Robust Reconfigurable Erlang Component System

Gabor Batori, Zoltan Theisz, Domonkos Asztalos
Software Engineering Group, Ericsson Hungary Ltd.
H1037 Laborc u. 1. Budapest, Hungary
{Gabor.Batori, Zoltan.Theisz, Domonkos.Asztalos}@ericsson.com

Abstract

In this paper a new robust reconfigurable component system is described which is based on the innovative combination of Erlang/OTP and the concepts of reflective interacting concurrent components.

1. Introduction

Wireless Sensor Networks (WSN) [1] are an intensely researched topic nowadays, however, there is no widely accepted middleware implementation available for application development. The aim of the ongoing RUNES IST [2] project is to create a middleware architecture that can be successfully deployed in heterogenous WSNs. The basis of the middleware architecture consists of a reconfigurable component system and a corresponding Component Run-Time Kernel (CRTK). The concepts of the component system rely on well-known component-based software engineering principles described in [3]. The novelty of our robust reconfigurable component system, ErlCOM, originates from the innovative combination of the beneficial aspects of component-based programming and the merits of the message passing concurrent functional programming paradigm. Moreover, ErlCOM is supported by its concept aware IDE that automatically generates the component architecture in Erlang [4] letting the programmer think more about communicating components and less worry about editing files.

In the remainder of the paper, Section 2 overviews the concept of Concurrency Oriented Programming (COP). In Section 3, we introduce the principles of ErlCOM, then in Section 4 the implementation details are explained. Section 5 describes the ideas behind ErlCOM IDE and finally Section 6 concludes with future work.

2. Concurrency Oriented Programming

COP plays a central role in creating fault tolerant reconfigurable systems by partitioning the complexity of the problem into a number of concurrent processes that interact via message passing. The message passing interfaces between the communicating components are specified by protocols and the messages of the protocols are forwarded via communication channels between the concurrent activities. In accordance with [5], Concurrency Oriented Programming Languages (COPL) support at least the following three-step analysis process:

1. All the truly concurrent activities in our real world should be identified and they should be represented as concurrent *component*s.
2. Communication channels between the concurrent components should be identified and the *message* passing interfaces should be set accordingly.
3. The flows of messages via the communication channels should be investigated and their behavior should be formalized into *protocol*s.

If the component representation is isomorphic to the problem the conceptual gap between the ideal solution of the problem and the implementation of the solution in a particular COPL is minimal, therefore, the reasoning about the implementation details is rooted into the original real world concepts.

Erlang is a COPL developed inside Ericsson and it is widely used in the development of different telecommunications products. As the complexity of real world problems increases above a particular level like in AXD it seems that the isomorphic mapping between the Erlang processes and the real world concurrent activities cannot be sustained if the modularity of the produced code is to be kept intact. ErlCOM tries to maintain that modularity by introducing a component layer on top of Erlang/OTP that satisfies to be regarded as a COPL with some additional abstractions and supporting mechanisms. In other words, it is positioned as a Domain Specific Language (DSL) for robust reconfigurable components written in Erlang.

3. Introduction into ErlCOM

ErlCOM intends to provide a super-structure on top of the well-established Erlang/OTP environment. Since the basic concepts are similar to the ones of Erlang, it seems obvious to explain the ErlCOM component system via analogy.

ErlCOM's components correspond to the simple Erlang processes since they obey to "information sharing via message passing" semantics, however, they are implemented as *gen_server*s to be able to fully utilize the advanced features of OTP. Similarly to the Erlang process Ids each component has a unique name that is registered in the global registry. The components are spread around in caplet hierarchies located inside the Erlang nodes and the caplets provide Erlang-like supervisor facilities to maintain the robustness of the component system. The supervisory decisions are taken according to a set of defined constraints of the particular component framework. Examples of robust auto-configuration are the recreation of crashed components or the migration of a couple of running components due to e.g. load balancing. The interaction between the components is carried out in the form of message passing through bindings representing the behavior of the communication channels. Message passing is synchronous; messages can be intercepted before entering the interfaces of the recipients and after the replies have been exited the same interfaces. The pre- and post-actions of the bindings constitute a list of additional processing on the individual messages. It is important to emphasize that by the introduction of bindings both the concurrent activities represented by the components and the individual message passings represented by the bindings will be reified and be able to be reasoned on. It is in contrast with plain Erlang where processes are first class citizens, but individual messages are not. Bindings are regarded as components and implemented as gen_servers, however, pre- and post-actions do not contain state information, so their implementation relies on Erlang processes. Bindings are created when a receptacle of a particular component is to be bound to an interface of another component and they have been found to be compatible. Both the components and the bindings can contain explicit state information and can be associated with metadata stored in a global repository that is implemented by Mnesia.

Before going into the details of ErlCOM a short summary of its characteristics is worth of being enumerated.

- ErlCOM supports concurrently executing *component*s that can be dynamically *create*d, *load*ed, *update*d, *unload*ed and *destroy*ed.
- Components are managed by *caplet*s that can be thought of as self-contained virtual machines. The root caplet is called *capsule*.
- The components are strongly isolated, that is, they are allowed to interact only by *message*s.
- The message ingresses are called *interface*s and the message egresses are referred to as *receptacle*s, respectively.
- Each component is identified by a unique identifier.
- Message passing is assumed to be *synchronous*, *interceptable* and dynamically modifiable.
- Message passing is realized via updateable *binding* components that connect compatible receptacles and interfaces.
- The components can migrate from caplet to caplet to reconfigure themselves in response to real world events.
- The components build up distributed components frameworks that impose constraints on the participating components and in case of faults the component framework reconfigures itself to maintain the constraints satisfied.
- The components contain *metadata* that enable attribute based component look-up.

4. Detailed description of ErlCOM

4.1. Component

The principal element of ErlCOM is the component that possesses some receptacles and interfaces. All the three ingredients of the component are implemented in the form of *gen_server* structures. Figure 1 shows the Entity Relationship Diagram (ERD) of the component and the related concepts.

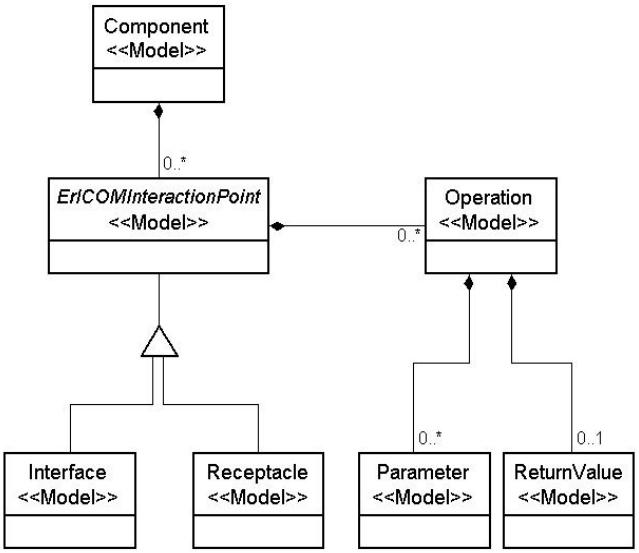

Figure 1. ERD of component and related concepts

Both the interfaces and the receptacles contain a list of operations specified by their signatures and the implementation of the operations is given by ordinary Erlang code. When called the operations automatically get the input parameters bound and they may provide a return parameter in response. What happens in the body of the operations is totally up to the programmers; any kind of Erlang code can be put there. Analogously, the component may contain ordinary Erlang code, too. The Erlang code excerpts implementing a sample component, an interface and a receptacle are shown in Section 4.1.1, Section 4.1.2 and Section 4.1.3 respectively. The code is slightly modified due to better legibility.

4.1.1. Sample Component code

```
-module(e_Component).
-behaviour(gen_server).
-export([init/1, handle_call/3,terminate/2, handle_cast/2,code_change/3,handle_info/2]).
-export([load/1,unload/1]).
-record(stateData,{instanceName,capletName}).

init(Arg) ->
       process_flag(trap_exit,true),
       [InstanceName]=Arg,
       put(instanceName,InstanceName),
       State=#stateData{instanceName=InstanceName},
       {ok,State}.

handle_call(getInterfaces, _Client,State) ->
       InstanceName=State#stateData.instanceName,
       Interfaces=[list_to_atom(atom_to_list(InstanceName)++InterfaceName++"Interface")||
              InterfaceName<-["Interface"|[]]],
       {reply,{result,Interfaces},State};
handle_call(getReceptacles, _Client,State) ->
       Receptacles=[e_ComponentReceptacle],
       {reply,{result,Receptacles},State};

handle_call(AnyMessage,_Client, State)->{reply,ok,State}.

handle_cast(stop, State) ->{stop, normal, State};
```

```
handle_cast(AnyMessage,Req)-> {noreply, Req}.

handle_info(AnyMessage,State)-> {noreply, State}.

code_change(_Vsn, Chs, _Extra) ->
        load_interface(e_ComponentInterface),
        {ok, {Chs, 0}}.

terminate(Reason,_State) ->
        io:format("Component was terminated by the reason: ~w~n",[Reason]).

load(InstanceName)->
        V_type={type,text,[type]},
        MetaDataList=[V_type],
        meta:deleteallprop(InstanceName),
        [meta:putprop(InstanceName,MetaDataType,MetaDataName,MetaDataValue)||
                {MetaDataType,MetaDataName,MetaDataValue}<-MetaDataList],
        %initialize the interfaces and load the meta data of the interfaces
        CapletName = meta:getCaplet(InstanceName),
        I_Interface = list_to_atom(atom_to_list(InstanceName)++" Interface"),
        gen_server:start_link({global,I_Interface},e_ComponentInterface,
                [InstanceName,I_Interface],[]),
        insert_component(I_Interface,interface,InstanceName,CapletName),
        e_ComponentInterface:load(I_Interface),
        %initialize the receptacles and load the meta data of the receptacles
        CapletName = meta:getCaplet(InstanceName),
        R_Receptacle = list_to_atom(atom_to_list(InstanceName)++" Receptacle"),
        gen_server:start_link({global,R_Receptacle},e_ComponentReceptacle,
                [InstanceName,R_Receptacle],[]),
        insert_component(R_Receptacle,receptacle,InstanceName,CapletName),
        e_ComponentReceptacle:load(R_Receptacle).

unload(InstanceName)->
        meta:deleteallprop(InstanceName),
        %destruct the interfaces and delete the meta data of the interfaces
        I_Interface = list_to_atom(atom_to_list(InstanceName)++"Interface"),
        e_ComponentInterface:unload(I_Interface),
        delete_component(I_Interface),
        gen_server:cast({global,I_Interface},stop),
        global:unregister_name(I_Interface),
        R_Receptacle = list_to_atom(atom_to_list(InstanceName)++"Receptacle"),
        e_ComponentReceptacle:unload(R_Receptacle),
        delete_component(R_Receptacle),
        gen_server:cast({global,R_Receptacle},stop),
        global:unregister_name(R_Receptacle).

load_interface(InterfaceName)->
        compile:file(InterfaceName),
        sys:suspend({global,InterfaceName}),
        code:purge(InterfaceName),
        code:load_file(InterfaceName),
        sys:change_code({global,InterfaceName},1,InterfaceName,2),
        sys:resume({global,InterfaceName}).
```

4.1.2. Sample Interface Code

```
-module(e_ComponentInterface).
-behaviour(gen_server).
-export([init/1, handle_call/3,terminate/2, handle_cast/2,code_change/3,handle_info/2]).
-export([load/1,unload/1]).
-record(stateData,{componentName,instanceName,bindingName=undefined}).

init(Arg) ->
        process_flag(trap_exit,true),
        [ComponentName,InstanceName] = Arg,
        put(instanceName,InstanceName),
        State=#stateData{componentName=ComponentName,instanceName=InstanceName},
        {ok,State}.

handle_call({o_operation,[X,Y]},_Client, State)->
        ReturnValue=NEEDED IMPLEMENTATION IN ERLANG
```

```
        {reply,{result,ReturnValue},State};
handle_call(getComponent,_Client,State)->
        Component = State#stateData.componentName,
        {reply,{result,Component},State};
handle_call({refreshBinding,BindingName},_Client,State)->
        OldBindingName=State#stateData.bindingName,
        if
                BindingName /= undefined->link(global:whereis_name(BindingName));
                true->unlink(global:whereis_name(OldBindingName))
        end,
        NewState=State#stateData{bindingName=BindingName},
        {reply,ok,NewState};

handle_call(AnyMessage,_Client, State)->{reply,ok,State}.

handle_cast(stop, State) ->{stop, normal, State};

handle_cast(AnyMessage,Req)-> {noreply, Req}.

handle_info({'EXIT',Pid,noconnection},State)->
        NewState=State#stateData{bindingName=undefined},
        {noreply, NewState};

handle_info(AnyMessage,State)->{noreply, State}.

code_change(_Vsn, Chs, _Extra) ->{ok, {Chs, 0}}.

terminate(Reason,_State) ->
        io:format("ComponentInterface was terminated by the reason: ~w~n",[Reason]).

load(InstanceName)->
        V_key={key,int,1024},
        MetaDataList=[V_key],
        meta:deleteallprop(InstanceName),
        [meta:putprop(InstanceName,MetaDataType,MetaDataName,MetaDataValue)||
                {MetaDataType,MetaDataName,MetaDataValue}<-MetaDataList].

unload(InstanceName)->
        meta:deleteallprop(InstanceName).
```

4.1.3. Sample Receptacle Code

```
-module(e_ComponentReceptacle).
-behaviour(gen_server).
-export([init/1, handle_call/3,terminate/2, handle_cast/2,code_change/3,handle_info/2]).
-export([load/1,unload/1]).
-record(stateData,{componentName,instanceName,bindingName=undefined}).

init(Arg) ->
        process_flag(trap_exit,true),
        [ComponentName,InstanceName] = Arg,
        put(instanceName,InstanceName),
        State=#stateData{componentName=ComponentName,instanceName=InstanceName},
        {ok,State}.

handle_call({operation,Operation,Arg},_Client, State)->
        BindingName = State#stateData.bindingName,
        if
                BindingName /= undefined->
                        case gen_server:call({global,BindingName},{Operation,Arg}) of
                                {result,Result}->
                                        ReturnValue={result,Result};
                                AnyMsg->
                                        ReturnValue={result,AnyMsg}
                                end;
                        true->
                                ReturnValue={result,notconnected}
                end,
        {reply,ReturnValue,State};
handle_call(getComponent,_Client,State)->
        Component = State#stateData.componentName,
        {reply,{result,Component},State};
```

```
handle_call({refreshBinding,BindingName},_Client,State)->
        OldBindingName=State#stateData.bindingName,
        if
                BindingName /= undefined->link(global:whereis_name(BindingName));
                true->unlink(global:whereis_name(OldBindingName))
        end,
        NewState=State#stateData{bindingName=BindingName},
        {reply,ok,NewState};


handle_call(AnyMessage,_Client, State)->{reply,ok,State}.

handle_cast(stop, State) ->{stop, normal, State};

handle_cast(AnyMessage,Req)->{noreply, Req}.

handle_info({'EXIT',Pid,noconnection},State)->
        NewState=State#stateData{bindingName=undefined},
        {noreply, NewState};

handle_info(AnyMessage,State)->{noreply, State}.

code_change(_Vsn, Chs, _Extra) -> {ok, {Chs, 0}}.

terminate(Reason,_State) ->
        io:format("ComponentReceptacle was terminated by the reason: ~w~n",[Reason]).

load(InstanceName)->
        V_type={type,text,[type]},
        MetaDataList=[V_type],
        meta:deleteallprop(InstanceName),
        [meta:putprop(InstanceName,MetaDataType,MetaDataName,MetaDataValue)||
                {MetaDataType,MetaDataName,MetaDataValue}<-MetaDataList].

unload(InstanceName)->
        meta:deleteallprop(InstanceName).
```

4.2. Component Communication

Components communicate via message passing, that is, messages are sent from the receptacle of a component to the interface of the other component through the binding. The binding contains a list of pre- and post-actions that are activated every time a message has reached the binding. Both the pre- and post-actions are implemented as processes and they are activated one by one as the message passes through the binding. The behavior of the binding is shown in Figure 2 and the implementation is given in Section 4.2.1. The code is slightly modified due to better legibility.

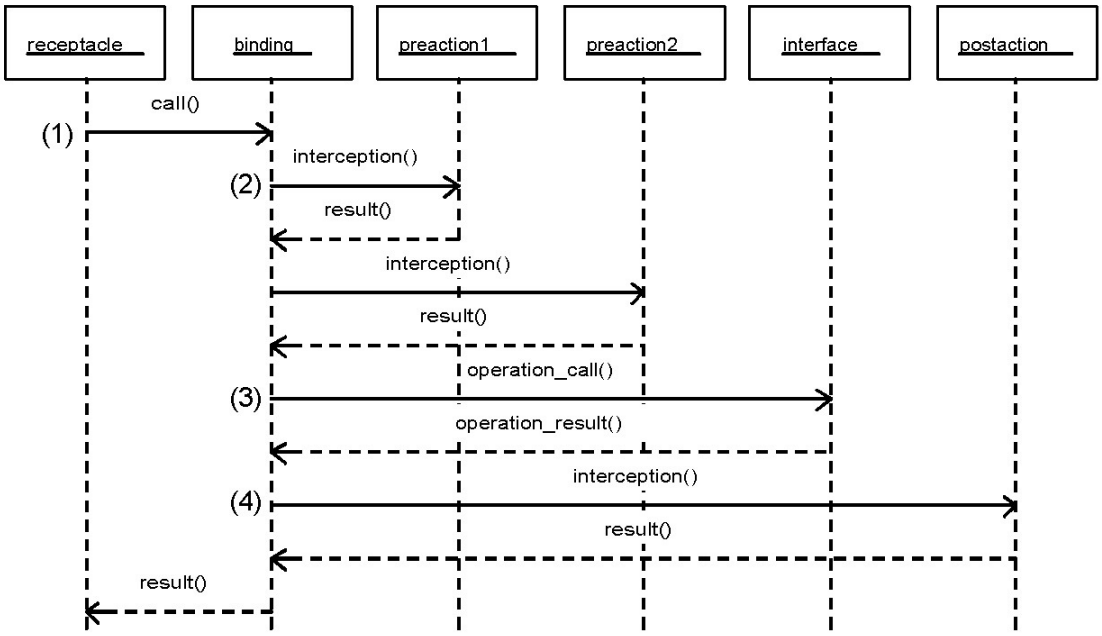

Figure 2. Component Communication

4.2.1. Component Communication Code

```
%Receptacle code:
(1) handle_call({operation,Operation,Arg},_Client, State)->
        BindingName = State#stateData.bindingName,
        if
                BindingName /= undefined->
                        case gen_server:call({global,BindingName},{Operation,Arg}) of
                                {result,Result}->
                                        ReturnValue={result,Result};
                                AnyMsg->
                                        ReturnValue={result,AnyMsg}
                                end;
                        true->
                                ReturnValue={result,notconnected}
                end,
        {reply,ReturnValue,State};

%Binding behavior:
handle_call({Operation,Arg},_Client, State)->
        InterfaceName=State#stateData.interfaceName,
        PreActionList=State#stateData.preActionList,
        PostActionList=State#stateData.postActionList,
(2)     case interception(PreActionList,{Operation,Arg}) of
                {NewOperation,NewArg}->
(3)             case gen_server:call({global,InterfaceName},{NewOperation,NewArg}) of
                        {result,Result}->
(4)                             {PostActionResult}=interception(PostActionList,{Result}),
                                ReturnValue={result,PostActionResult};
```

```
                        AnyMsg->ReturnValue={error,AnyMsg},
                end;
                AnyMsg->
                        ReturnValue={error,AnyMsg},
        end,
        {reply,ReturnValue,State};

%Interception:
(2) interception([{_,Pid}|T],Msg)->
        Pid!{self(),Msg},
        receive
                NewMsg->
                case NewMsg of
                        {result,ReturnValue}->interception(T,ReturnValue);
                        AnyMsg->{error,AnyMsg}
                end
        end;
interception([],Msg)->
        Msg.
```

4.3. Component Run-Time Kernel

The CRTK provides the API that specifies and implements the operations according to the principles described in Section 3. It is very important from the point of view of the components that the CRTK is available ubiquitously, in other words, it is floating above the available resources. Erlang/OTP offers an elegant solution by combining a particular module (*crtk* in our case) with Mnesia so that the global repository is accessible from anywhere. Taking into consideration the performance aspects some maintenance data related to the caplet provided robustness of the components are stored in ETS tables. Each caplet maintains its local ETS database where data about its currently contained components are stored. Those data enable the caplet to provide supervisory activities on the components. In addition, Mnesia tables empower ErlCOM to feature distributed behavior and the storage of component metadata facilitates reflective operations. *Reflectivity* is the key concept of the adaptive behavior of the components as the components can consult the global repository to fetch information on any other components and their metadata and to interact with them properly.

In the following the CRTK API will be explained. The behaviors will be given in the form of Entity Relationship Diagrams (ERD) and the behaviors are presented in the form of Sequence Diagrams (SD). The code excerpts represent slightly modified code due to better legibility.

4.3.1. Component Operations

The CRTK API provides five operations regarding the life-cycle components. They are the following:

- **create**: create a new component in a caplet (Section 4.3.1.1)
- **load**: load the code of the component (Section 4.3.1.2)
- **update**: update the code of the component (Section 4.3.1.5)
- **unload**: unload the code of the component (Section 4.3.1.3)
- **destroy**: destroy the component in the caplet (Section 4.3.1.4)

In the following sections the details of the above mentioned operations will be shown.

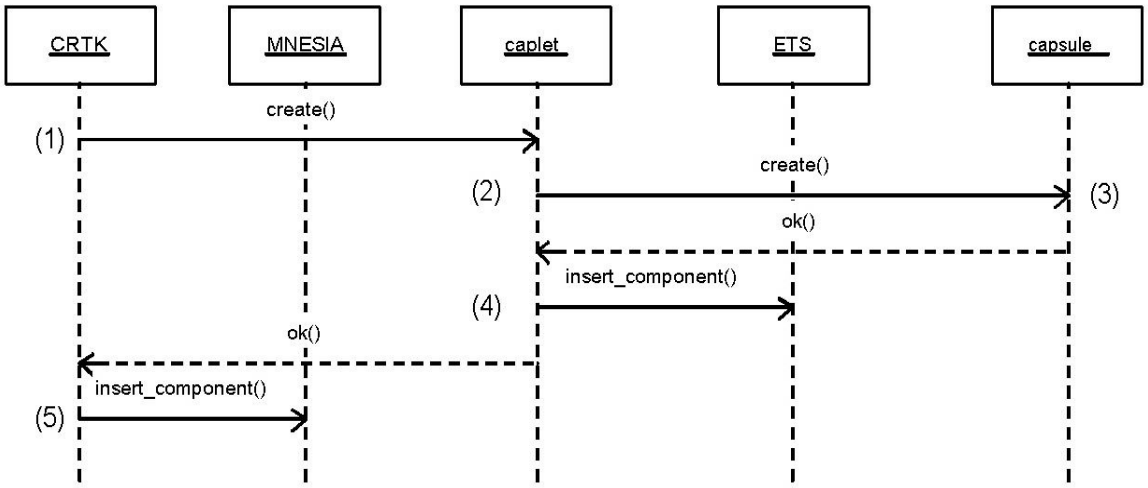

Figure 3. Create Component

4.3.1.1. Create Component Code

```
%create in CRTK
(1) create(CapletName,InstanceName)->
        gen_server:call({global, CapletName }, {create, InstanceName }),
(5)     insert_component(InstanceName,component,CapletName,CapletName).

%create in Caplet
(2) create(InstanceName,Type)->
        CapsuleName = crtk:getOwner(crtk:getSelfName()),
        gen_server:call({global,CapsuleName}, {create,InstanceName}),
(4)     insert_component(InstanceName,Type).

%create in Capsule
(3) create(InstanceName)->
        gen_server:start_link({global,ComponentName},e_EmptyComp, [InstanceName],
                []).

%insert_component in Caplet
(4) insert_component(InstanceName,Type)->
        ets:insert(get(componentTable),#component{componentName=InstanceName,
                componentData=#componentData{componentType=Type,state=created}}).

%insert_component in CRTK
(5) insert_component(ComponentName,ComponentType,Owner,RegistryOwner) ->
        NodeName=node(),
        Fun = fun() ->
        mnesia:write(#component{componentName=ComponentName,componentType=ComponentType,
                owner=Owner,registryOwner=RegistryOwner,nodeName=NodeName})
        end,
        mnesia:transaction(Fun).
```

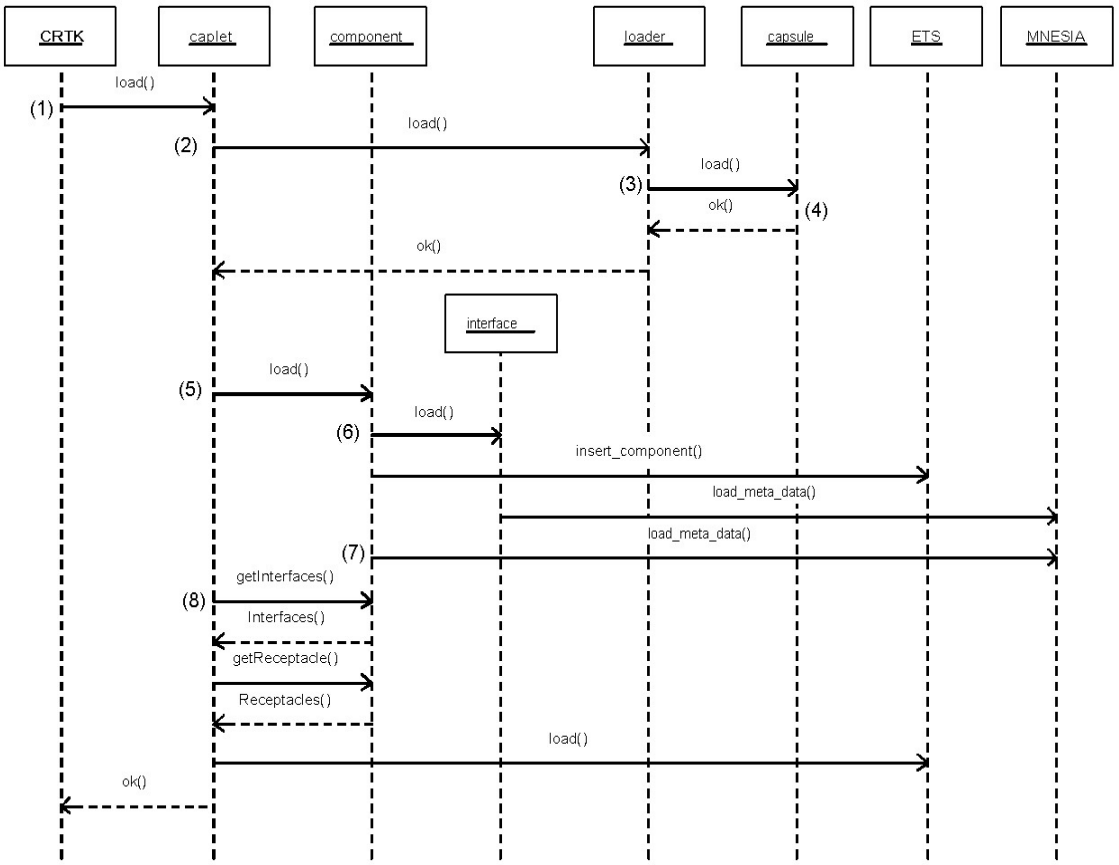

Figure 4. Load Component

4.3.1.2. Load Component Code

```
%load in CRTK
(1) load(LoaderName,ComponentType,InstanceName)->
        CapletName = getRegistryOwner(InstanceName),
        State=getState(InstanceName),
        if
                State == created->
                        gen_server:call({global,CapletName},
                                {load,LoaderName,ComponentType,InstanceName});
                true->
                        io:format("~w was loaded previously.!!!~n",[InstanceName]);
        end.

%load in Caplet
(2) load(LoaderName,ComponentType,InstanceName)->
        State=getState(InstanceName),
        gen_server:call({global,LoaderName}, {load,ComponentType,InstanceName,State}),
        {result,Interfaces}=gen_server:call({global,InstanceName},getInterfaces),
        {result,Receptacles}=gen_server:call({global,InstanceName}, getReceptacles),
(5)     ComponentType:load(InstanceName).
(8)     %get the interfaces and receptacles of the component and register the componenent
        %in the ETS
        ProvidedInterfacesList=[#interface{interfaceName=InterfaceName}||
                InterfaceName<-ProvidedInterfaces],
        RequestedInterfacesList=[#interface{interfaceName=InterfaceName}||
                InterfaceName<-RequestedInterfaces],
        load(ComponentType,InstanceName,component,LoaderName,ProvidedInterfacesList,
                RequestedInterfacesList),
```

```
%load in Loader
(3) load(ComponentType,InstanceName,ComponentState)->
        gen_server:call({global,capsule},{load,ComponentType,InstanceName}).

%load in Capsule
(4) load(ComponentType,InstanceName)->
        ComponentPid = global:whereis_name(InstanceName),
        if
                ComponentPid /= undefined ->
                        gen_server:cast({global,InstanceName}, stop),
                        global:unregister_name(InstanceName),
                        gen_server:start_link({global,InstanceName},ComponentType,
                                [InstanceName], []);
                true->io:format("~w should be created before loading!!!~n",[InstanceName])
        end.

%load the Interfaces and the meta data of a component
load(InstanceName)->
(6)     %initialize the interfaces and load the meta data of the interfaces
        CapletName = crtk:getCaplet(InstanceName),
        Interface = list_to_atom(atom_to_list(InstanceName)++"Interface"),
        gen_server:start_link({global,Interface},
                interfaceModule,[InstanceName,Interface],[]),
        interfaceModule:load(Interface),
        insert_component(Interface,interface,InstanceName,CapletName),
(7)     %load the meta data of the component
        V_type={type,text,[adder,mult]},
        MetaDataList=[V_type],
        crtk:deleteallprop(InstanceName),
        [crtk:putprop(InstanceName,MetaDataType,MetaDataName,MetaDataValue)||{MetaDataType
                ,MetaDataName,MetaDataValue}<-MetaDataList].
```

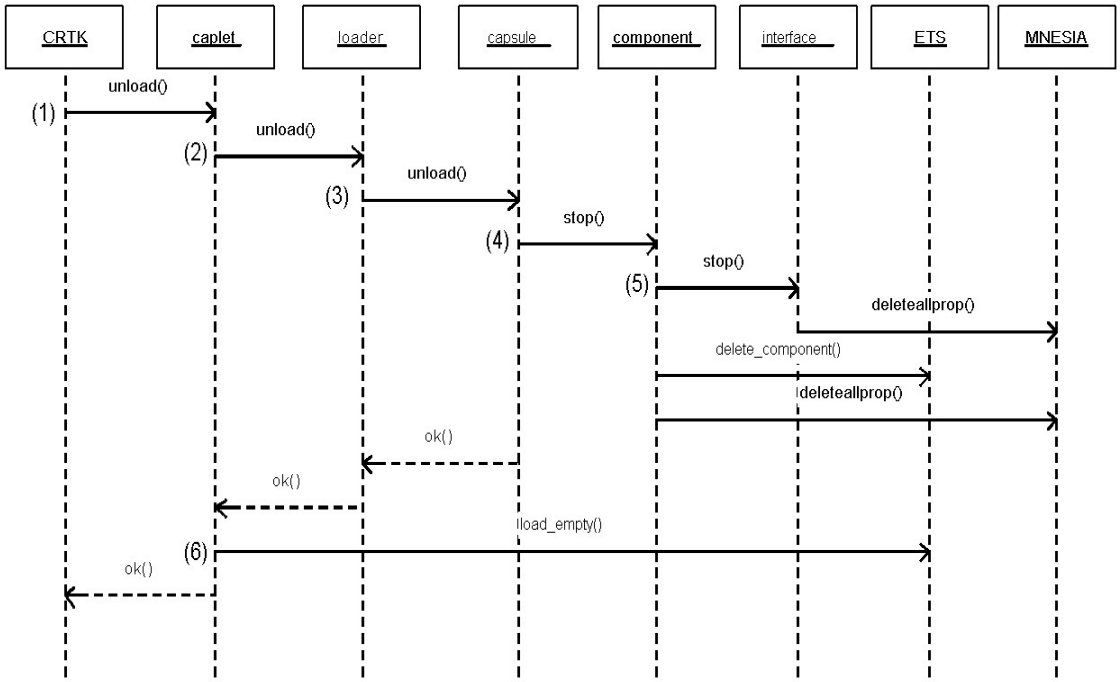

Figure 5. Unload Component

4.3.1.3. Unload Component Code

```
%unload component in CRTK
(1) unload(InstanceName)->
        CapletName = getRegistryOwner(InstanceName),
        if
                CapletName /= undefined->
                        gen_server:call({global,CapletName}, {unload,InstanceName});
                true->badarg
        end.

%unload component in Caplet:
unload_component(InstanceName)->
(2)     LoaderName=getLoader(InstanceName),
        ModuleName = getModuleName(InstanceName),
        gen_server:call({global,LoaderName}, {unload, InstanceName }),
        ModuleName:unload(ComponentName).
(6)     %insert the empty component data to the ETS table
        insert_component(InstanceName,component).


%unload in Loader
(3) unload(InstanceName)->
        gen_server:call({global,capsule},{unload,InstanceName}).


%unload in capsule:
unload(InstanceName)->
(4)     gen_server:cast({global,InstanceName}, stop),
        global:unregister_name(InstanceName),
        %start the empty component
        gen_server:start_link({global,InstanceName},e_EmptyComp, [], []),

%stop interfaces and delete meta data in component:
unload(InstanceName)->
(5)     %destruct the interfaces and delete the meta data of the interfaces
        Interface = list_to_atom(atom_to_list(InstanceName)++"Interface"),
        interfaceModule:unload(Interface),
        delete_component(Interface),
        gen_server:cast({global,Interface },stop),
        global:unregister_name(Interface).
        crtk:deleteallprop(InstanceName).
```

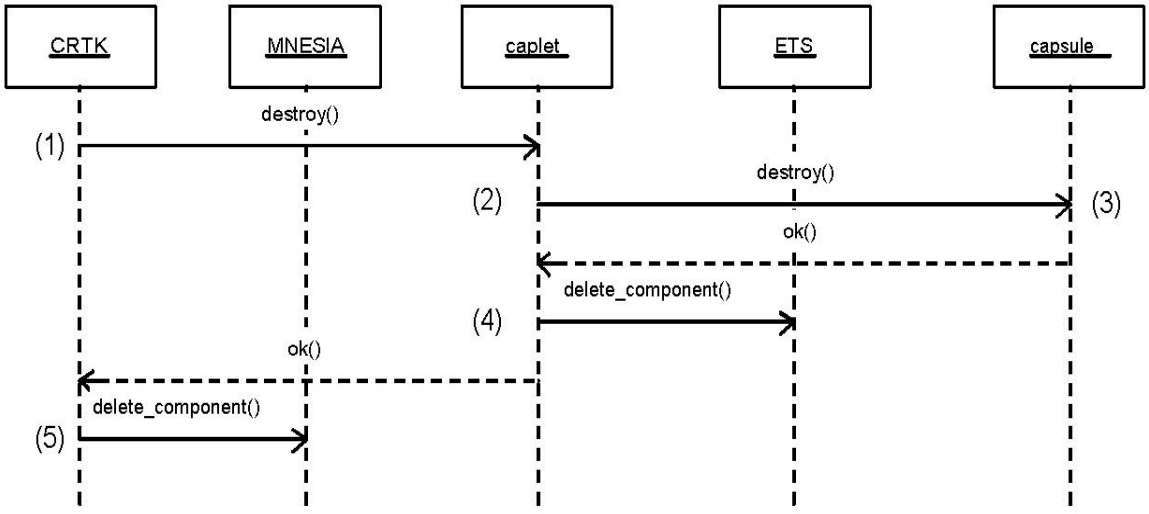

Figure 6. Destroy Component

4.3.1.4. Destroy Component Code

```
%destroy component in CRTK
(1) destroy(InstanceName)->
        CapletName = getRegistryOwner(InstanceName),
        if
                CapletName /= undefined->
                        gen_server:call({global,CapletName}, {destroy,InstanceName}),
(5)                     delete_component(InstanceName);
                true->badarg
        end.

%destroy component in caplet:
destroy(InstanceName)->
(2)     CapsuleName = crtk:getOwner(crtk:getSelfName()),
        gen_server:call({global,CapsuleName}, {destroy,InstanceName}),
        State=getState(InstanceName),
        if
                State==loaded->unload_component(InstanceName);
                true->ok
        end,
(4)     delete_component(InstanceName).

%destroy component in capsule:
(3) destroy(InstanceName)->
        gen_server:cast({global,InstanceName}, stop).


%delete a component from the ETS
(4) delete_component(InstanceName)->
        ets:delete(get(componentTable),ComponentName),

%delete component from MNESIA:
(5) delete_component(InstanceName) ->
        Fun = fun() ->
                mnesia:delete({component,ComponentName})
        end,
        mnesia:transaction(Fun).
```

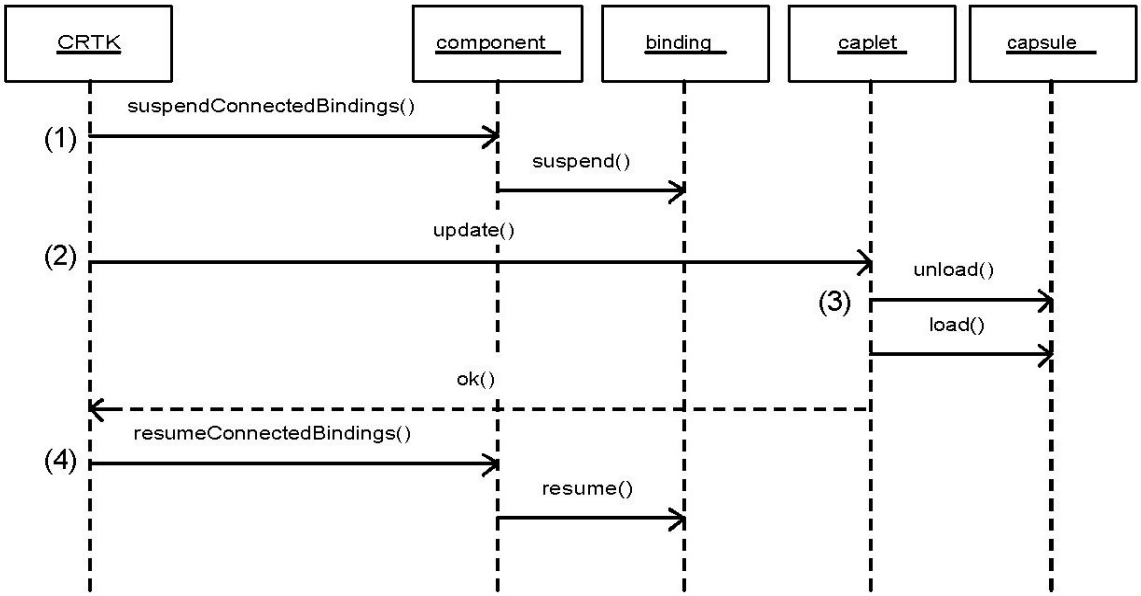

Figure 7. Update Component

4.3.1.5. Update Component Code

```
%update component in CRTK
update(ComponentType,InstanceName)->
        State=getState(InstanceName),
        if
                State == loaded->
(1)                     suspendConnectedBindings(InstanceName),
(2)                     gen_server:call({global,CapletName},
                                {update,ComponentType,InstanceName}),
(4)                     resumeConnectedBindings(InstanceName);
                State == true->
                io:format("~w should be loaded before updating.!!!~n",[InstanceName])
        end.

%update component in caplet:
(3) update(ComponentType,InstanceName)->
        LoaderName = getLoader(InstanceName),
        unload_component(InstanceName),
        load_component(LoaderName,ComponentType,InstanceName).

%suspend the bindings connected to the component:
(1) suspendConnectedBindings(InstanceName)->
        F=fun(X)->
                BindingName=getBinding(X),
                sys:suspend({global,BindingName})
        end,
        ConnectedInterfaces=getConnectedInterfaces(InstanceName),
        [F(Interface)||Interface<-ConnectedInterfaces],
        ConnectedReceptacles=getConnectedReceptacles(InstanceName),
        [F(Receptacle)||Receptacle<-ConnectedReceptacles].

%resume the bindings connected to the component:
(4) resumeConnectedBindings(InstanceName)->
        F=fun(X)->
                BindingName=getBinding(X),
                sys:resume({global,BindingName})
        end,
        ConnectedInterfaces=getConnectedInterfaces(InstanceName),
        [F(Interface)||Interface<-ConnectedInterfaces],
        ConnectedReceptacles=getConnectedReceptacles(InstanceName),
        [F(Receptacle)||Receptacle<-ConnectedReceptacles],
```

4.3.2. Binding Operations

The CRTK API requires that before two components are able to communicate to each other a communication channel should be established between the two parties. Two operations are provided to manage the binding between the receptacle and the interface of the communicating parties. They are the following:

- **bind**: create a binding between the receptacle and the interface (Section 4.3.2.1)
- **unbind**: destroy the binding between the receptacle and the interface (Section 4.3.2.2)

In the following sections the details of the above mentioned operations can be seen.

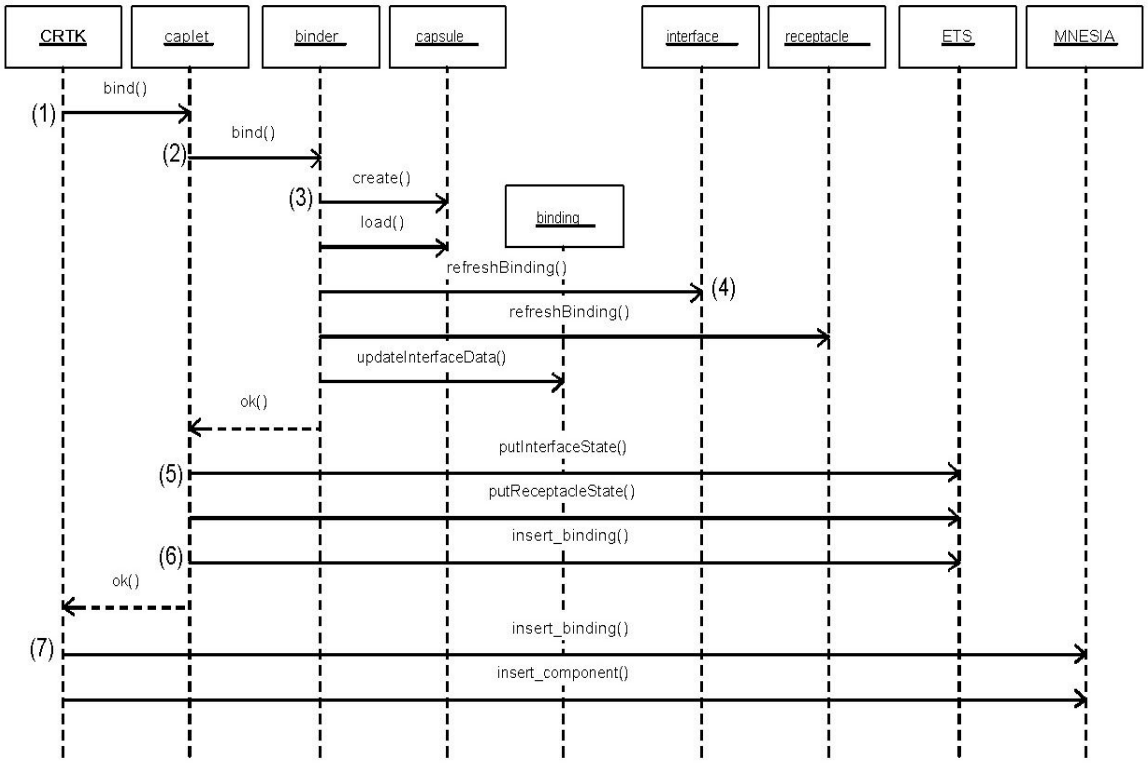

Figure 8. Bind

4.3.2.1. Bind Code

```
%Bind an interface to a receptacle in CRTK
bind(BinderName,InterfaceName,ReceptacleName,BindingName,CapletName)->
(1)     gen_server:call({global,CapletName},
              {bind,BinderName,InterfaceName,ReceptacleName,BindingName,BindingName}),
(7)     insert_binding(BindingName,InterfaceName,ReceptacleName),
        insert_component(BindingName,binding,CapletName,CapletName),
        BindingName:load(BindingName).

%Bind an interface to a receptacle in caplet:
bind(BinderName,InterfaceName,ReceptacleName,BindingName,ModuleName)->
(2)     gen_server:call({global,BinderName},{bind,InterfaceName,ReceptacleName,
              BindingName}),
        CapletName = crtk:getSelfName(),
        meta_private:insert_component(BindingName,binding,CapletName,CapletName),
(5)     crtk:putInterfaceState(InterfaceName,connected,BindingName,provided),
        crtk:putInterfaceState(ReceptacleName,connected,BindingName,requested),
(6)     insert_binding(BindingName,BinderName,InterfaceName,ReceptacleName).

%Bind an interface to a receptacle in Binder:
bind(InterfaceName,ReceptacleName,BindingName)->
(3)     gen_server:call({global,e_node1Capsule},{create,BindingName}),
        gen_server:call({global,e_node1Capsule},{load,defaultBinderBehavior,
              BindingName,created}),
        gen_server:call({global,InterfaceName},{refreshBinding,BindingName}),
        gen_server:call({global,ReceptacleName},{refreshBinding,BindingName}),
        gen_server:call({global,BindingName},{updateInterfaceData,InterfaceName}),

%insert_binding in Caplet
(6) insert_binding(BindingName, BinderName, InterfaceName, ReceptacleName)->
        ets:insert(get(componentTable),#component{componentName=BindingName,
        componentData=#componentData{componentType=binding,binderName=BinderName,
        state=loaded,bindingData=#bindingData{source=ReceptacleName,
        target=InterfaceName},interceptionList=#interceptionList{preAction=[],
        postAction=[]},moduleName=ModuleName}}).
```

```
%insert_binding in CRTK
(7) insert_binding(BindingName,InterfaceName,ReceptacleName,IsOnDemand) ->
        Fun = fun() ->
                mnesia:write(#binding{bindingName=BindingName,interfaceName=InterfaceName,
                        receptacleName=ReceptacleName})
        end,
        mnesia:transaction(Fun).

%refreshBinding in an Interface/Receptacle
(4) handle_call({refreshBinding,BindingName},_Client,State)->
        OldBindingName=State#stateData.bindingName,
        if
                BindingName /= undefined->link(global:whereis_name(BindingName));
                true->unlink(global:whereis_name(OldBindingName))
        end,
        NewState=State#stateData{bindingName=BindingName},
        {reply,ok,NewState};
```

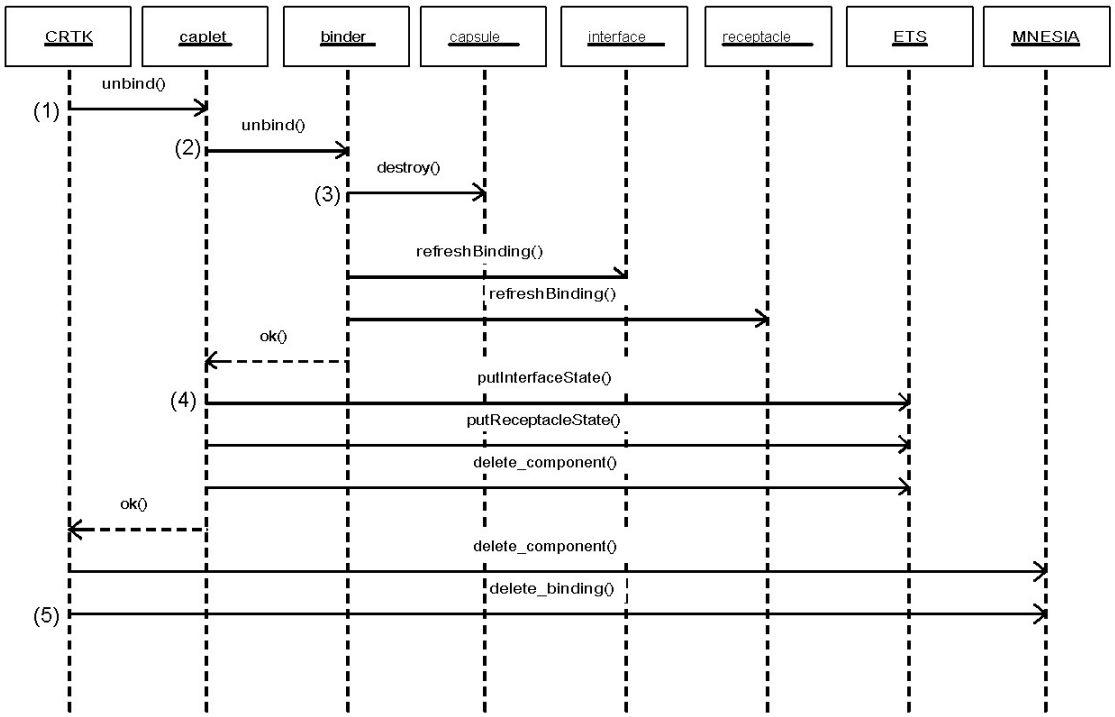

Figure 9. Unbind

4.3.2.2. Unbind Code

```
%Unbind an interface from a receptacle in CRTK
(1) unbind(InterfaceName,ReceptacleName)->
        BindingName = getBinding(InterfaceName,ReceptacleName),
        gen_server:call({global,CapletName},
                {unbind,InterfaceName,ReceptacleName,BindingName}),
        BindingName:unload(BindingName);
        delete_component(BindingName),
(5)     delete_binding(BindingName).

%Unbind an interface from a receptacle in caplet:
unbind(InterfaceName,ReceptacleName,BindingName)->
        BinderName = getBinder(BindingName),
        if
                BinderName /= undefined->
(2)                     gen_server:call({global,BinderName},
```

```
                        {unbind,InterfaceName,ReceptacleName,BindingName}),
(4)                     crtk:putInterfaceState(InterfaceName,unconnected,nil,provided),
                        crtk:putInterfaceState(ReceptacleName,unconnected,nil,requested),
                        delete_component(BindingName);
                true->io:format("BinderName is undefined for ~w~n",[BindingName])
        end.

%Unbind an interface to a receptacle in Binder:
(3) unbind(InterfaceName,ReceptacleName,BindingName)->
        gen_server:call({global,CapsuleName}, {destroy, BindingName }),
        gen_server:call({global,InterfaceName},{refreshBinding,undefined}),
        gen_server:call({global,ReceptacleName},{refreshBinding,undefined}).

%delete_binding in CRTK
(5) delete_binding(BindingName) ->
        Fun = fun() ->
                mnesia:delete({binding,BindingName})
        end,
        mnesia:transaction(Fun).
```

4.3.3. Reflective Operations

The CRTK API provides reflective operations for the components to be able to look up run-time and meta information to adapt their behavior to the changing environment. The following operations are supported in the current version of ErlCOM:

- **getallprop**/1: Get all meta data of an Entity. An Entity could be a Capsule, Caplet, Loader, Binder, Component, Interface, Receptacle or Binding Component.
- **deleteallprop**/1: Delete all meta data of an Entity.
- **getAllComponents**/0: Get all the Components of the system.
- **getAllBindings**/0: Get all Binding Components of the system.
- **getAllInterfaces**/1: Get all interfaces of a Component.
- **getAllReceptacles**/1: Get all receptacles of a Component.
- **getAllInterfaces**/0: Get all interfaces of the system.
- **getAllReceptacles**/0: Get all receptacles of the system.
- **getAllCaplets**/1: Get all caplets of the system.
- **getBinders**/1: Get all Binders of a Capsule.
- **getLoaders**/1: Get all Loaders of a Capsule.
- **addPreAction**/5: Add a pre-action to a Binding Component.
- **addPostAction**/5: Add a post-action to a Binding Component.
- **deletePreAction**/2: Delete a pre-action from a Binding Component.
- **deletePostAction**/2: Delete a post-action from a Binding Component.
- **getSelfName**/0: Get the global registered name of an Entity.
- **getCaplet**/1: Get the globally registered name of a Component, Binding Component, Interface or Receptacle.
- **getBinding**/2: Get the Binding Component connected to the given Interface and Receptacle.
- **getOwner**/1: Get the owner of the given Entity.
- **putInterfaceState**/4: Set the connectivity state (connected, unconnected) and the Binding component of the given Interface or Receptacle.
- **isConnected**/1: Get the connectivity status of the given Interface or Receptacle.
- **getBinding**/1: Get the Binding Component that is connected to the given Receptacle or Interface.

5. ErlCOM IDE

In Section 3 and Section 4 we described the principles and the implementation details of the ErlCOM. As it has been demonstrated ErlCOM needs a relatively complex architectural implementation in Erlang so that flexible component deployment and interaction could be achieved. In order to alleviate the programmer's task to worry about editing files we created an IDE that looks like a Service Creation Environment. The principles of the IDE derive from our methodology [6] that relies on the fact that every Domain Specific Language (DSL) can be regarded as a tuple of *concrete syntax*, *abstract syntax* and *semantics*. The concrete syntax specifies the textual and/or graphical representation of the language elements, the abstract syntax describes the relationships among the concepts of the language elements and the semantics enfleshes the abstract concepts with meaning. Our methodology encourages the programmer to create a series of DSLs to attack the problem; each language should be created in such a manner that it is isomorphic to the problem solution on that particular level. The refinement of the solutions is realized by a translation process between the languages. In the case of ErlCOM we have two domain specific languages in action: ErlCOM and Erlang/OTP. The language aspects of Erlang/OTP are well known, so the language definition of ErlCOM can be based on them. Our methodology utilizes Generic Modeling Environment (GME) [7] to provide precise language definition, therefore, ErlCOM is described in GME. The abstract syntax is specified by Entity Relationship Diagrams (ERD) and related constraints formulated in OCL. The ERD describing the core of ErlCOM is shown in Figure 10

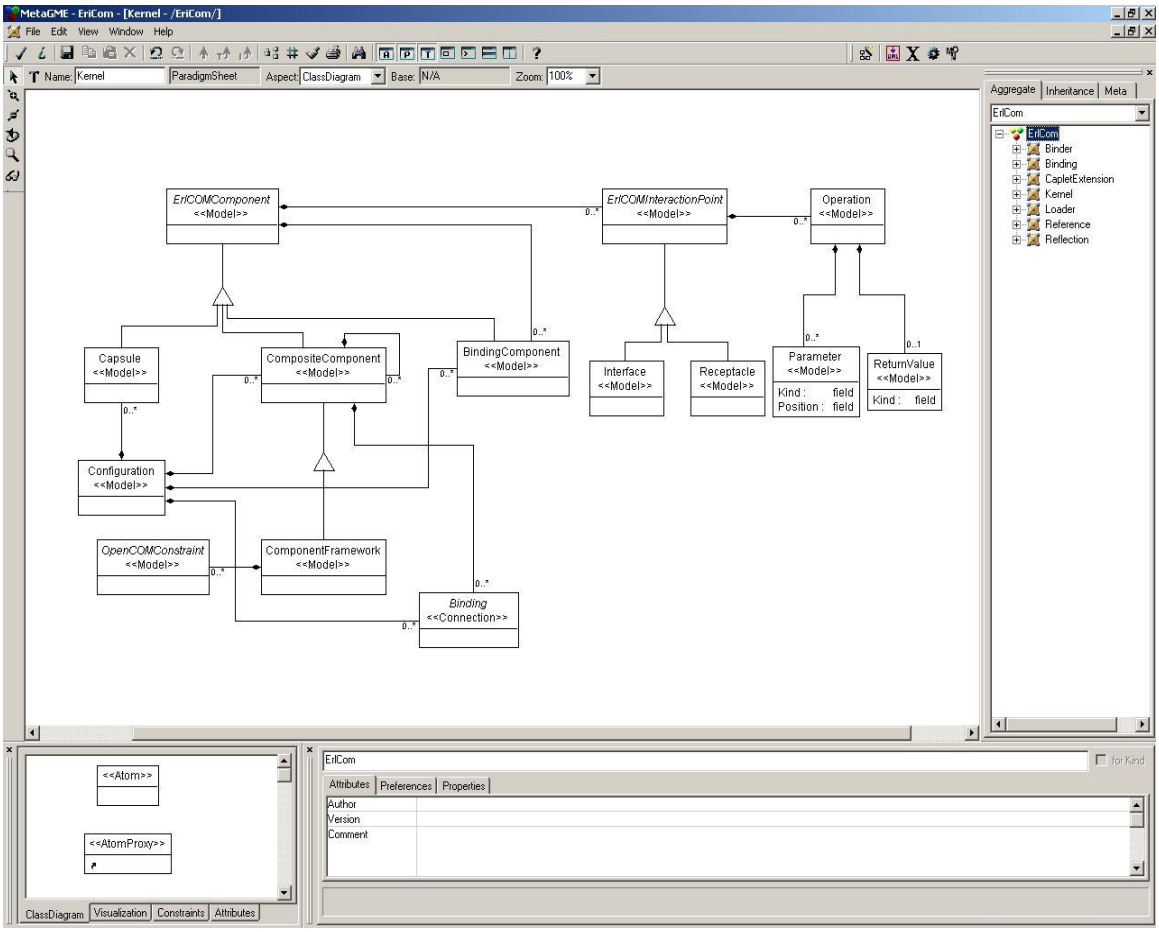

Figure 10. Abstract Syntax of ErlCOM

The concrete syntax provided by the IDE is the graphical representation of the ErlCOM concepts. Obviously, it concerns only the concepts of ErlCOM, that is, the Erlang code residing inside the components and the interaction points is not touched in any ways and can be produced arbitrarily. A sample application using ErlCOM's concrete syntax is shown in Figure 11.

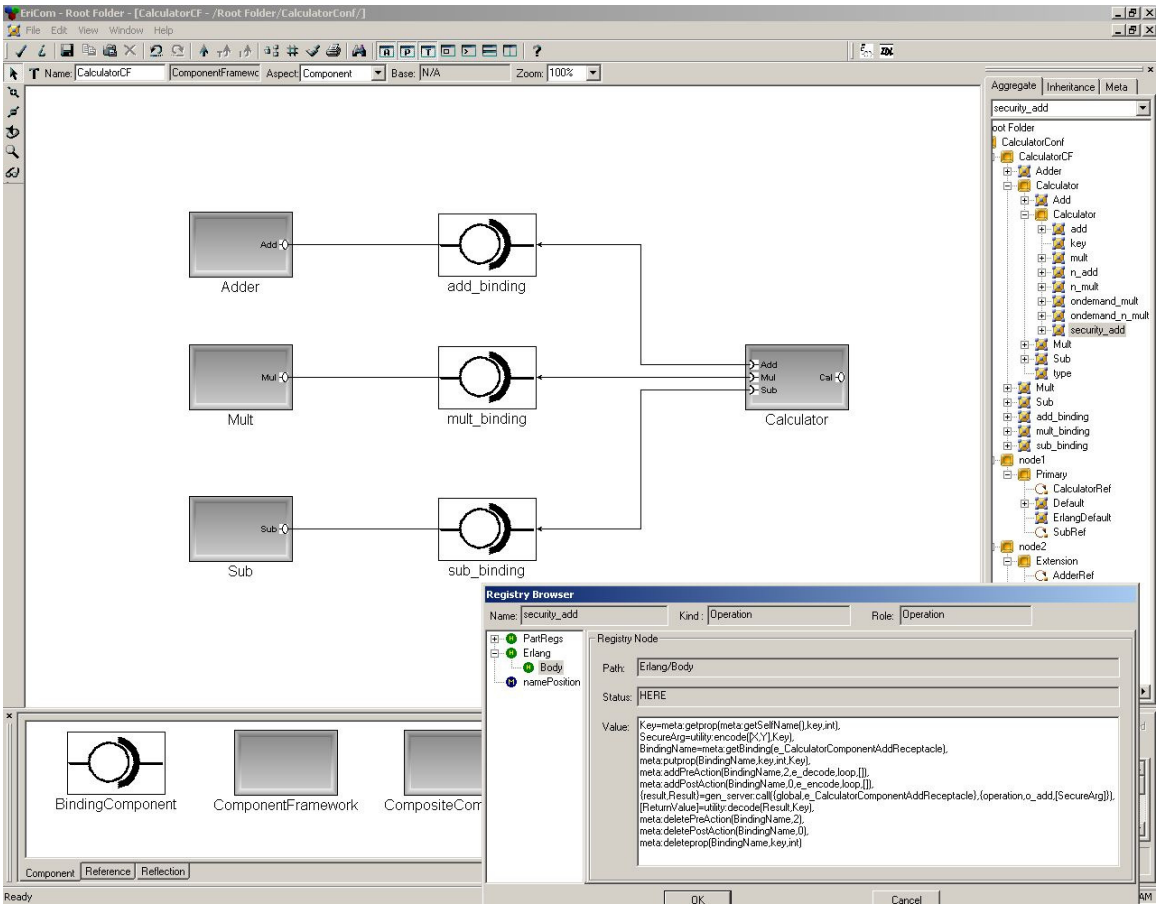

Figure 11. Concrete Syntax of ErlCOM

The semantics of ErlCOM is defined relative to Erlang, that is, ErlCOM's concepts are translated into Erlang. The implementation of ErlCOM, explained in Section 4, is automatically produced by a translator that understands the abstract syntax and generates Erlang code accordingly. The translator only deals with the architecture code; the Erlang code in the body of the components and the interaction points is treated transparently. It means that if the model graph of the component framework has been modified and the translation has been carried out the corresponding gen_servers get updated and redeployed on the fly.

To summarize our approach, the ERDs provide the abstract syntax; the concrete syntax is designed to facilitate the programmer's task by assigning textual and/or graphical mnemonics and syntactical sugar to the elements of the abstract syntax and the semantics utilizes the SDs and the corresponding Erlang code snippets to assign meaning to the elements of the abstract syntax.

6. Future Work

The robust reconfigurability of ErlCOM promotes it as a possible candidate to be profitably applied in constantly changing environments where applications should be able to frequently adapt to new circumstances. Moreover, our methodology enables the programmer to concentrate on the application logic and the generated component architecture automatically provides access to the distributed reflective CRTK. In the framework of the ongoing RUNES IST project we propose to use ErlCOM in the gateway nodes of the sensor network since it seems that the gateways own enough resources to be able to run Erlang virtual machines and sensor network reflectivity might be sufficiently represented in the gateways. We hope that our efforts help us discover new terrains where Erlang can be successfully deployed in the future.

7. References

[1] H. Karl, A. Willig: Protocols and Architectures for Wireless Sensor Networks, John Wiley & Sons, 2005.
[2] RUNES IST Project, http://www.ist-runes.org/
[3] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama: OpenCOM v2: A component model for building systems software, Proceedings of IASTED Software Engineering and Applications (SEA'04),
[4] Open Source Erlang, http://www.erlang.org/
[5] Joe Armstrong: Making reliable distributed systems in the presence of software errors, Doctoral Thesis, Stockholm, Sweden, 2003
[6] Z. Theisz, G. Batori, D. Asztalos: On a model based methodology, ACM Symposium on Applied Computing Special Track on Model Transformation (MT'06), Dijon, France (submitted).
[7] GME Documentation, http://www.isis.vanderbilt.edu/Projects/gme/