abstractions like cross-cuttings aspects [10] and coordination
contracts [9] in the LOAF using the infrastructure provided by LuaCCM and then experiment with these abstractions on the definition of dynamic changes of CCM applications.

Some of the main limitations of the LuaCCM framework rely on the static nature of the underlining C++ ORB used in the implementation of the LuaOrb system. Therefore, we developed the OiL ORB a replacement for it. OiL is an ORB completely written in the scripting language Lua that presents particular characteristics suited for experimentation on middleware development [11]. Now, we are working in the implementation of the CCM support on the OiL ORB to provide features similar to the LuaCCM infrastructure and replace the underpinnings of LOAF.

References