11th International Erlang/OTP User Conference

Stockholm, November 10, 2005

Proceedings

http://www.erlang.se/euc/05/

ERICSSON

Erlang Training and Consulting

corelatus

Mobile Arts

Sjöland&Thyselius

ERLANG

Erlang/OTP User Conference 2005

Conference Programme

08.30 *Registration.*

Session I

09.00 **Robust Reconfigurable Erlang Component System.**
Gabor Batori, Zoltan Theisz and Domonkos Asztalos, Ericsson, Hungary.

09.30 **Performance Measurement and Applications Benchmarking with Erlang.**
Mickaël Rémond, Process-one, France.

10.00 **A Virtual World Distributed Server developed in Erlang as a Tool for analysing Needs of Massively Multiplayer Online Game Servers.**
Michal Slaski, Erlang Training and Consulting, UK.

10.30 *Coffee.*

Session II

11.00 **Third Party Gateway.**
Chandrashekhar Mullaparthi, T-Mobile, UK.

11.30 **eXAT: Software Agents in Erlang.**
Corrado Santoro, University of Catania, Italy.

12.00 **e-TopUp.**
Eduardo Figoli, Bernardo Paroli and Carlos E. Silva, IN Switch Solutions Inc, USA.

12.30 *Lunch.*

Session III

14.00 **Teaching Functional Programming and Erlang.**
Victor M. Gulias, University of A Coruña, Spain.

14.30 **Concurrent Erlang Flow Graphs.**
Manfred Widera, FernUniversität Hagen, Germany.

15.00 **Structured Network Programming.**
Ulf Wiger, Ericsson, Sweden.

15.30 *Coffee.*

Session IV

16.00 **wxErlang.**
Mats-Ola Persson, Chalmers, Sweden.

16.20 **gtkNode - Yet Another GUI Framework for Erlang.**
Mats Cronqvist, Ericsson, Sweden.

16.40 **Bit-level Binaries and Generalized Comprehensions in Erlang.**
Per Gustafsson and Kostis Sagonas, University of Uppsala, Sweden.

17.10 **OTP Development Update.**
Kenneth Lundin, OTP team, Ericsson, Sweden.

17.30 *Close followed by bus transport to an ErLounge.*

Demonstrations (during intermissions)

Vlad Dumitrescu demonstrates **Erlide**, Eclipse IDE for Erlang.
Simon Aurell demonstrates controlling remote appliances using instant messaging.

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 *components*.
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 *protocols*.

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_servers* 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.

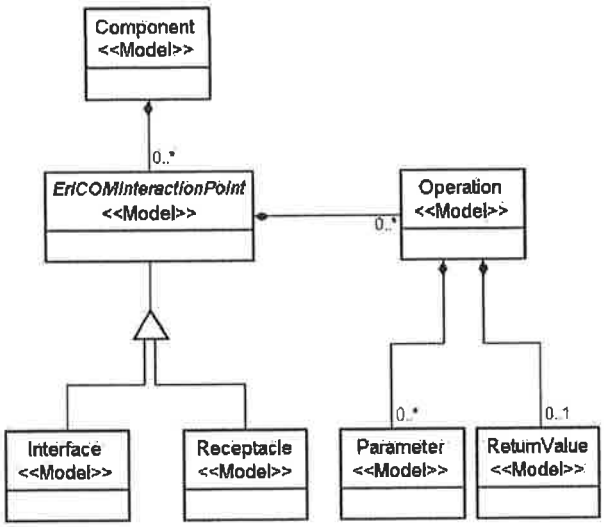


Figure 1. ERD of component and related concepts

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.

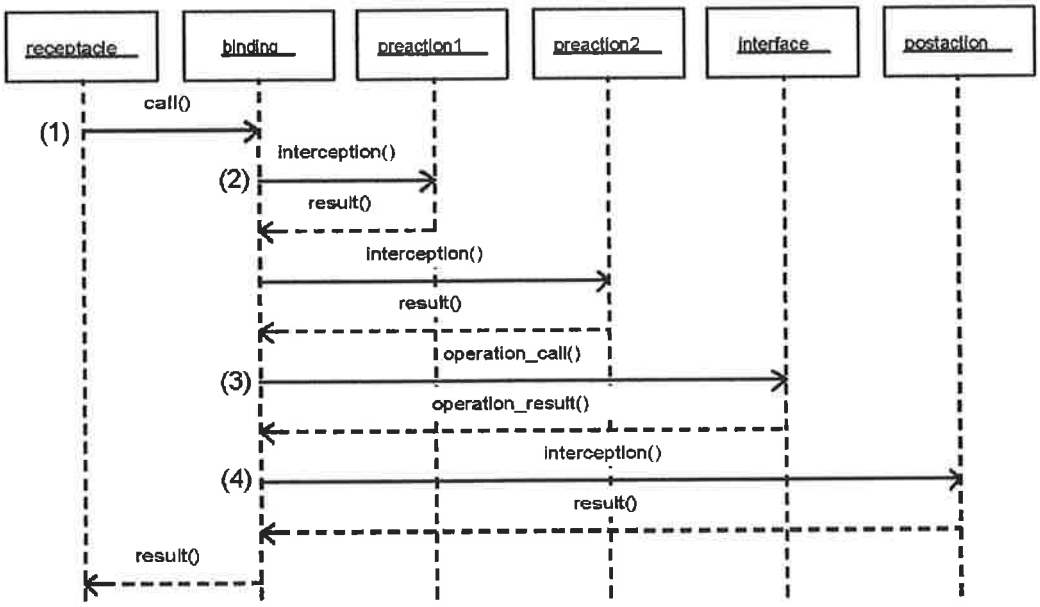


Figure 2. Component Communication

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.

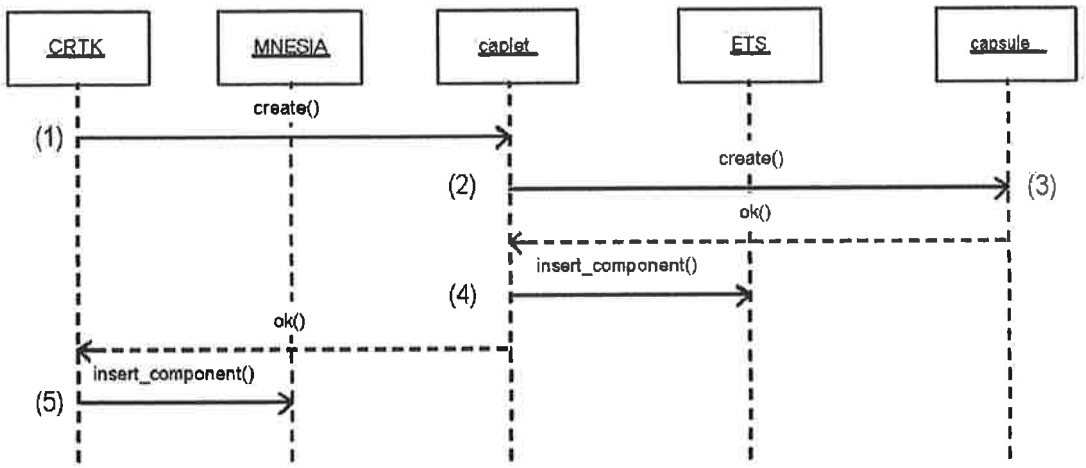


Figure 3. Create Component

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).
```

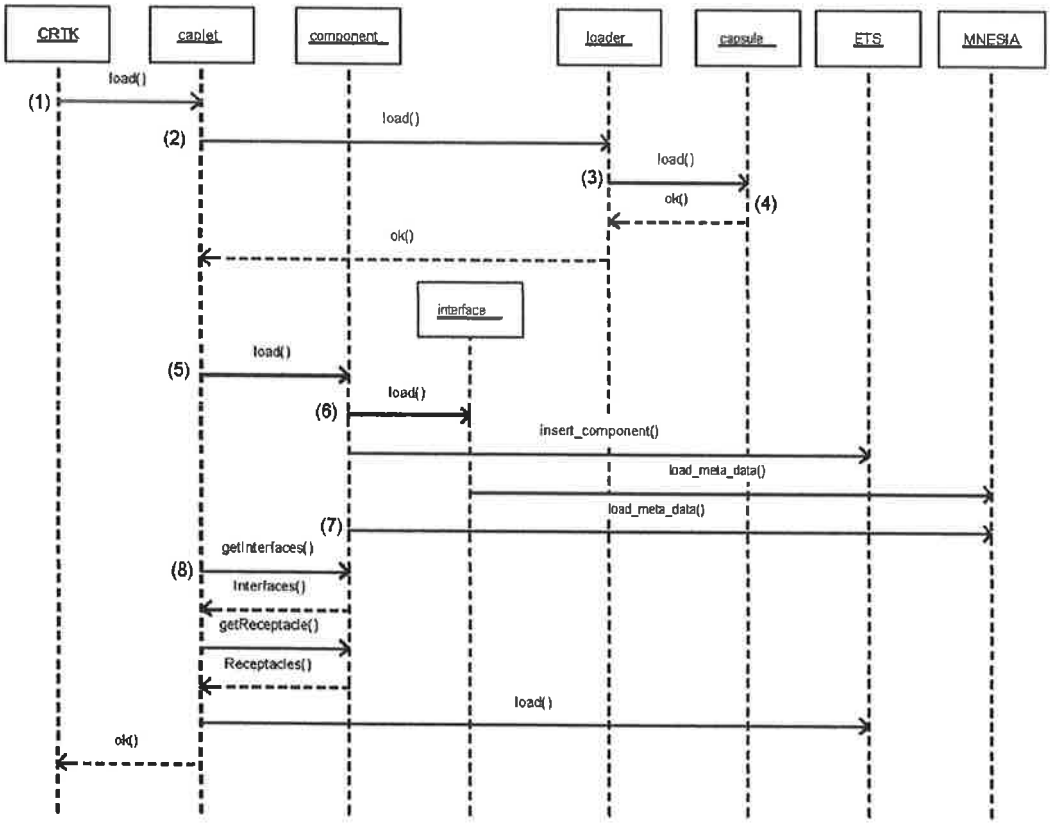


Figure 4. Load Component

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].
```

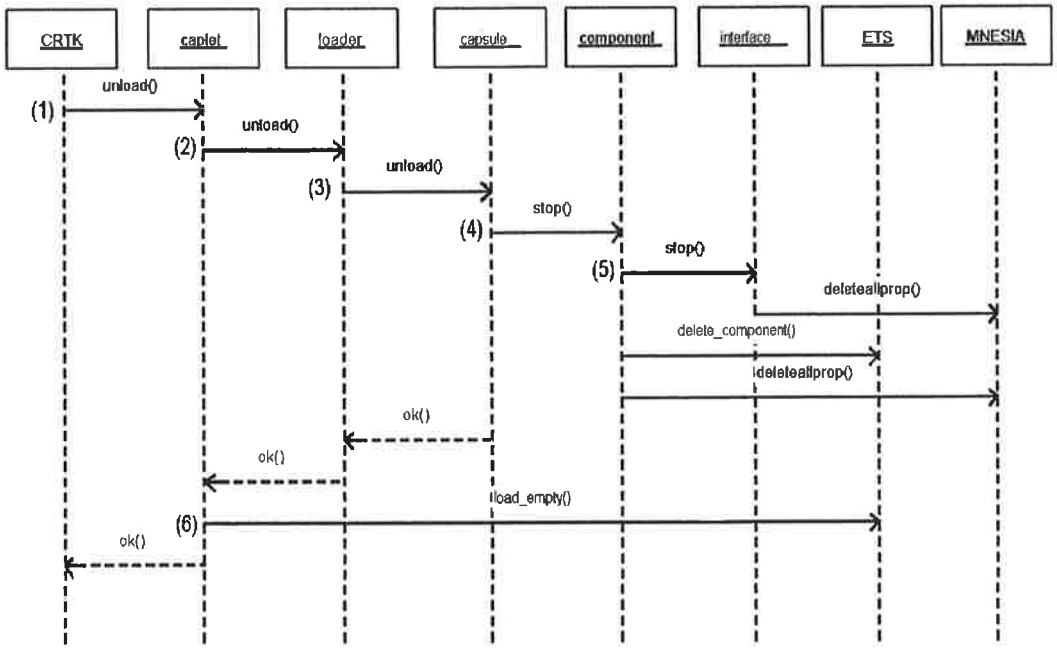


Figure 5. Unload Component

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).
```

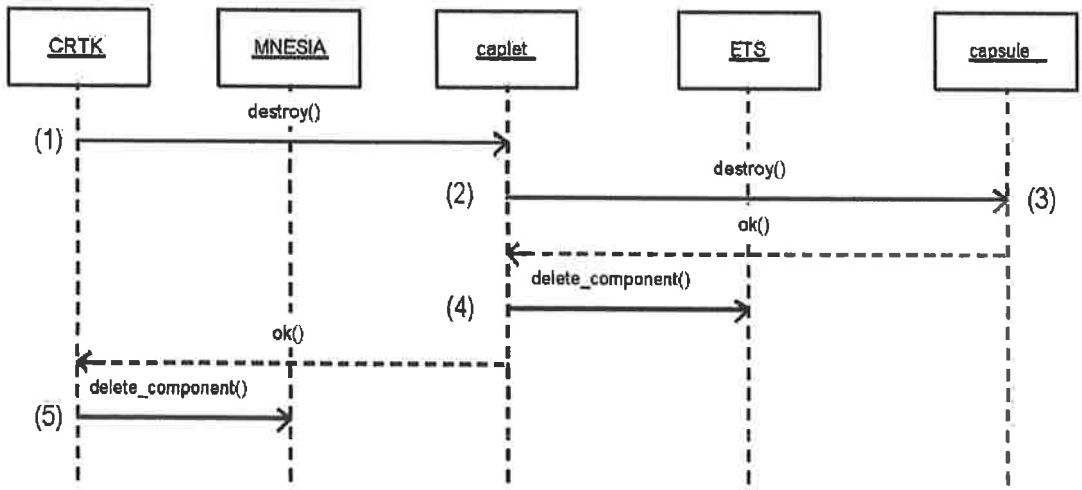


Figure 6. Destroy Component

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).
```

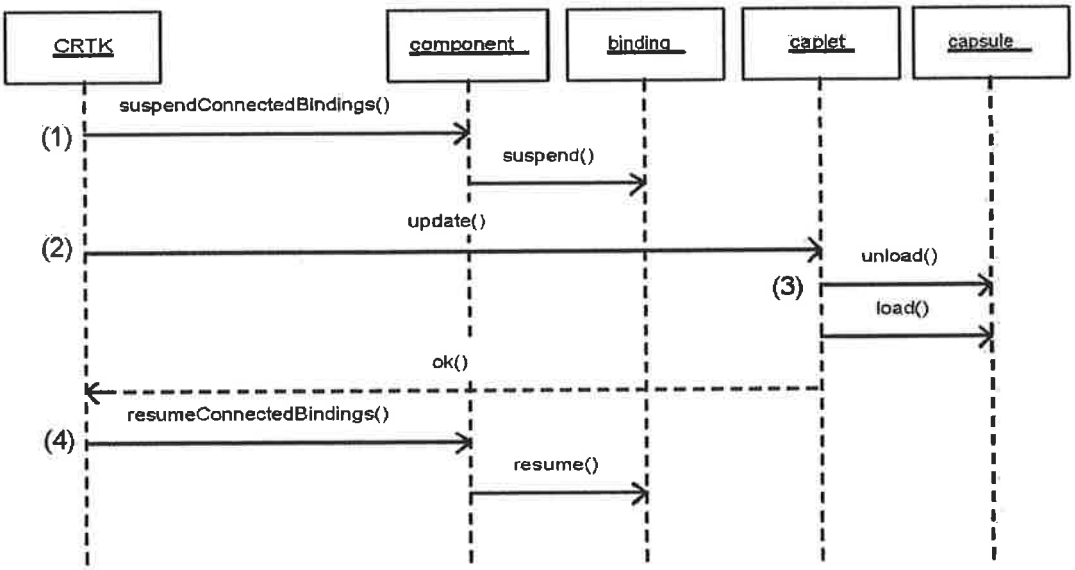


Figure 7. Update Component

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.

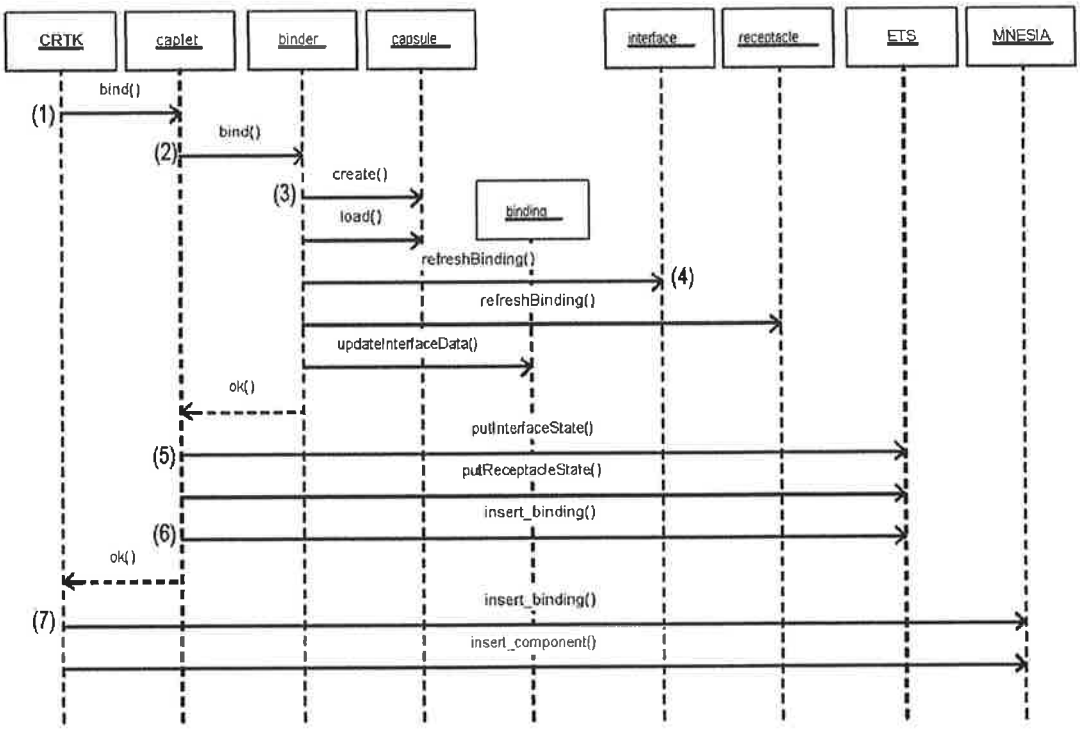


Figure 8. Bind

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};
```

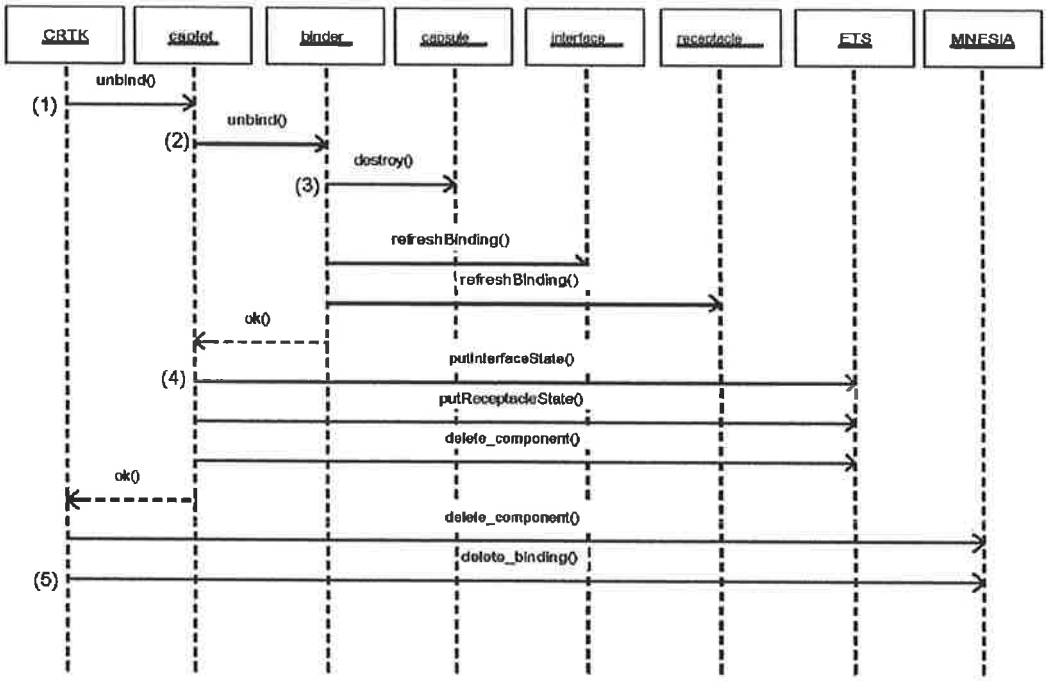


Figure 9. Unbind

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

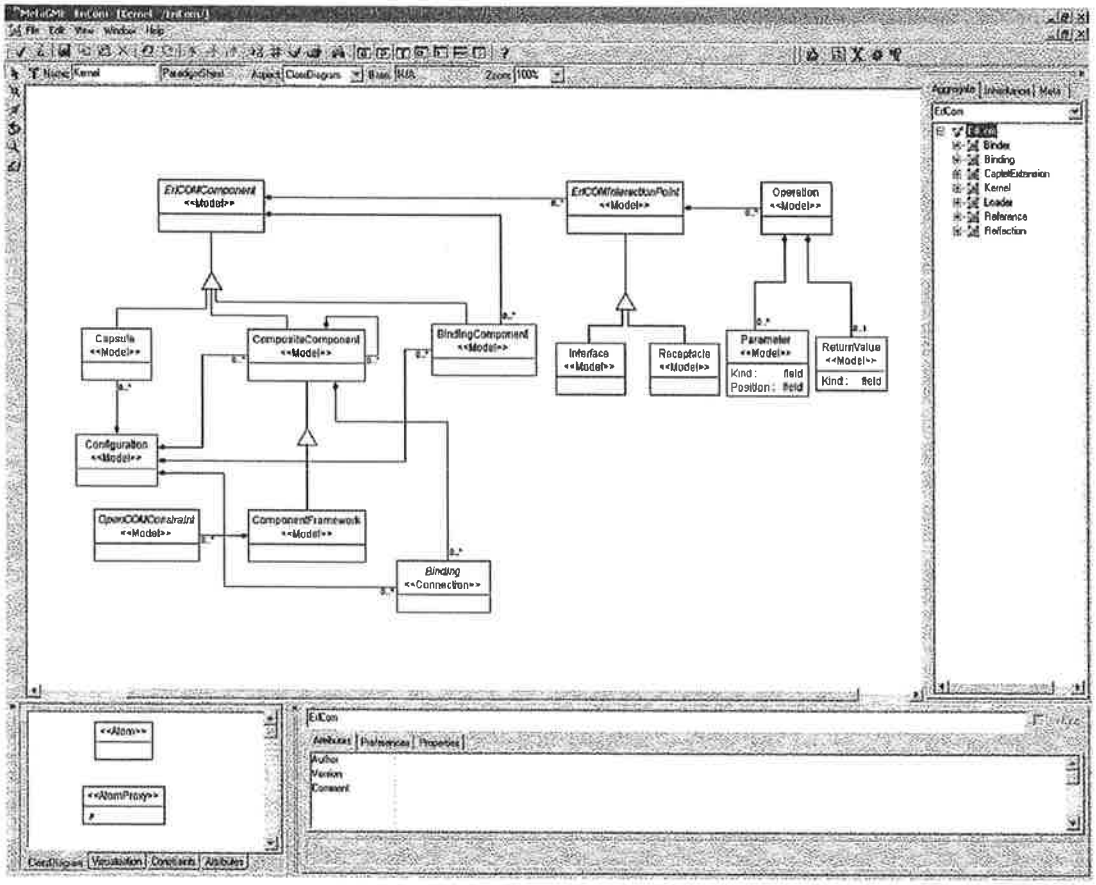


Figure 10. Abstract Syntax of ErlCOM

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.

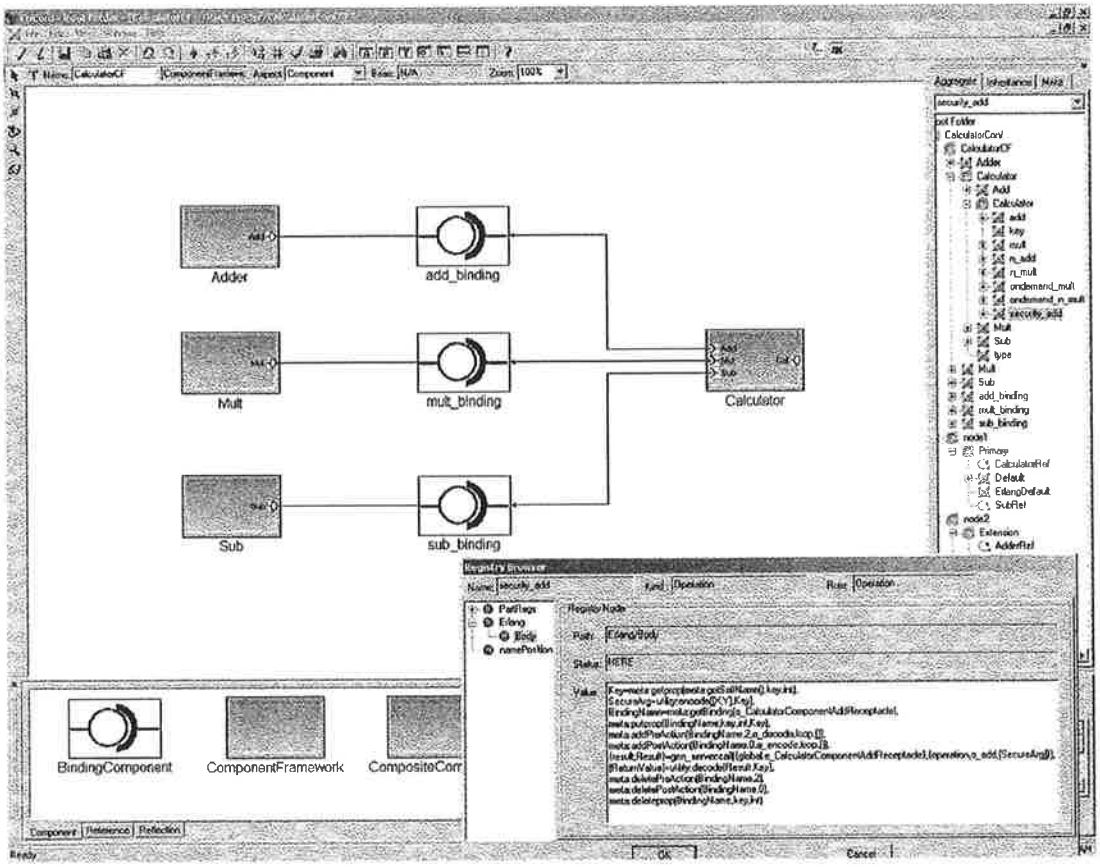


Figure 11. Concrete Syntax of ErlCOM

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/

Performance Measurement and Applications Benchmarking with Erlang

11th Erlang User Conference
November 10, 2005

Mickaël Rémond

The benchmarking challenge

- **Applications benchmarking and performance mesurement requires:**
  - To be able to simulate an important number of users playing real-life scenarii.
  - To be able to do near real-time and high throughput mesurements to provide reliable figures.
- **Tsunami is a distributed load testing tool that has been designed as a heavy duty benchmarking tool and framework**
- **The software is protocol-independent. It currently can be used to stress HTTP, SOAP and Jabber/XMPP servers, but other protocols can be added.**
- **This talk presents Tsunami main achievements, along with real life use cases, and explore possible framework extensions and improvements.**

Summary — What is Tsunami ?

- **Tsunami main strength rely in its ability to simulate:**
  - a huge number of simultaneous users from a single CPU,
  - an heavier load in cluster mode.
- **When used on cluster you can generate a really impressive load on a server with a modest cluster, easy to set-up and maintain.**
- **Tsunami is developed in Erlang and this is where the power of Tsunami relies. Tsunami is based on the Erlang OTP (Open Transaction Platform) and inherits several characteristics from Erlang:**
  - **Performance:** Erlang has been made to support hundred thousands of lightweight processus in a single virtual machine.
  - **Scalability:** Erlang development environnement is naturally distributed, promoting the idea of processus location transparence.
  - **Fault-tolerance:** Erlang has been built to develop robust, fault-tolerant systems. As such, wrong answer sent from the server to Tsunami does not make the whole running benchmark crash.

Tsunami background: A strong simulation model

- **Tsunami has been developed by Nicolas Niclausse**
- **It is an industrial implementation of a stochastic model for real users simulation. User events distribution is based on a Poisson Law. This model is used to closely simulate real-world user behaviour.**
- **Tsunami is being developed since 2001, first by IDEALX and now by Process-one.**
- **This model has already been tested in the INRIA Wagon project (Web trAffic GeneratOr and beNchmark).**

  The Wagon project has been developed in the context of MISTRAL. Its main objectif was to simulate various types of Internet trafic to study server behaviours. WAGON has been used in the context to the French national VHTD (Vraiment Très Haut Débit) project.
- **Tsunami is based on the result of Nicolas Niclausse PHD Thesis:**

  Modeling, performance analysis and dimensioning of the WWW

Tsunami main features (1/2)

- **High Performance:**
  - Tsunami can simulate a huge number of simultaneous users per physical computer. It can simulates up to 10000 users on a single CPU. Traditionnal injection tools can hardly go further than 200 users.
- **Distributed:**
  - The load can be distributed on a cluster of client machines
- **Multi-Protocols using a plugin system:**
  - HTTP (both standard web traffic and SOAP) and Jabber are currently supported. LDAP and SMTP are on the TODO list.
- **SSL support**
- **Several IP addresses can be used on a single machine using the underlying OS IP Aliasing**
- **OS monitoring:**
  - CPU, memory and network trafic can be monitored using Erlang agents on remote servers or SNMP

Tsunami main features (2/2)

- **XML configuration system**
- **Mixed behaviours:**
  - Several sessions can be used to simulate differents type of users during the same benchmark. You can define the proportion of the various behaviours in the benchmark scenario.
- **Stochastic processes:**
  - In order to generate a realistic trafic, user thinktimes and the arrival rate can be randomize using a probability distribution (exponential currently)
- **Adaptative scenarii:**
  - Scenarii can have dynamic part, that depends on the result of the current scenario request or can be generated by Erlang code.
- **Complete statistic set**

HTTP related features

- HTTP/1.0 and HTTP/1.1 support
- GET and POST requests
- Cookies: Automatic cookies management
- 'GET If-modified since' type of request
- WWW-authentication Basic
- Proxy mode to record sessions using a Web browser
- SOAP support using the HTTP mode (the SOAPAction HTTP header is handled).

Jabber related features

- Authentication, presence and register messages
- Chat messages to online or offline users
- Roster set and get requests
- Global users' synchronisation can be set on specific actions

Complete reports set

- **Mesures and statistics produced by Tsunami are extensive. They are all represented as a graphe. Tsumami produces statistics regarding:**
  - **Performance:** response time, connexion time, decomposition of the user scenario based on request grouping instruction, requests per second
  - **Errors:** Statistics on page return code to trace errors
  - **Target cluster behaviour:** An Erlang agent can gather information from the target cluster. Tsunami produce graphes for CPU and memory consumption and network traffic. SNMP is also supported.

- **Note that Tsunami take care of the synchronisation process by itself. Gathered statistics are «synchronized».**

- **It is possible to generate graphes during the benchmark as statistics are gathered in real-time. This make it possible to see if the benchmark has to be stopped before the end of the benchmark.**

HTTP benchmark approach

- **Record scenario: tsunami start_recorder**

- **Edit / organise scenario**

- **Write small code for dynamic parts if needed and place dynamic mark-up in the scenario**

- **Test and adjust scenario to have a nice progression of the load. This is highly dependent of the application and of the size of the target cluster. Calculate the normal duration of the scenario and use the interarrival time between users and the duration of the phase to estimate the number of simultaneous users for each given phase.**

- **Launch benchmark with your first application parameters set-up: tsunami start**

- **Analyse results, change parameters and launch another benchmark**

Understanding tsunami.xml file: File structure

- Scenarii are enclosed into tsunami tags:

```
<?xml version="1.0"?>
<tsunami loglevel="info" dumptraffic="false">
...
</tsunami>
```


Understanding tsunami.xml file: Clients and server

- Scenarii start with a clients (Tsunami cluster) and server definition:

```
<clients>
   <client host="louxor" weight="1" maxusers="500">
       <ip value="10.9.195.12"></ip>
       <ip value="10.9.195.13"></ip>
   </client>
   <client host="memphis" weight="3" maxusers="250" cpu="2">
       <ip value="10.9.195.14"></ip>
   </client>
</clients>

<server host="10.9.195.1" port="8080" type="tcp"></server>
```

- Several virtual IP can be used to simulate more machines. This is very useful when a load-balancer use the client's IP to distribute the traffic amoung a cluster of servers. In this example, a second machine is used in the Tsunami cluster, with a higher weight, and 2 cpus. Two Erlang virtual machines will be used to take advantage of the number of CPU.

- The server is the entry point into the cluster (Only one server should be defined).

Understanding `tsunami.xml` file: Monitoring

- **Scenarii can contain optional monitoring informations. For example, here is a cluster monitoring definition based on Erlang agents, for a cluster of 6 computers:**

```
<monitoring>
  <monitor host="geronimo" type="erlang"></monitor>
  <monitor host="bigfoot-1" type="erlang"></monitor>
  <monitor host="bigfoot-2" type="erlang"></monitor>
  <monitor host="f14-1" type="erlang"></monitor>
  <monitor host="f14-2" type="erlang"></monitor>
  <monitor host="db" type="erlang"></monitor>
</monitoring>
```

- **The type keyword snmp can replace the erlang keyword, if SNMP monitoring is prefered. They can be mixed. erlang is the default value for monitoring.**

- **Note: For Erlang monitoring, monitored computers need to be accessible through the network. SSH needs to be configured to allow connection without password on**
  - (See: Tutorial: Erlang - Starting a set of cluster nodes on Erlang-projects.org for details)

Understanding `tsunami.xml` file: Defining the load progression

- **The load progression is set-up by defining several arrival phases:**

```
<arrivalphase phase="1" duration="10" unit="minute">
  <users interarrival="2" unit="second"> </users>
</arrivalphase>

<arrivalphase phase="2" duration="10" unit="minute">
  <users interarrival="1" unit="second"> </users>
</arrivalphase>

<arrivalphase phase="3" duration="10" unit="minute">
  <users interarrival="0.1" unit="second"> </users>
</arrivalphase>
```

Understanding `tsunami.xml` file: Default values

- **Default values can be set-up globally: thinktime between requests in the scenario and ssl cipher algorithms. These values overrides those set in session configuration tags.**

```
<default name="thinktime" value="3" random="false"/>
<default name="ssl_ciphers"
         value="EXP1024-RC4-SHA,EDH-RSA-DES-CBC3-SHA"/>
```

- **Default values for specific protocols can be defined. Here is an example of default values for Jabber:**

```
<default type="ts_jabber" name="global_number" value="5" />
<default type="ts_jabber" name="userid_max" value="100" />
<default type="ts_jabber" name="domain" value="jabber.org" />
<default type="ts_jabber" name="username" value="glop" />
<default type="ts_jabber" name="passwd" value="glop" />
```


Understanding `tsunami.xml` file: Sessions (1/2)

- **Sessions define the content of the scenario itself. They describe the requests to execute.**

```
<session name="http-example" popularity="70"
         persistent="true" messages_ack="parse" type="ts_http">

  <request> <http url="/" method="GET" version="1.1">
                 </http> </request>
  <request> <http url="/images/logo.gif"
            method="GET" version="1.1"
            if_modified_since="Fri, 14 Nov 2003 02:43:31 GMT">
           </http></request>

  <thinktime value="20" random="true"></thinktime>

  <transaction name="index_request">
   <request><http url="/index.en.html"
                      method="GET" version="1.1" >
           </http> </request>
   <request><http url="/images/header.gif"
                      method="GET" version="1.1">
           </http> </request>
  </transaction>
```

Understanding `tsunami.xml` file: Sessions (2/2)

```
    <thinktime value="60" random="true"></thinktime>
    <request>
      <http url="/" method="POST" version="1.1"
              contents="bla=blu">
      </http> </request>
    <request>
       <http url="/bla" method="GET" version="1.1"
             contents="bla=blu&name=glop">
       <www_authenticate userid="Aladdin"
                         passwd="open sesame"/></http>
    </request>
  </session>

  <session name="backoffice" popularity="30" ...>
  ... </session>
```

- The popularity is the frequency of this type of session. This is used to decided which session a new user will execute. The sum of all session's popularity must be 100.

- This example show several features of the HTTP protocol support in Tsunami: GET and POST request, basic authentication, transaction for statistics definition, ... The same approach can be used for defining Jabber/XMPP session.

Understanding `tsunami.xml` file: Dynamic substitutions (1/2)

- Dynamic substitution are mark-up placed in element of the scenario. For HTTP, this mark-up can be placed in basic authentication (www_authenticate tag: userid and passwd attributes), URL (to change GET parameter) and POST content.

- Those mark-up are of the form `%%Module:Function%%`. Substitutions are executed on a request-by-request basis, only if the request tag has the attribute subst="true".

- When a substitution is asked, the substitution mark-up is replaced by the result of the call to the Erlang function: Module:Function(Pid).

- Here is an example of use of substitution in a Tsunami scenario:

```
<session name="rec20040316-08:47" popularity="100"
        persistent="true" messages_ack="parse" type="ts_http">
 <request subst="true">
  <http url="/echo?symbol=%%symbol:new%%" method="GET">
  </http></request>
</session>
```

Understanding `tsunami.xml` file: Dynamic substitutions (2/2)

- **Here is the Erlang code of the module used for dynamic substitution:**

```
-module(symbol).
-export([new/1]).

new(Pid) ->
    case random:uniform(3) of
        1 -> "IBM";
        2 -> "MSFT";
        3 -> "RHAT"
    end.
```

- **As you can this, writing scenario with dynamic substitution is trivial.**

10

Example graphes: Tsunami summary

File Edit View Go Bookmarks Tab Tools Help

file:///home/nniclausse/cvs/min-fin/P1156/tests/janvier/testmontee30s/rep

Mozilla Firebird Help | Mozilla Firebird Discussi... | Plug-in FAQ

IDX-Tsunami - | IDX-Tsunami - | IDX-Tsunami - | IDX-Tsunami -

IDX-Tsunami

version 1.0.beta3

Stats Report

- Main statistics
- Transactions
- Network Throughput
- Counters
- Server monitoring

Graphs Report

- Response times
- Throughput graphs
- Simultaneous Users
- Server monitoring

IDX-Tsunami - Statistics

Main Statistics

Name	highest mean value	lowest mean value	Highest Rate
connect	0.151422 sec	0.000658711 sec	24.4 / sec
page	3.12082 sec	0.0548961 sec	177.6 / sec
request	0.240578 sec	0.000805374 sec	3259.5 / sec
session	328.313 sec	4.48013 sec	23.3 / sec

Transactions Statistics

Name	highest mean value	Highest Rate
tr_accueil	6.90009 sec	16 / sec
tr_applet	3.33463 sec	6.8 / sec

Network Throughput

Name	Highest Rate	Total
size	83364.25 Kb/sec	268589.01 MB

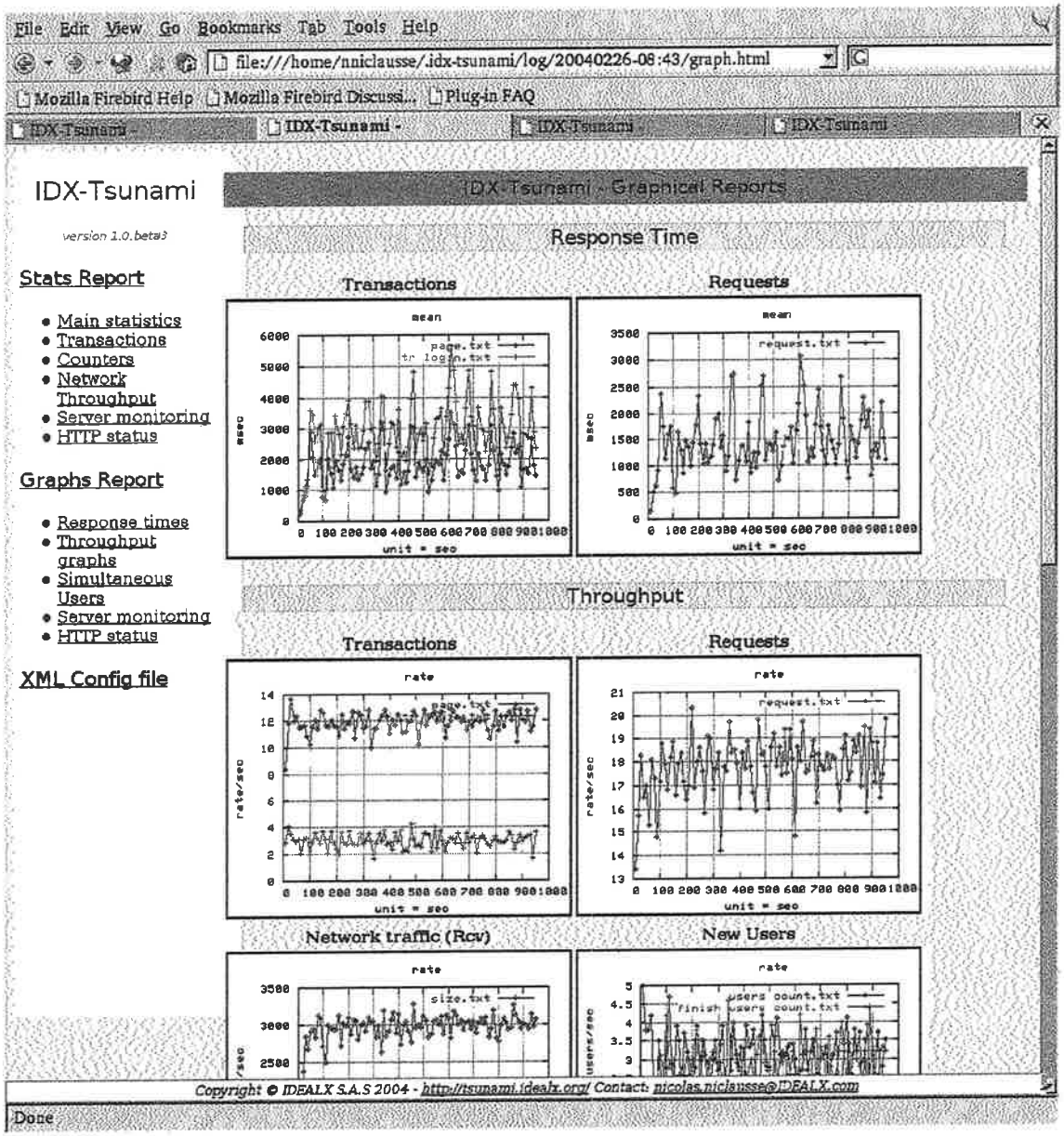
Counters Statistics

Name	Highest Rate	Total number
closed_when_send	3.7 / sec	1173
error_reconnect_econnreset	0.2 / sec	2

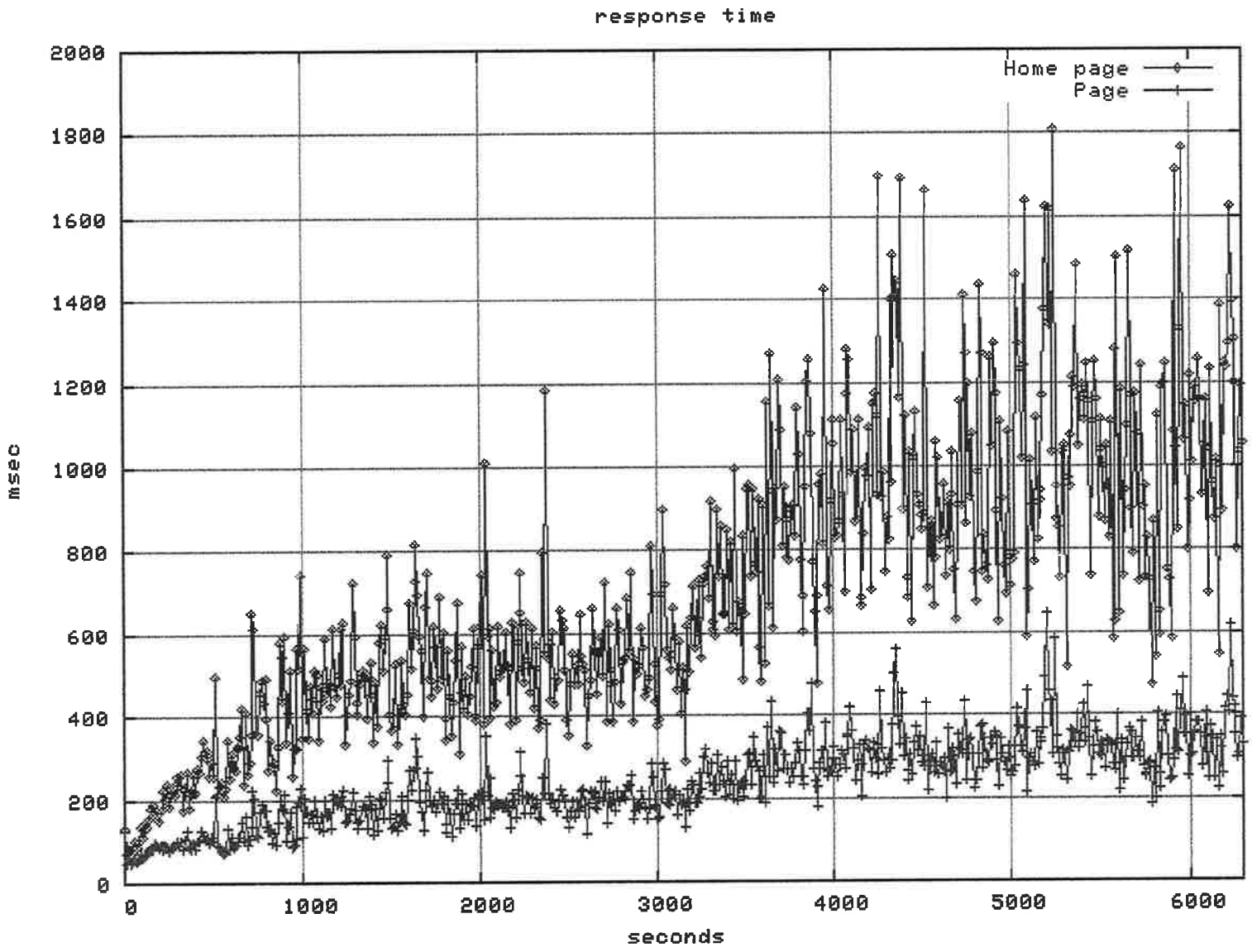
Copyright © IDEALX S.A.S 2004 - http://tsunami.idealx.org/ Contact: nicolas.niclausse@IDEALX.com

Done

Example graphes: Statistics overview

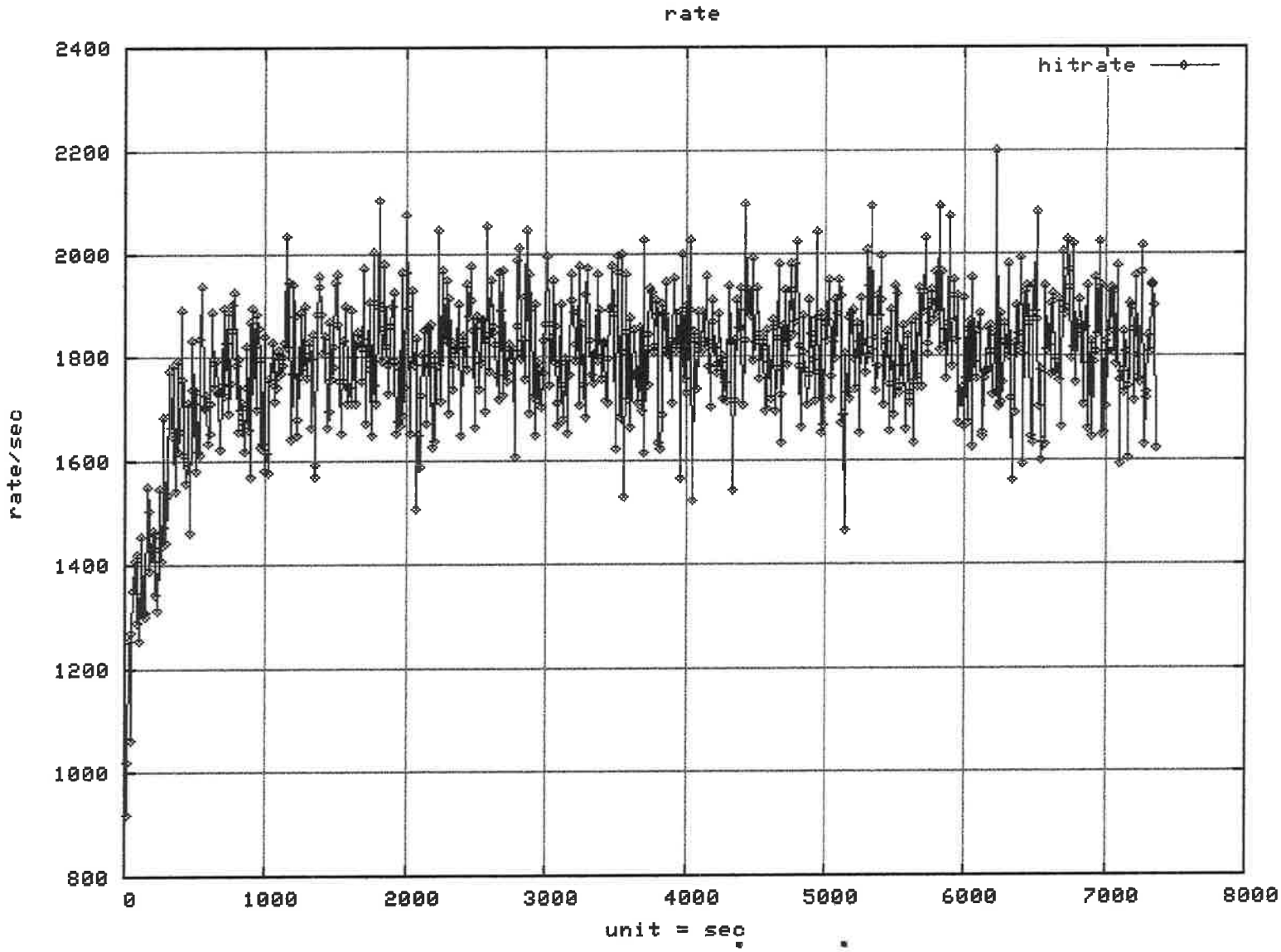


Example graphes: Response time



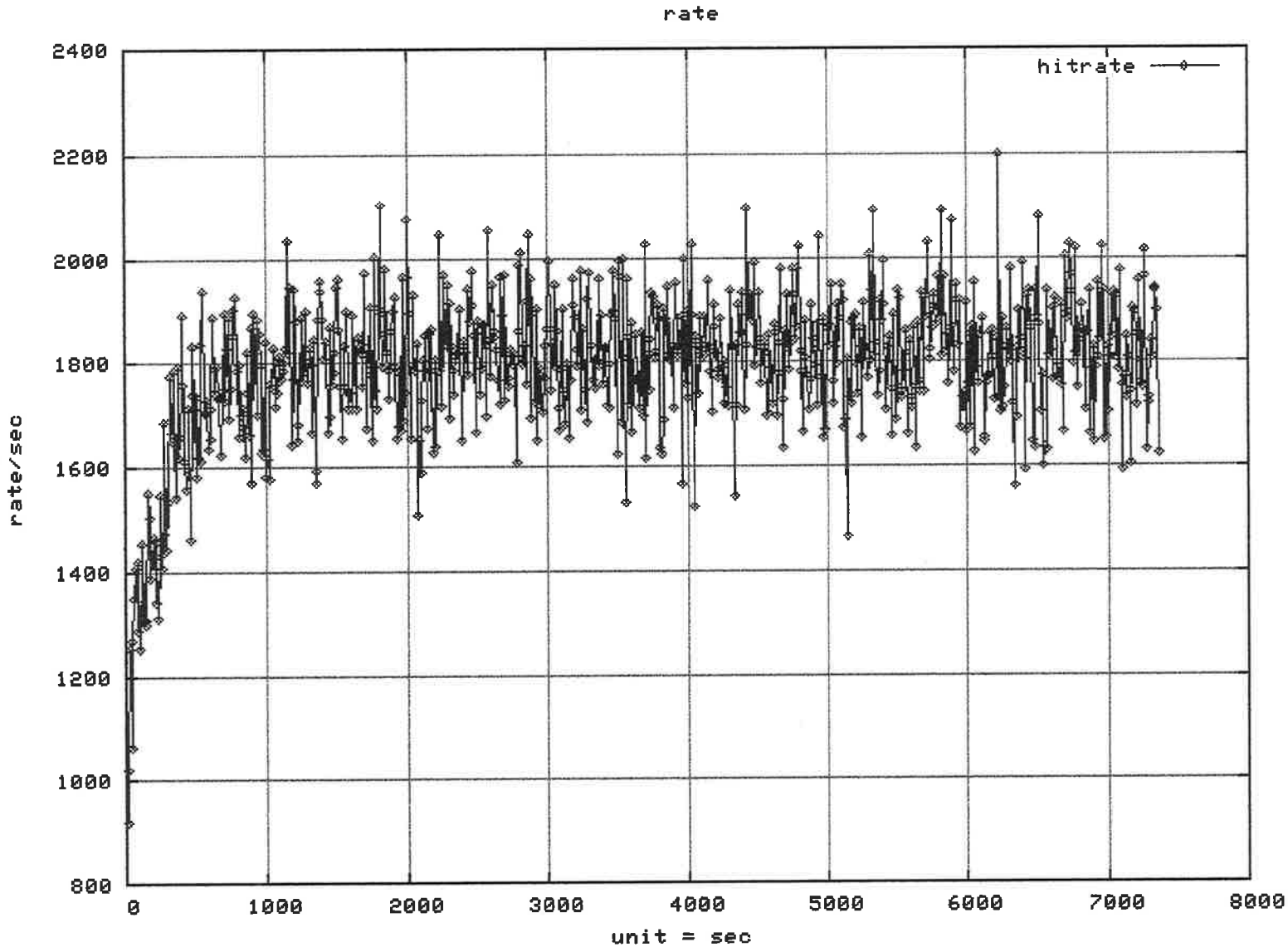
13

Example graphes: Hit rate



14

Example graphes: Network traffic



Figures and organisations using Tsunami

- **Tsunami has been successfully used for huge benchmark:**
  - Jabber protocol: 10 000 simultaneous users. Tsunami were running on a 3-computers cluster (CPU 800Mhz)
  - HTTP and HTTPS protocol: 25 000 simultaneous users. Tsunami were running on a 4-computers cluster. The tested platform reached 3 000 request per second.

- **Tsunami has been used for benchmark at:**
  - DGI (Direction Générale des impôts): French finance ministry
  - Cap Gemini Ernst & Young
  - IFP (Institut Français du Pétrole): French Research Organisation for Petroleum
  - LibertySurf
  - ...

Conclusion

- **Tsunami has several advantages over other injection tools:**
  - Outstanding **performance** and distributed benchmark
  - **Ease of use:** The hard work is already done for all supported protocole No need to write complex scripts. Dynamic scenarii only require small trivial piece of code. Tsunami scenarii realisation is mostly based on
  - **Multi-protocol support:** Tsunami is for example one of the only tool to benchmark SOAP applications
  - **Monitoring** of the target cluster to analyse the behaviour and find bottlenecks. For exemple, I did use it to analyse cluster symmetry (is the load properly balanced ?) and to determine the best combination of machines on the three cluster tiers (Web engine, EJB engine and database)

A Virtual World Distributed Server developed in Erlang as a Tool for analysing Needs of Massively Multiplayer Online Game Servers

Michał Ślaski
Erlang Training and Consulting Ltd.
London, United Kingdom
michal@erlang-consulting.com

Marcin Gazda
AGH University of Science and Technology
Al. Mickiewicza 30
30-059 Krakow, Poland

ABSTRACT
At present massively multiplayer online games allow several thousands of players to stay in a single, persistent virtual world. Because of the fast growing interest in this type of servers, we started researching their efficiency and scalability. Our target was an analysis of the MMOG server, which could service up to 1000 players in a single virtual world. We made an assumption that the server will be distributed and running on a dedicated cluster. As the implementation platform we chose Erlang/OTP its main advantages being integration with a distributed database, soft real-time and supporting distributed applications. In this paper we discuss the realisation of the project, and practical aspects of the measurement of server parameters.

Keywords
Massively Multiplayer Online Game, Erlang, Load Testing, Distributed Server.

1. INTRODUCTION
Massively multiplayer online games (MMOG) are a dynamically developing segment of the computer games industry. Even though there are many difficulties that you need to overcome while building this type of systems, there has been a significant increase in the interest in MMOG throughout the last couple of years. In the year 2004 the total income from selling online games was over 1.5 billion US dollars. In the year 2006 the total income is predicted to be twice as big.

In classic multiplayer games, like Quake, the number of players usually is under 20. At present one of the most popular MMOG in Europe and USA is World of Warcraft released by Blizzard Entertainment, in which several thousands users play on each server. MMOGs are characterised by big demands for the the system and for the network infrastructure. Thousands of players staying concurrently in the same virtual world, interacting with each other and changing the state of the game environment, generate heavy network traffic and a heavy server load.

Not many titles were as successful as the World of Warcraft. Most of them had technical faults and lacked in attractive gameplay. To provide a good level of gameplay, every user needs to have an up-to-date information about the state of the virtual world. This state changes every time when the user takes an action (like moving his character or collecting an object) leading to the conflict between capacity and coherence. It is impossible to guarantee that a dynamically shared game state will change frequently and that every user will have a permanent access to the same and most current state.

2. PROTOTYPING
The main reason of failures in creating MMO games is the fact that you can only fully test it at the end of the developing process. All of the leaks and mistakes in the design can be noticed during beta tests when thousands of users start playing it. It is important to prototype every solution in order to simulate players and check if the solution is appropriate. During the phase of prototyping you can experiment with the functional and technical aspects of the system and you should determine the architecture of the system.

2.1 Using Erlang/OTP for the prototype
Usually prototypes are developed on hardware not as strong as the target one, so it is important that the platform used for prototyping is compatible with different operation systems and it is not dedicated to a specific one. Another requirement for the platform is the possibility of a quick and easy development process. There is a need for mechanisms that can support distributed architectures, because these are the most promising directions of research. Supporting high availability and efficient internal communication are also significant. This is why Erlang Open Telecom Platform was chosen. Important features from our point of view are: open source, its own virtual machine, light processes and the distributed database Mnesia.

Handling a big amount of users requires concurrent oriented programming. This can be accomplished with Erlang processes – one process for each user. There is a need of scalability of the system, what can be realised by distributing the server on a cluster of machines. Erlang allows processes to communicate with each other by knowing only their PID and without knowing if they are run on the same node, so you can treat the cluster as one coherent system. Such persistence allows you to build prototypes effectively.

3. IMPLEMENTATION

For the sake of research a game client in Java and with Java3D library was developed. The application provides functionality similar to role playing games. Users can move their characters, collect objects, chat with each other and do some magic. Messages with the information about user actions are sent to the server and then dispatched to other connected users.

The server was implemented in Erlang. All of the game data is stored in Mnesia tables. The Mnesia's scheme was configured to keep the replicas of all tables on all nodes. Most frequently used data like socket binding or player's position are stored in ETS hash tables.

Erlang processes are organised in a supervision tree. The root process on every node is responsible for starting services assigned to this node. Services were implemented with the *gen_server* behaviour. There are two kinds of services: processing network traffic and processing user's actions. When 'network processing' service is run, the node becomes an access point and it starts to listen on a TCP port for new connections. When 'action processing' service is run, then the node takes part in distributing the computations.

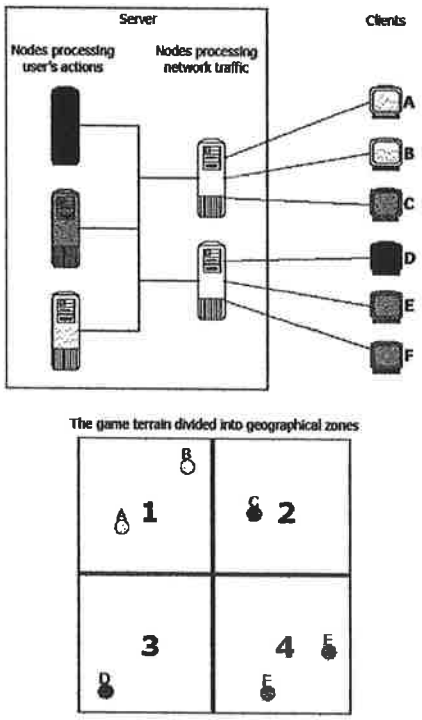


Figure 1. System's architecture

The game terrain is divided into geographic zones assigned to the nodes processing the player's actions. Every player is assigned to the zone in which he is standing. Processes controlling the players placed within the same zone are run on the same node. When the player moves to another zone, its controlling process is moved to the appropriate node. This operation needs a lot of calculations, so the algorithm was implemented to prevent often zone switches. The player is switched into another zone not when he crosses a zone border, but when he gets out of the zone range.

4. SYSTEM PERFORMANCE ANALYSIS

When the implementation of the system was finished, a game session with real players was organised. Seven people were playing for several hours so we could log all of their actions and carry out analysis on how often statistically a user takes an action. Then we generated scenarios for IDX-Tsunami which is a distributed load testing tool that can simulate TCP clients. A plug-in for IDX-Tsunami was developed to support our protocol and to log the server performance.

The research was done in two phases. During the first one two types of architectures were tested. The architecture of the first type is built by single-function nodes, which means that every node is processing either network traffic or player's actions. The architecture of the second type is built by double-function nodes, which means that every node provides both services.

For every architecture the maximum number of players that it can handle was determined. We determined the number by analysing several different indicators, such as the time in which the server replies for messages, outgoing network traffic, number of outgoing packets and the utilisation of the CPU on the server side.

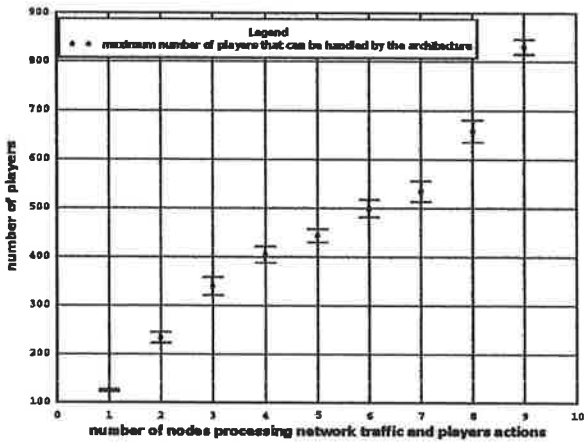


Figure 2. Architectures with double-function nodes

According to research, zone based solutions are highly dependant on the distribution of players in the world. We made an assumption that players are spread around the world equally and this kind of situation could be possible if the game world is big enough. But the fact is that when many players are standing in the same region, like during battles, the server can be overloaded.

During the second phase we worked on synchronizing issues. Let imagine that two players are looking at the same mushroom lying on the ground. One of them moves near the mushroom and picks it up. Then the message is sent to the server. Before the server passes this message to the second player, he can still see the mushroom, so he can try to pick it up. If he dose so, obviously he will fail, because the mushroom is already collected. We measured the probability that a player would or would not fail in two kinds of situations: when the server is not overloaded and when it is overloaded. In the second case players more often cannot take successful actions, because the state of the world which they can observe is desynchronized from the state on the server.

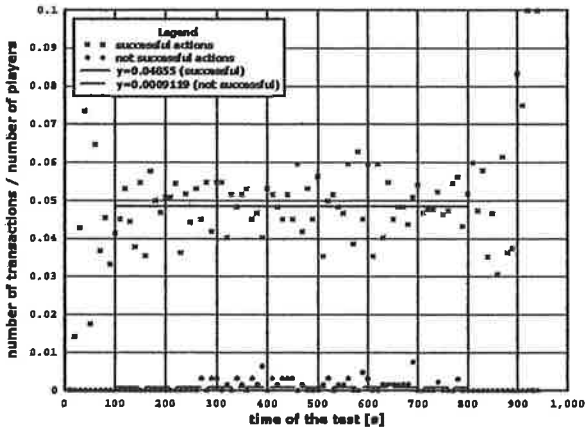


Figure 3. Server not overloaded, more successful actions.

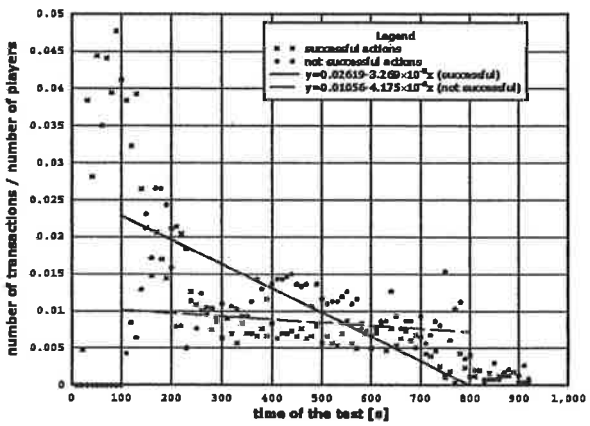


Figure 4. Server overloaded, more actions fail.

5. CONCLUSION

To summarize we found the Erlang Open Telecom Platform very appropriate for developing prototypes of the distributed MMOG system. Because of the fact that you can quickly build a solution and then experiment with it, you can examine different algorithms in a relatively short period of time.

6. ACKNOWLEDGMENTS

University supervisor: Stanisław Ciszewski, AGH University of Science and Technology, Kraków, Poland.

Nicolas Niclausse, author of the IDX-Tsunami tool, which was used for the players' simulation.

7. REFERENCES

[1] J. Armstrong, R. Virding, C.Wikstrom, M. Williams, Concurrent Programming in Erlang. *Prentice-Hall*, 1996.

[2] A.G. Bosser, Massively Multiplayer Online Games: matching Game Design with Technical Design, *IMAGINA*, June, 2004.

[3] IGDA Online Games SIG, 2004 Persistent Worlds Whitepaper, *http://www.igda.org/online/*, December, 2004.

[4] B. Knutsson, H. Lu, W. Xu, B. Hopkins, Peer-to-Peer Support for Massively Multiplayer Games, *INFOCOM 2004*, March, 2004.

[5] D. Saha, S. Sahu, A. Shaikh, A Service Platform for On-Line Game, *Proceedings of NetGames 2003 Workshop*, May, 2003.

[6] MMOGCHART.COM, *http://www.mmogchart.com*

[7] K. Milligan, Massively successful - MMORPGs come of age, *http://keathmilligan.net/view.php?id=448*, December, 2004.

[8] IDX-Tsunami distributed load testing tool, *http://tsunami.idealx.org/*

Third Party Gateway

EUC 2005

Chandrashekhar Mullaparthi
T-Mobile (UK)

TPG - What is it

TPG is a third party gateway.

- Bulk MT-SMS
- Premium MT-SMS
- Receive MO-SMS
- Location Based Services

EUC 2005

Chandrashekhar Mullaparthi
T-Mobile (UK)

Why did we do it?

- Replace a failing IT system
  - Slow response times
  - Lots of unwanted features
  - Missing wanted features
  - Unstable
  - Expensive
  - Built using Weblogic, Java, DB2

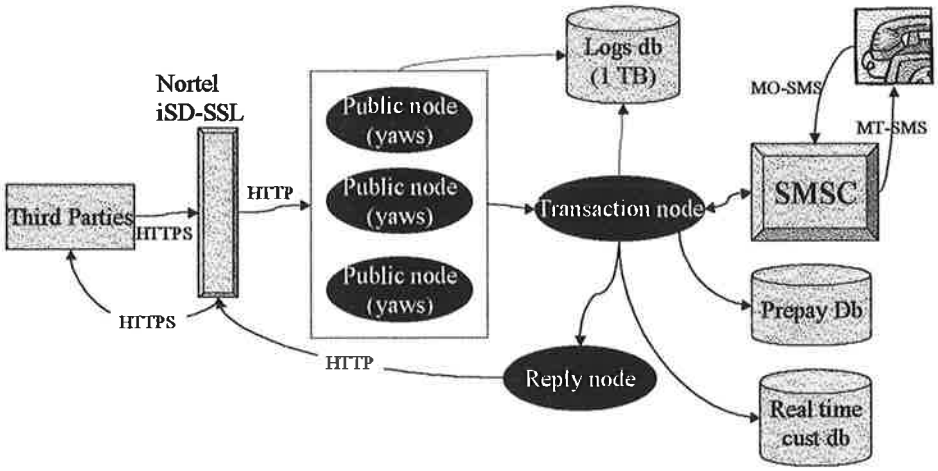
Chandrashekhar Mullaparthi
T-Mobile (UK)

How did we do it?

- 1st version in 2002
  - Written in 3 weeks (warts and all)
- 2nd version in 2003
  - Half hearted attempt to win the political battle
- 3rd version in 2004
  - Political battle won!
  - Got rid of most of the warts (except XML)

Chandrashekhar Mullaparthi
T-Mobile (UK)

Architecture



Chandrashekhar Mullaparthi
T-Mobile (UK)

Some Details

- yaws as the front end webserver
- XML parsing using eXpat (http://www.libexpat.org/)
- ibrowse as the HTTP client
- mnesia based message queues
- Configurable throughput for each 3rd party
- All events logged in an external 1TB Oracle DB
- Public node uses Sun Fire V120
- Trx node uses Sun Netra T4 (dual processor)
- Business Logic: nearly 11000 lines of code
- Libraries: nearly 19000 lines of code

Chandrashekhar Mullaparthi
T-Mobile (UK)

ibrowse

http://jungerl.sourceforge.net

- My first contribution to the erlang community!
- RFC2616 compliant (AFAIK)
- HTTP/0.9, HTTP/1.0, HTTP/1.1
- Understands chunked encoding
- Named pools of connections to each webserver
- Pipelining support
- Download to file
- Asynchronous requests. Responses are streamed to a process
- Basic authentication
- Proxy authentication
- Can talk to Secure webservers using SSL
- ToDo - Use inets driver HTTP parsing
- ToDo - Support the CONNECT method

Chandrashekhar Mullaparthi
T-Mobile (UK)

Performance

Traffic Level

Availability

RTT: 120ms

Chandrashekhar Mullaparthi
T-Mobile (UK)

Effort

- 1st version developed by me in 3 weeks
- I've forgotten what I did in the 2nd version
- 3rd version
  - Updated to meet all "known" requirements
  - Kept updating as new requirements were "discovered"
  - Performance tuned with Sean's help
  - Peter Lund developed a load tool
  - Francesco Cesarini took over from me and did load testing, bug fixing and training of support staff. Lots of everything!
  - Chris Newman helped with end to end testing and SSL troubleshooting
  - Peter Whitaker held endless meetings with support to get TPG accepted
  - Tammy Saunders helped setup a database for logs produced by TPG and made a poster to inspire everyone!
  - Haider Mohammed was brave enough to take ownership of TPG!

Customer comments

- "You have gone from the worst performing network [out of 40 networks] to the best" - WIN (verbal quote)
- "We have achieved a 100% improvement in performance and we have not runup as any connections as we had before" - WIN CTO John Rands
- "Requests now take between 60 and 100ms. This compares well against ATK (very rarely see a transaction < 500ms) and is approaching Grouse (where most transactions tend to take 20-50ms)" - MX CTO Chris Wilson

eXAT: Software Agents in Erlang

Corrado Santoro
University of Catania - Engineering Faculty
Department of Computer and Telecommunication Engineering
Viale Andrea Doria, 6 - 95125 - Catania, Italy
EMail: csanto@diit.unict.it

***Abstract*— This paper describes eXAT, a new agent programming platform to write and execute agents using the Erlang language. The main characteristic of eXAT is that it provides an "all-in-one framework" for the design, with a single tool, of *agent intelligence*, *agent behavior* and *agent communication*. This is made possible by means of a set of modules strongly tied to one another: (*i*) an Erlang-based rule-processing engine, (*ii*) an execution environment for agent tasks, based on object-oriented finite-state machines, and (*iii*) a module able to handle FIPA-ACL messages. Agent tasks are coupled with rule-processing engines in order to support transition triggering on the basis of agent's mental state. Moreover, the agent communication facility provided by eXAT can not only trigger task's events but also influence agent's mental state according to FIPA-ACL semantics.**

Index Terms—Software Agents, Agent Programming Platforms, FIPA, Inference Systems, Ontologies.

1 Introduction

To date, agent technology [40] is becoming widely used as an interesting approach to build autonomous software systems. Many agent programming platforms and tools have been developed [4, 10, 1, 8, 38, 32, 34, 35, 7], aiming at providing an execution environment for agent-based applications, together with a set of libraries for agent developers.

In order to offer a standardized and cross-platform environment, the majority of such tools are developed in Java, while few of them employ *ad-hoc* programming languages. However, the use of Java is able to support only some aspects of agent-oriented programming, while other aspects, such as the intelligence, require external tools.

Let us remind that, by definition [31, 40, 41], an "agent" is a software entity that, while situated in an environment, *reacts* to environmental changes and *elaborates plans* to be executed in order to achieve a specific goal; an agent can also *interact* with other agents, if such interactions provide help in achieving the goal. From the developer point of view, this implies three specific requirements that should be met by an agent programming platform or language:

a. *Ability of specifying and implementing the reactive behavior of an agent.* This is in general supported by a modeling based on finite-state machines.

b. *Ability of specifying and implementing agent reasoning*, which can be supported by artificial intelligence tools such as expert systems, rule-processing engines, etc.

c. *Ability of supporting interaction with other agents*, by means of suitable message exchange abstractions.

In such a scenario, imperative and object-based capabilities of Java are suitable to support FSMs and interactions [10, 1, 8], but fail to take into account "agent intelligence": as we argued many times [15, 17, 18], the Java language does not seem an appropriate choice for agent system implementation since it is not able to offer native statements to express logic construct like predicates or production rules. In fact, each time a reasoning process has to be included in an agent system developed with a Java platform, additional tools are introduced [3, 2, 36], which, however, use logic/declarative languages that differs from Java in both syntax and semantics. The result is a mixing of programming approaches that does not help the developed because s/he is forced not only to deal with two completely different languages but also to handle data exchanging between the different language domains.

Following such statements, we found that Erlang, thanks to two main features, pattern matching in function clause declaration and handling of symbols in data, is instead a good candidate to offer a complete solution that takes care of all the aspects of agent programming. On this basis, we developed the eXAT[1] platform [5] with the aim of providing an "all-in-one" environment that considers together the three main aspects of agent-oriented programming: *behavior*, *intelligence* and *semantic and syntactic interoperability*. This paper, which is a synthesis of our previous works [12, 14, 13, 15, 17, 18, 16, 11], describes the eXAT platform, focusing on its internal structure and functionalities. After a brief discussion about the motivations that led us to choose Erlang, the internals of eXAT will be presented, together with some code samples that will show how to use the various modules provided by eXAT to build a complete multi-agent system.

The paper is structured as follows. Section 2 illustrates the reasons for choosing Erlang for agent system implementation (and thus the reasons why we developed eXAT). Section 3 gives a brief overview of the eXAT platform. Section 4 describes the agent behavior model and the abstraction provided to write agent's tasks. Section 5 focuses on agent intelligence by presenting the rule-processing engine included in eXAT. Section 6 deals with agent interaction and illustrates the tools and modules to handle message exchanging, from both the syntactic and semantic point of view. Section 7 concludes the paper.

[1] erlang eXperimental Agent Tool.

```
1  -module(reactive_agent).
2
3  agent_loop() ->
4    E = wait_for_next_event(),
5    act(E),
6    agent_loop().
7
8  act({switch, on}) ->   % act when switch is turned on
9  act({switch, off}) -> % act when switch is turned off
10 act({temperature, X}) when X > 30 ->
11   % act when the temperature is greater than 30
12 act({temperature, X}) when X < 20 ->
13   % act when the temperature is less than 20
14 act(_) ->   % unknown event, no action
```

Figure 1: A simple pure-reactive agent in Erlang

2 Why Erlang?

The main reasons that led us to choose Erlang as a possible language for the development of agent systems derive from the basic properties of agents listed in [40]—*reactivity*, *pro-activeness* and *social ability*; each of this property is analyzed to evaluate if—and how—it can be supported by Erlang.

2.1 Reactivity

An agent has the basic capability of reacting to incoming events. They include e.g. a change of the state of the reference environment, the arrival of a messages from the user or other agents, the occurrence of exceptional conditions, etc. An event can be considered featured by a *type* and *additional data* bound to the event itself (e.g. for an incoming message, the additional data could be the payload) and, on this basis, suitable predicates on bound data can discriminate various reaction cases to events of the same type.

From the programmer's point of view, reacting to events implies to provide (*i*) an abstraction for modeling events and (*ii*) some constructs or library calls to specify the computation to be triggered when a particular event occurs, also given that the bound data could be subject to certain conditions. Erlang seems particularly suitable to face such requirements for the following reasons:

1. Erlang is a symbolic language (like Prolog or LISP), and it is known that the use of *literal symbols (atoms)* facilitates the representation of constants in data. Structured information can be represented as *tuples* and, since they are untyped, are well-suited for heterogeneous data [39] and thus particularly appropriate for event types that could be very different one another. For example, the state of a switch can be represented as `{switch, on}` or `{switch, off}`, a sensed temperature with `{temperature, 25}`, an incoming message as `{message, 'QUERY-IF', {sender, 'UserAgent'}}`, etc.

2. Erlang is a functional language and functions can have multiple clauses. Matching on function definition can be exploited to specify the computation to execute following an incoming event formed as desired: function clause declaration will specify the matching criteria relevant to a triggering event, while function body will implement the associated action.

```
rule(Engine, {'child-of', X, Y}, {female, Y}) ->
  eresye:assert(Engine, {'mother-of', Y, X});

rule(Engine, {'child-of', X, Y}, {male, Y}) ->
  eresye:assert(Engine, {'father-of', Y, X}).
```

Figure 2: Some Erlang function clauses expressing inference rules

The example in Figure 1 shows a practical usage of the concepts indicated above. The listing in the Figure reports a possible implementation of a (very simple) pure reactive agent programmed in Erlang. Agent's main loop (function `agent_loop`, lines 3–6) waits for an incoming event and then executes the associated action; computations tied to events are specified by using multiple clauses of the function `act`, each one matching a different value of the parameter: when the function is invoked using the event acquired (line 5), only the matching clause is activated (if one exists, otherwise the default clause—line 14—is chosen). As the reader can appreciate, using symbols, structured data and functions with several clauses improve not only engineering and implementing reactive agents, but also the readability of the source code.

2.2 Pro-Activeness

Pro-activeness means the capability of an agent to develop and execute *plans* in order to achieve a specific goal. Unless specific BDI[2] tools are employed [7, 36], such an ability is generally supported by means of a rule production system [3, 2, 6, 11], featuring a *knowledge base* and a set of *inference rules*. In this context, Erlang's features are particularly interesting for the following reasons:

1. Symbols and primitive types (i.e. atoms and tuples) are well suited to represent *facts* of a knowledge base; moreover the use of the same types for facts and events (i.e. tuples) facilitates agent design, allowing programmers to direct use event data in the knowledge base.

2. Function clauses, which indeed represent *predicates* on parameters that if matched activate the clause, fit well in the representation of the *precondition* part of a rule; at the same time, the function body can represent the *action* part.

Note that despite Erlang's capability to represent rules, the language and run-time system do not include an engine for rule processing, which has to be provided by an external tool. For this reason, the ERESYE system has been designed by the authors [11] and it has been included in the eXAT platform. ERESYE is an Erlang-based rule production system featuring the same characteristics (from both the syntactic and semantic point of view) of other well-known similar tools, such as OPS5 [20, 21], CLIPS [3], Jess [2], etc.

The example in Figure 2 gives a sketch of Erlang function clauses used as rules of an ERESYE inference system. In the example, the rules shown permit to enrich the knowledge by deriving the concepts of `'father-of'` and `'mother-of'`, on

[2]BDI means "belief-desire-intention" and it is one of the most widely accepted paradigms for rational agents.

the basis of the knowledge of the `'child-of'` and "gender" concepts.

2.3 Social Ability

Agent-oriented engineering is based on subdividing a whole application into a set of *goals* to be achieved by several cooperating agents; thus the possibility of supporting interaction among agents is a mandatory functionality of any agent programming language or platform. As it is known, the Erlang language and its run-time system have been explicitly designed to support communication; moreover, the Erlang programming model [9, 17] is based on subdividing a problem into a set of tasks to be assigned to the same number of *concurrent processes* that *share nothing* and interact each other only by means of *message passing*. The reader can appreciate the similarity between this model and the basics of multi-agent systems [40]: Erlang concurrency model and interaction constructs seem thus perfect "as-is" to support interactions among (Erlang-programmed) agents. The only concern is with the exchanging protocol and data representation, which is Erlang-proprietary and thus non-standard (even if it is documented). An agent platform is thus needed when standard messaging, as in FIPA[3], is required to favor the interoperability with different platforms and agents written in other programming languages.

3 Overview of eXAT

The eXAT platform [12, 13, 14, 15, 18, 17, 11] has been designed with the objective of providing an "all-in-one" environment to execute agents and to program them in their *behavioral (reactive)*, *intelligent (pro-active)* and *cooperative (social)* parts, all with the same language (Erlang).

Agent behaviors can be programmed by specifying *tasks* modeled as finite-state machines (FSMs), enriched with the possibility of using *composition*, i.e. serial and parallel execution of sub-FSMs, and *extension*, i.e. refining some parts of an existing FSM (according to the concept of virtual inheritance proper of the object-oriented technology) in order to support new requirements. Task model and programming are detailed in Section 4.

Agent intelligence is instead programmed by means of rule-based code, supported and executed by the ERESYE tool (as briefly illustrated in the Section 2). An ERESYE engine, together with its programmed rules, can be bound to an agent of the platform in order to support agent's inference: the knowledge base of the engine can thus represent agent's mental state, while production rules support agent's reasoning process. ERESYE engine's events can be bound to behaviors, thus allowing reasoning processes to also trigger user-defined agent actions. The details of ERESYE are reported in Section 5.

Agent interaction is performed by means of the exchange of FIPA-ACL [24] messages[4]; this is supported by eXAT's modules that include library functions to send and receive messages, encoding them through (user-defined) ontologies. Message exchanging is mainly connected to behavior execution thus making possible the occurrence of a proper event when a new message is delivered to the agent. But message exchanging is also able to influence agent's mental state thanks to the support of *FIPA-ACL semantics*: an incoming message is processed by the ACL semantics module and, according to the *meaning* it carries (as defined in FIPA-ACL standard [24]), suitable actions are performed on the knowledge base of the ERESYE engine bound to the receiving agent. This allows for the implementation of "more rational" multi-agent systems. Details are provided in Section 6.

[3]FIPA, which means "Foundation for Intelligent and Physical Agents" is a non-profit IEEE organization for the standardization of agent technology [30].

[4]FIPA-ACL (Agent Communication Language) is a standardized interaction language for agents.

4 Writing Agent Tasks

4.1 Basic Task Model

The behavioral part of an agent is programmed in eXAT by means of one or more *tasks*, expressed as finite-state machines (FSM). The FSM model used in eXAT is an extension of the basic model that maps the occurrence of an event in a given state to an action and a new state, i.e.:

$$(Event, CurrentState) \rightarrow (Action, NewState)$$

In eXAT, as introduced in Section 2.1, an event is characterized by a *type* and additional *information* (event data) bound to the event itself. On this basis, a FSM, or agent task, is modeled with the following elements:

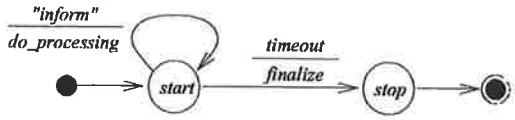
- E is the set of *event types*. The event types handled by eXAT are:
  - *acl*, the reception of an ACL message;
  - *timeout*, the expiry of a given timeout;
  - *eresye*, the assertion of a particular fact in an ERESYE engine;
  - *silent*, the silent event.
- P is the set of *data patterns* to be bound to a certain event type. A *data pattern* specifies a template to be matched with event's data, for the event to be able to trigger a transition.
- S is the set of *states* of the FSM.
- A is the set of *actions* to be done.
- $f : S \times E \times P \rightarrow A \times S$ is the transition function that maps an event occurring with a given pattern and in a certain state to an action execution and a new state of the FSM.

An agent task is specified in a single Erlang module (the name of the module becomes also the name of the task) that implements the following three main functions:

```
action(State) -> [{EventName, ActionFunc}, ... ]
event(EventName) -> {EventType, PatternName}
pattern(PatternName) -> PatterSpec
```

Function `action` specifies the transitions exiting from the state given as parameter; it returns a list of couples *event name* and *action function*, meaning that, at the occurrence of that event, the associated Erlang function will be executed[5]. The other two functions, `event` and `pattern`, are used to fully specify the event bound to a certain transition. An event is characterized by a *type* and a *template* (or *pattern*) that must be

[5]The state reached by the FSM after the occurrence of an event is encoded in the *ActionFunc* and, for this reason, it does not explicitly appear here.



(a)

```
-module(first).
-export([action/2, event/2, pattern/2,
         do_processing/4, finalize/4]).
-include("acl.hrl").

action(Self, start) ->
  [{new_message_event, do_processing},
   {timeout_event, finalize}].

event(Self, new_message_event) ->
  {acl, inform_pattern};
event(Self, timeout_event) ->
  {timeout, timeout_pattern}.

pattern(Self, inform_pattern) ->
  [#aclmessage {speechact = 'INFORM'}];
pattern(Self, timeout_pattern) -> 10000.

do_processing(Self, EventName, Message, ActName) ->
  % Perform processing ....
  object:do(Self, start).

finalize(Self, EventName, Data, ActName) ->
  % Finalize behaviour ....
  object:do(Self, stop).
```

(b)

Figure 3: A Simple Task in eXAT

matched by the data associated to the event in order to activate the transition. The former function associates to each event name its type (chosen among *acl*, *timeout*, *eresye* and *silent* as reported above) and a *pattern name*. Function **pattern** then maps each pattern name with the relevant matching template, whose structure depends on the type of the event itself: for an *acl* event, the template is specified by indicating matching values in the various field of a **#aclmessage** record (see Section 6); a *timeout* event requires a value in milliseconds; event *eresye* requires the specification of the template of the fact to be matched (see Section 5).

As a first example, the FSM depicted in Figure 3a shows a tasks that executes action "*do_processing*" each time a new "*inform*" message is received, unless a timeout of ten seconds occurs. This task can be implemented, in eXAT, by means of the module **first** reported in Figure 3b. The reader can note the use of functions **action**, **event** and **pattern**, and the way in which the concrete actions can be implemented. As it can be noted, function **object:do** is used to set the next state after action execution.

4.2 Composing Tasks

Depending on the application to be realized, agent tasks could be very complex and require FSMs composed of a large number of states and transitions; such cases could be hard to handle during development stage. As it is widely known, the use of *modularization*, i.e. the possibility of decomposing a large FSM into a set of smaller FSMs, helps the designer in tackling these situations. In addition, modularization favors reuse, as there could be cases in which parts of an overall agent task could be reused in another different agent application[6]. To face these situations, eXAT allows a designer to engineer an agent by composing tasks *in sequence*—to support serial activities—or *in parallel*—to support multiple concurrent activities. This is done by exploiting function **behave** (exported by the **agent** module), which, when called in the body of an action implementation, causes the execution of the specified task(s). The function takes, as parameter, either a single task name or a list of tasks names; in the latter case, all the specified tasks are executed in parallel[7].

As an example, Figure 4a illustrate a FSM that:

- starts the "*english-auction*" task, if it receives an "*inform*" message; or
- starts the "*dutch-action*" task, if a timeout occurs; and
- in any case, after executing one of the (sub-)tasks, it stops.

The implementation is reported in Figure 4b.

4.3 Specializing and Extending Tasks

Composing tasks according to the concepts dealt with in the SubSection above improves agent engineering a lot. However, in some cases reusing an existing task "as-is" is not enough, because the implementation could be not so general to allow its direct use in other contexts. In such a situation, an implemented task should be modified in some elements or, in other words, *specialized* for a new purpose. As it widely known, the object-oriented technology makes specialization possible thanks to *virtual inheritance*; the same concept is exploited in eXAT to support *task extension* and, in particular, to permit:

a. Adding new states and transitions;

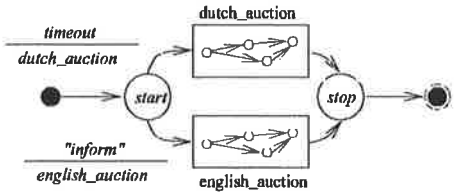
b. Removing existing states and/or transitions;

c. Modifying existing states and/or transitions by changing

   1. the state reached by a transition,

[6] As in using standard FIPA interaction protocols [25, 28, 27, 26].

[7] The function is synchronous, that is, it waits for the complete execution of the given task(s) before returning to the caller.



(a)

```
-module(second).
-export([action/2, event/2, pattern/2,
         do_english_auction/4, do_dutch_auction/4]).
-include("acl.hrl").

action(Self, start) ->
  [{first_event, do_english_auction},
   {second_event, do_english_auction}].

event(Self, first_event) -> {acl, inform_pattern};
event(Self, second_event) -> {timeout, timeout_pattern}.

pattern(Self, inform_pattern) ->
  [#aclmessage {speechact = 'INFORM'}];
pattern(Self, timeout_pattern) -> 10000.
  % Wait ten seconds.

do_english_auction(Self, EventName, Data, ActionName) ->
  % behaviour 'english_auction' is executed
  agent:behave(Self, english_auction),
  % stops current behaviour when the 'english_auction' is over
  object:do(Self, stop).

do_dutch_auction(Self, EventName, Data, ActionName) ->
  agent:behave(Self, dutch_auction),
  object:do(Self, stop).
```

(b)

Figure 4: Composing Tasks in eXAT

2. the action function bound to a transition,
3. the event type bound to a transition,
4. the data pattern bound to a transition,
5. one or more elements of a data pattern.

In concrete, task extension is made possible in eXAT thanks to the provided **object** module, whose first aim is the introduction of object-orientation in Erlang programs; it is intended for writing *class/modules with attributes and methods*, featuring virtual inheritance as in Java or C++. The provided object model is very close to that of Java. A class is declared and implemented in a single Erlang module, which has to export function **extends** that returns the name of the ancestor class/-module[8]. Then, functions of the module can be treated as *methods* by adding another parameter, called **Self**, in function declaration: this parameter represents the object's instance within which the method is invoked and plays the same role of keyword **this** in C++ and Java. According to Erlang style, a method can have multiple clauses and guards and, unlike other traditional object-oriented languages, methods feature a fine grained overriding model: we can override all clauses of a method (the whole method), a single clause of a method, or even add another clause to a method defined in the ancestor class. This characteristic provides a very flexible and expressive programming environment.

Task engineering in eXAT exploits this Erlang-based object-oriented programming capability: each task is indeed a *class*, all defined functions, i.e. **action**, **event**, **pattern** and the functions implementing the actions, are methods[9], and task extension is performed by deriving a class/module and accordingly overriding one or more methods or method clauses. In details, task specialization implies to change the return value of the interested function or function clause, i.e. to modify

a. The couple {*event, action*} bound to a certain state—if the function is **action**;

b. The couple {*event type, pattern*} defining a certain event—when function **event** is considered;

c. The specification of a given pattern—through redefinition of function **pattern**.

For example, if we would design a task behaving like that in Figure 4a, but using a "*confirm*" message instead of a timeout to trigger the Dutch auction (see Figure 5a), we can use the code reported in Figure 5b.

5 Adding Intelligence to Agents

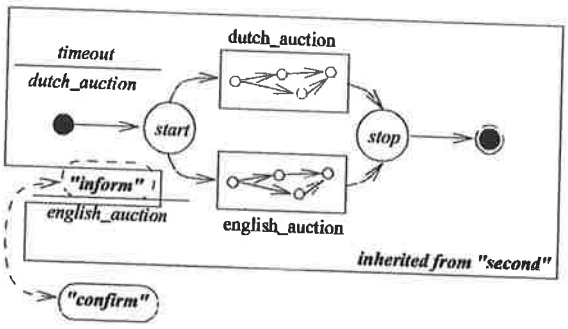
eXAT tasks are designed for the development of the reactive part of an agent, but, as stated in Section 1, agents also feature "intelligence" that has thus to be supported by a suitable AI tool. To this aim, we chose to include, in eXAT, a reasoning system, called ERESYE [11], which is able to allow the creation, management and execution of *rule-based processing engines*. Such engines can be connected with tasks in order to provide an agent programming and execution environment where the behavioral part is strictly coupled with the intelligence. In the following, an overview of ERESYE is first provided; then the way in which ERESYE engines can be integrated with agent's tasks is discussed.

5.1 Overview of ERESYE

ERESYE is an Erlang tool for programming and executing rule-processing engines. Each engine is featured by a *name* and a *knowledge base (KB)* made of a *fact base (FB)*, storing the set of

[8]This function may be not declared if the class/module has no ancestors.

[9]This is the reason why the sample codes in Figures 3 and 4 report function declarations with **Self** as the first parameter.



(a)

```
-module(third).
-export([extends/0, event/2, pattern/2]).
-include("acl.hrl").

extends() -> second.

event(Self, second_event) -> {acl, confirm_pattern}.

pattern(Self, confirm_pattern) ->
  [#aclmessage {speechact = 'CONFIRM'}].
```

(b)

Figure 5: Extending Tasks in eXAT

```
-module(buyer_intelligence).
-export([out_of_balance/2, preference_rule/3,
         purchase_rule/4, start/0]).

out_of_balance(Engine,
               {money, Agent, X})
             when X < 100 ->
  io:format("Warning! ~p is out of balance~n",
            [Agent]),
  eresye:assert(Engine, {out_of_balance, Agent}).

preference_rule(Engine,
                {interest, Agent, Item, high},
                {availability, Item, Avail})
             when Avail > 0 ->
  eresye:assert(Engine,
                {interested, Agent, Item}).

purchase_rule(Engine,
              {interested, Agent, Item },
              {price_of, Item, Price},
              {money, Agent, M})
            when (M - Price) > 1000 ->
  eresye:assert(Engine,
                {intend, Agent, {buy, Item}}).

start() ->
  eresye:start(buyer_engine),
  eresye:add_rule(buyer_engine,
          {buyer_intelligence, out_of_balance}),
  eresye:add_rule(buyer_engine,
          {buyer_intelligence, preference_rule}),
  eresye:add_rule(buyer_engine,
          {buyer_intelligence, purchase_rule}).
```

Figure 6: A Simple Reasoning Process ERESYE

facts representing the current knowledge, and a *rule base (RB)*, storing the set of inference rules representing the reasoning capability of the engine. Each fact is written in the form of an Erlang tuple, e.g. `{temperature, 50, 'F'}`, `{alarm,on}`, `{buy, 'Computer'}`, `{interested, 'Alice', 'Computer'}`; records can be used as well. In such a scenario, tuples or records are useful to represent *concepts*, e.g. `{interested, A, I}` can mean that agent `A` is interested in item `I`, `{money, A, M}` can mean that current amount of money of agent `A` is `M`, etc.

Inference rules, which represent the actions to be executed by the engine when one or more particular facts are asserted in the FB, are instead written using standard Erlang functions. An ERESYE rule is implemented with an Erlang function clause where the first parameter represents the engine name in which the rule is executed and the other parameters are tuples representing the templates of the facts that must be asserted in the FB for the rule to be activated. Guards can also be specified, thus creating additional conditions to be met in order for the rules to be fired. The body of a rule implements the action to be executed when the rule is fired; it can contain any Erlang expression, as well as calls to functions for KB manipulation. To this aim, a suitable set of functions of the ERESYE API allows Erlang programs to interact with an ERESYE engine in order to assert a fact, retract a fact, wait for the presence of a fact with a given pattern, add a new rule, change rule priority, delete a rule, etc.

Figure 6 shows a simple reasoner that uses ERESYE. Here, functions `out_of_balance`, `preference_rule` and `purchase_rule` are the rules. This means that, for example, rule `out_of_balance` will be fired when the fact represented by the tuple `{money, Agent, X}`, with $X < 100$ will be asserted; the action will consists in printing a warning message and then asserting the fact `{out_of_balance, Agent}` in the engine. The sample listing shows also the way in which an ERESYE engine is created and activated; to this aim, function `start` first performs engine instantiation and then adds to the engine the rules defined by the functions.

5.2 Tasks and Intelligence

In order to allow the interaction between agent behavior and agent intelligence, eXAT tasks can be connected with ERESYE engines. This is performed by means of a twofold mechanism.

On one hand, a transition of a task can be activated following the assertion of a fact with a given template. This is performed by specifying, in a task, an event of the *eresye* type, while the associated function `pattern` indicates (*i*) the template of the fact to be waited for, (*ii*) the name of the ERESYE engine and (*iii*) if the fact, once the event has been triggered, must be retracted from or left in the fact base of the engine.

On the other hand, a task can perform any operation onto an engine by using, in the body of its action functions, the function of the ERESYE API.

Figure 7 reports an example of an agent task that is connected with the engine in Figure 6; first, the `buyer_engine` is instantiated in function `on_starting` (which is a callback executed automatically when the task is started); then, the task behaves with a single state and two transitions. The first transition (`inform_event`) is activated when an "*inform*" message is received by the agent; the action performed, in this case, is the direct assertion of the message content in the `buyer_engine`.

```
-module(buyer).
-export([action/2, event/2, pattern/2
         get_inform/4, perform_purchasing/4,
         on_starting/1]).
-include("acl.hrl").

action(Self, start) ->
  [{inform_event, get_inform},
   {eresye_event, perform_purchasing}].

event(Self, inform_event) ->
  {act, inform_pattern};
event(Self, eresye_event) ->
  {eresye, intention_pattern}.

pattern(Self, inform_pattern) ->
  [#aclmessage { speechact = 'INFORM'}];
pattern(Self, inform_pattern) ->
  {buyer_engine, get, {intend, '_', '_'}}.

get_inform(Self, Event, Message, Action) ->
  eresye:assert(buyer_engine,
                Message#aclmessage.content),
  object:do(Self, start).

perform_purchasing(Self, Event,
                   {intend, _, {buy, Item}},
                   Action) ->
  agent:behave(fipa_request_protocol),
  object:do(Self, stop).

on_starting (Self) ->
  buyer_intelligence:start().
```

Figure 7: An agent's task that uses the reasoner of Figure 6

The second transition (**eresye_event**) is activated when a fact with pattern {**intend**, _, _} is asserted in **buyer_engine**; the action, in this case, is to start a sub-task implementing the FIPA request protocol [28], and then stopping.

6 Making Agents Interacting

6.1 FIPA-ACL Background

A key aspect of software agents is their ability to communicate in order to reach their goals; such a communication must be done using messages formed in such a way as to allow interacting agents to understand each other. This is achieved by means of three mechanisms:

1. A common way to represent data in messages (*message syntax*).
2. A common network protocol to exchange such messages between agents (*message transport*).
3. Sharing the meaning of the symbols used in message content; in order word, to use a common *ontology* (*message semantics*).

All these aspects have been standardized by FIPA [30], which released a specification for an *agent communication language (FIPA-ACL)* [24], which comprises message representation [22], transport protocol [23] and message semantics [29]. Like other agent communication languages (such as KQML [19]), FIPA-ACL is based on the *speech act theory* [37], a social theory that analyzes human communication in order to derive the type of action carried by a message, e.g. whether it is an assertion, a query, a commitment, etc. For this reason, a FIPA-ACL message, which is called *communicative act*, is a structured data whose fields comprise the following main elements:

- The communicative act type (see below).
- The identifiers of the sender and receiver agents.
- The message content, i.e. the true information (message payload) carried by the message.
- The name of the ontology used in message content, so as to make interacting agents understand the meaning of the information.

As for the communicative act type, it is chosen on the basis of the action that the sender intends to perform, e.g. an "*inform*" act is used when the sender wishes to communicate the truth of a given preposition; a "*call-for-proposal*" is used when the sender desires that the receiver makes a proposal on a specified item; a "*request*" specifies that the sender is asking the receiver to do an action (see [33, 24, 19] for more details).

As it can be noted, the communicative act type used depends on sender's desires or intention, and it is thus somewhat connected to sender agent's state. Such a connection is made "more rational" in FIPA-ACL by means of the introduction of *communicative act semantics*: for each communicative act type, two modal logic predicates have been specified, called the *feasibility precondition* (FP) and the *rational effect* (RE). *FP* indicates a precondition that must be met by sender agent's state for the communicative act to be sent, e.g. for an "*inform*" communicative act, the *FP* requires that the sender agent *believes* that the information sent is true and that receiver agent does not have any knowledge on such an information. *RE* instead is a condition to be met, by both sender and receiver agent's state, after sending and delivering the communicative act, e.g. for an "*inform*" communicative act, the *RE* requires that the receiver agent believes that the information sent is true and sender agent believes that receiver believes that the information is true.

The introduction of ACL semantics in agents implies a strong link between reasoning process and interaction, thus making agents "more intelligent" and, above all, more aware of their communicative actions.

6.2 Sending and Receiving Messages in eXAT

Message exchanging in eXAT is basically supported by the **acl** module. As it has been already introduced in the examples shown in the previous Sections, messages are handled by using the predefined Erlang record **#aclmessage**, whose fields correspond to the relevant fields of a FIPA-ACL message defined in [24].

As for message reception, incoming messages are treated as task events and thus are able to trigger agent actions. In this case, the event type is *acl* and the associated pattern specifies how the message has to be formed; such a specification uses a **#aclmessage** record where each field can be either a constant, to indicate a value to be directly matched, or a **fun**, for more complex matching expressions.

Message sending is instead be performed within task actions by using appropriate functions provided by the **acl** module; these functions are named using the same names of the communicative acts of the FIPA-ACL library, e.g. **inform**,

```
-ontology(book).

class(book) ->
  { title = [string, mandatory, nodefault],
    author = [string, mandatory, nodefault],
    genere = [string, mandatory, nodefault] };

class('adventure-book') ->
  is_a(book),
  { genere = [string, mandatory,
              default(adventure)] };

class('thriller-book') ->
  is_a(book),
  { genere = [string, mandatory,
              default(thriller)] };

class(buying_action) ->
  { item = [book, mandatory, nodefault] }.
```

(a)

```
-module(seller_task).
-export([action/2, event/2, pattern/2,
         on_starting/1, sell_item/4]).
-include("acl.hrl").

action(Self, start) ->
  [{new_request_event, sell_item}].

event(Self, new_request_event) ->
  {acl, request_pattern};

pattern(Self, request_pattern) ->
  [#aclmessage {speechact = 'REQUEST',
                ontology = "book"}].

on_starting(Self) ->
  ontology_service:register_codec("book",
                          book_ontology_sl_codec).

sell_item(Self, EventName, Message, ActionName) ->
  % Extract parsed content
  [RequestData] = Message#aclmessage.content,
  process_request(RequestData),
  object:do(Self, start).

process_request(Msg,
                RequestData = #buying_action {}) ->
  % Process the request ...
  % Prepare the reply
  ReplyContent = #done { action = RequestData },
  % Send the reply
  acl:reply(Msg, 'INFORM', ReplyContent);

process_request(Msg, RequestData) ->
  % Reply with a 'not understood'
  acl:reply(Msg, 'NOT-UNDERSTOOD', RequestData);
```

(b)

Figure 8: Ontology and Communication in eXAT

`call_for_proposal`, `agree`, `request`, etc., and take, as the sole parameter, an `#aclmessage` record.

As introduced in Section 6.1, in order to favor the interaction among heterogeneous agents, message exchanged must be encoded using an appropriate syntax; to this aim, FIPA specifies an ASCII representation [22]. Therefore, in order to allow eXAT agents to handle Erlang types, while maintaining interoperability, the provided `acl` module includes the functions to automatically perform the proper FIPA-ASCII/Erlang encoding/decoding process.

6.3 Handling Ontologies

Interoperability among different agents is ensured not only by using the same syntax for messages but also by making interacting agents to share the same concepts of their "universe of discourse": in other words, they should share the same *ontology*. For this reason, the structure of an ACL message includes a field for the specification of the name of the ontology used in the message content.

To this purpose, eXAT allows a programmer to write and manipulate ontologies by using concepts organized in *classes with hierarchies*. An agent programmer can write an ontology in a suitable specification file (using an Erlang-like notation); then an *ontology compiler*, provided with eXAT, is able to parse such a specification and generate the relevant Erlang type definitions to be used in agent source code. For each class of the ontology an Erlang record is defined; since Erlang is not object-oriented, Erlang records are generated by "flattening" the hierarchy, i.e. by embedding all the attributes of a class/record into the ancestor(s) class/record; in addition, to maintain the object structure, an Erlang source file is also generated, which includes appropriate functions to manage the class hierarchy.

The other task of the ontology compiler is the generation of the *Codec*, i.e. an Erlang code implementing the routines for the automatic FIPA/Erlang translation. This means that, agent programmers can use and refer Erlang records in message content, because, according to specified ontology, the relevant Codec is charged with the task of performing automatically translation from/to FIPA representation, making interoperability with other agents and platforms possible.

Figure 8 reports an example of a "book" ontology specification (Figure 8a) and its use in a "seller" agent (Figure 8b); as the listing shows, the first task of the seller agent (function `on_starting`) is the registration of the codec (generated by the ontology compiler) for the "book" ontology; then the agent waits for a "*request*" message, processing it: if the message carries a "*buying_action*" request, it is processed and then an "*inform*" communicative act is replied, signaling that the action has been done; otherwise a "*not-understood*" communicative act is replied.

The derived Erlang records and functions can be also used in ERESYE engines [11] in order to allow a programmer to manipulate the same concepts in managing both reasoning and inter-

acting aspects. This feature is exploited to support FIPA-ACL semantics, as detailed in [16], thus realizing a direct connection between message sending/receiving and agent intelligence; this makes the implementation of "true rational" agents possible. In this sense, eXAT is the first platform that concretely supports FIPA-ACL semantics.

7 Conclusions

This paper described eXAT, a FIPA-compliant platform realized by the authors for the implementation of software agents in Erlang. The platform has been designed in order to exploit Erlang native constructs for the purpose of facilitating agent implementation, and by taking care of not only behavioral aspects, but also reasoning and communication capabilities. To this aim, eXAT models agent behavior by means of finite-state machines enriched with composition and specialization abstractions, while agent intelligence is made possible through the provided rule-based inference engine. Finally, agent interaction is supported by suitable modules that handles ACL messages according to the FIPA standard; to facilitate such a process, an ontology compiler is provided, to allow a programmer to write her/his own ontology and use it in agents. Since the same ontologies can be used also in inference engines, a tight connection between behavior, interaction and reasoning is made possible; such a characteristic, however, is not featured by other widely known agent platform (mainly based on Java), thus making eXAT an interesting and effective alternative for the realization of multi-agent systems.

References

[1] `http://fipa-os.sourceforge.net/`. FIPA-OS Web Site., 2003.

[2] `http://herzberg.ca.sandia.gov/jess/`. JESS Web Site, 2003.

[3] `http://www.ghg.net/clips/CLIPS.html`. CLIPS Web Site, 2003.

[4] `http://www.agentlink.org/resources/agent-software.php`, 2004.

[5] `http://www.diit.unict.it/users/csanto/exat/`. eXAT Web Site, 2004.

[6] `http://www.drools.org`. Drools Home Page, 2004.

[7] `http://www.agent-software.com`, 2004.

[8] `http://sourceforge.net/projects/zeusagent/`. ZEUS Agent Toolkit Web Site., 2005.

[9] J. L. Armstrong, M. C. Williams, C. Wikstrom, and S. C. Virding. *Concurrent Programming in Erlang, 2nd Edition.* Prentice-Hall, 1995.

[10] F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with a FIPA-compliant agent framework. *Software: Practice and Experience*, 31(2):103–128, 2001.

[11] A. Di Stefano, F. Gangemi, and C. Santoro. ERESYE: Artificial Intelligence in Erlang Programs. In *Erlang Workshop at 2005 Intl. ACM Conference on Functional Programming (ICFP 2005)*, Tallinn, Estonia, 25 Sept. 2005.

[12] A. Di Stefano and C. Santoro. eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang. In *AI*IA/TABOO Joint Workshop on Objects and Agents (WOA 2003)*, Villasimius, CA, Italy, 10–11 Sept. 2003.

[13] A. Di Stefano and C. Santoro. eXAT: A Platform to Develop Erlang Agents. In *Agent Exhibition Workshop at Net.ObjectDays 2004*, Erfurt, Germany, 27–30 Sept. 2004.

[14] A. Di Stefano and C. Santoro. Designing Collaborative Agents with eXAT. In *ACEC 2004 Workshop at WETICE 2004*, Modena, Italy, 14–16 June 2004.

[15] A. Di Stefano and C. Santoro. On the use of Erlang as a Promising Language to Develop Agent Systems. In *AI*IA/TABOO Joint Workshop on Objects and Agents (WOA 2004)*, Torino, Italy, 29–30 Nov. 2004.

[16] A. Di Stefano and C. Santoro. Building Semantic Agents in eXAT. In *AI*IA/TABOO Joint Workshop on Objects and Agents (WOA 2005)*, Camerino, Italy, 14–16 Nov. 2005.

[17] A. Di Stefano and C. Santoro. Supporting Agent Development in Erlang through the eXAT Platform. In *Software Agent-Based Applications, Platforms and Development Kits.* Whitestein Technologies, 2005.

[18] A. Di Stefano and C. Santoro. Using the Erlang Language for Multi-Agent Systems Implementation. In *2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'05)*, Compiégne, France, 19–22 Sept. 2005.

[19] T. Finin and Y. Labour. A Proposal for a New KQML Specification. Technical Report TR-CS-97-03, Computer Science and Electrical Engineering Dept., Univ. of Maryland., 1997.

[20] C. Forgy. OPS5 Users Manual. Technical Report CMU-CS-81-135, Dept. of Computer Science, Carnegie-Mellon Univ., 1981.

[21] C. Forgy. The OPS Languages: An Historical Overview. *PC AI*, Sept. 1995.

[22] Foundation for Intelligent Physical Agents. FIPA ACL Message Representation in String Specification—No. SC00070I, 2002.

[23] Foundation for Intelligent Physical Agents. FIPA Agent Message Transport Protocol for HTTP Specification—No. SC00084, 2002.

[24] Foundation for Intelligent Physical Agents. FIPA Communicative Act Library Specification—No. SC00037J, 2002.

[25] Foundation for Intelligent Physical Agents. FIPA Contract Net Interaction Protocol Specification—No. SC00029H, 2002.

[26] Foundation for Intelligent Physical Agents. FIPA Dutch Auction Interaction Protocol Specification—No. XC00032F, 2002.

[27] Foundation for Intelligent Physical Agents. FIPA English Auction Interaction Protocol Specification—No. SC00031F, 2002.

[28] Foundation for Intelligent Physical Agents. FIPA Request Interaction Protocol Specification—No. SC00026H, 2002.

[29] Foundation for Intelligent Physical Agents. FIPA SL Content Language Specification—No. SC00008I, 2002.

[30] Foundation for Intelligent Physical Agents. `http://www.fipa.org`, 2002.

[31] S. Franklin and A. Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *Third International Workshop on Agent Theories, Architectures, and Languages (ATAL)*. Springer-Verlag, 1996.

[32] K. Hindriks, F. de Boer, W. van der Hoek, and J.-J. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[33] Y. Labrou, T. Finin, and Y. Peng. Agent Communication Languages: the Current Landscape. *IEEE Intelligent Systems*, March-April 1999.

[34] F. McCabe and K. Clark. April: Agent Process Interaction Language. In N. Jennings and M. Wooldridge, editor, *Intelligent Agents*. Springer, LNCS 890, 1995.

[35] F. McCabe and K. Clark. Go! - A Multi-Paradigm Programming Language for Implementing Multi-Threaded Agents. *Annals of Mathematics and Artificial Intelligence*, 41(2-4):171–206, August 2004.

[36] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: Implementing a BDI-Infrastructure for JADE Agents. *Telecom Italia Journal: EXP - In Search of Innovation (Special Issue on JADE)*, 3(3), Sept. 2003.

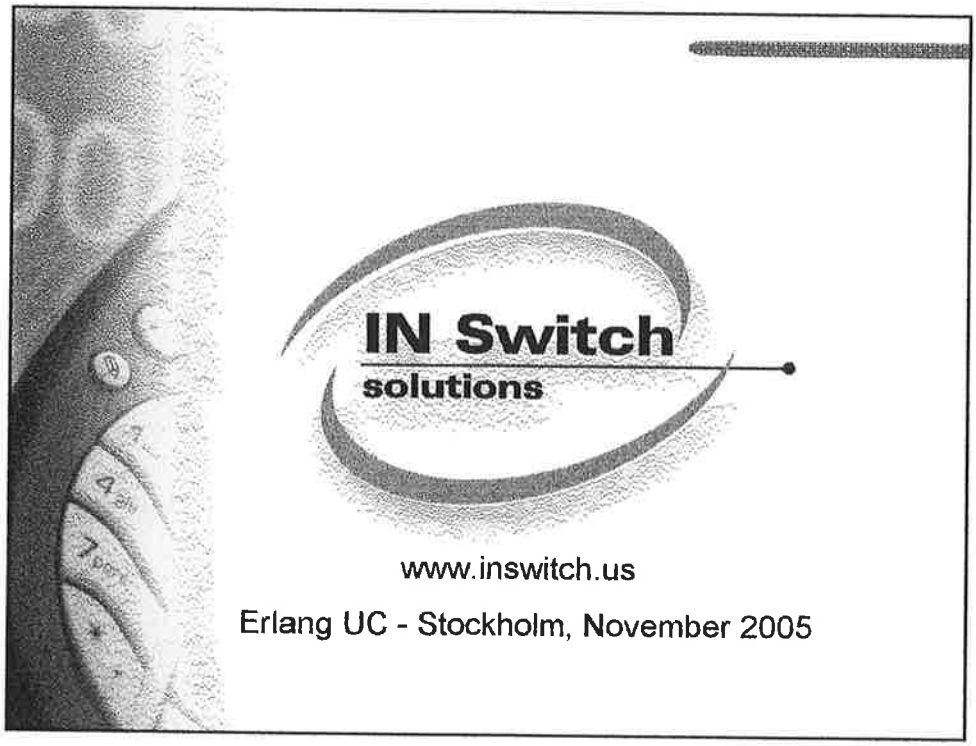
[37] J. R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, 1969.

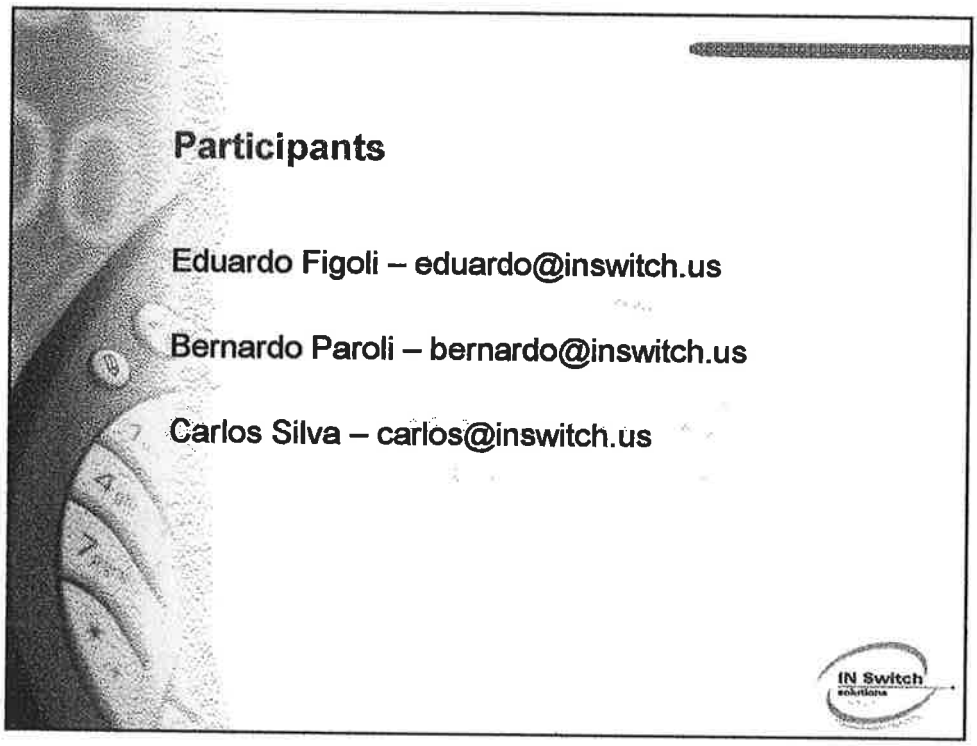
[38] K. Sycara, M. Paolucci, M. van Velsen, and J. Giampapa. The RETSINA MAS Infrastructure. *Special joint issue of Autonomous Agents and Multi-Agent Systems Journal*, 7(1 and 2), July 2003.

[39] C. van Reeuwijk and H. J. Sips. Adding tuples to Java: a study in lightweight data structures. *Concurrency and Computation: Practice and Experience*, 17(5–6):423–438, 2005.

[40] M. J. Wooldridge. *Multiagent Systems*. G. Weiss, editor. The MIT Press, April 1999.

[41] M. J. Wooldridge. *Reasoning About Rational Agents*. The MIT Press, July 2000.


IN Switch
solutions
www.inswitch.us
Erlang UC - Stockholm, November 2005


Participants
Eduardo Figoli – eduardo@inswitch.us
Bernardo Paroli – bernardo@inswitch.us
Carlos Silva – carlos@inswitch.us
IN Switch
solutions

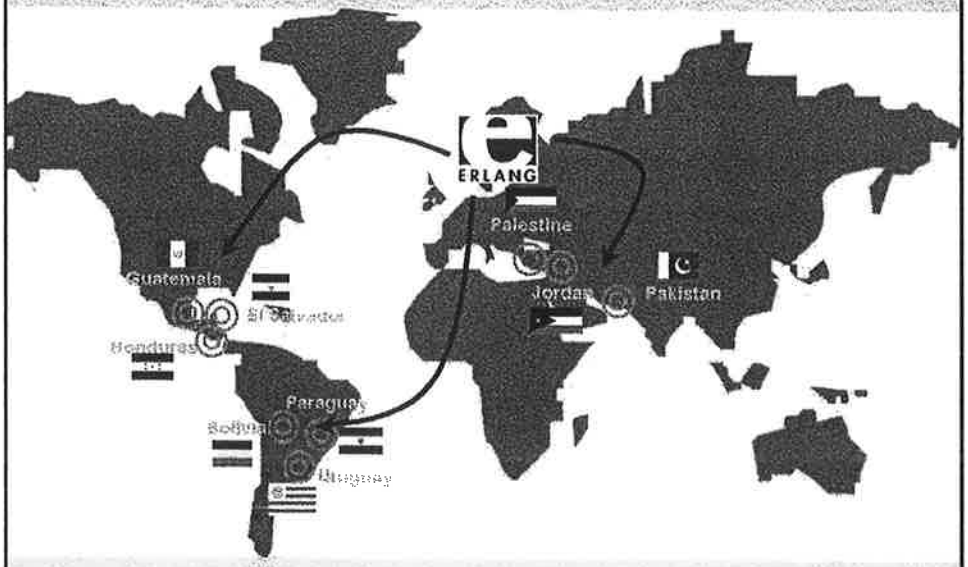
1

INSwitch Solutions

- Established as a corporation in the State of Florida, USA.
- Development Center is located in **Montevideo, Uruguay**.
- Additional business and support offices in Asia and Latin America.
- Support centres at Guatemala, Paraguay and Uruguay.
- Provides turn key solutions as well as technology building blocks (system modules) to system integrators, fixed and mobile operators.

IN Switch solutions

IN Switch Solutions Erlang installations



2

Mobile eTopUp

- Customers purchase prepaid air time at retailer's mobile agents using a standard mobile handsets as the distribution device
- Fixed or variable amounts
- Eliminate PIN hidden numbers generation errors and terminals costs
- Guarantee stock supply in the distribution network increasing availability

IN Switch solutions

eTopUp with USSD

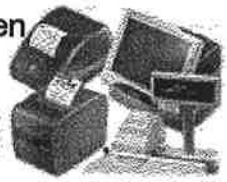
- Retailer places a recharge transaction using his mobile phone.
- USSD technology offers WAP style text.
- The credit is automatically loaded into subscriber prepaid account.
- Since no POS are needed, non traditional retailers can take part of the system expanding the solution while reaching more customers.
- Online transactions are faster and more secure since Operator's Signaling channels are used.

IN Switch solutions

3

eTopUp with POS Terminals

- POS Terminals at retail stores are used for printing PIN numbers. Vouchers are loaded in bulk in the terminal and printed out when required.



- VISA II protocol

Advantages
- ✓ Low operational costs
- ✓ Reduces Fraud
- ✓ Allows multiple voucher denominations

Disadvantages
- ✓High set up costs
- ✓Transaction (phone line) costs
- ✓Requires training
- ✓Low penetration
- ✓Requires PIN management
- ✓Use of IVR resources

IN Switch solutions

TIGO Case Study

Millicom mobile operators in Guatemala, El Salvador, Honduras, Paraguay and Bolivia, under the branded name TIGO launched ePIN (Inswitch's eTopUP Platform) in August 2005.

2 Centralized platforms:
- One in Guatemala attending Guatemala, El Salvador and Honduras
- One in Paraguay attending Paraguay and Bolivia

IN Switch solutions

4

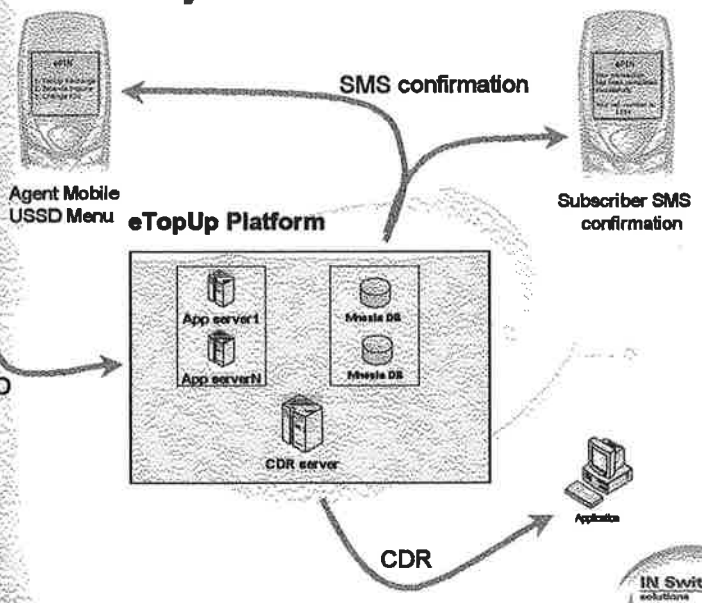
TIGO Case Study

Integration with:

- Ericsson and Motorola MSC/HRL
- Logica and Comverse SMSC
- Comverse Prepaid integration through SOAP and propietary protocols
- Billing and Banking through provided protocols

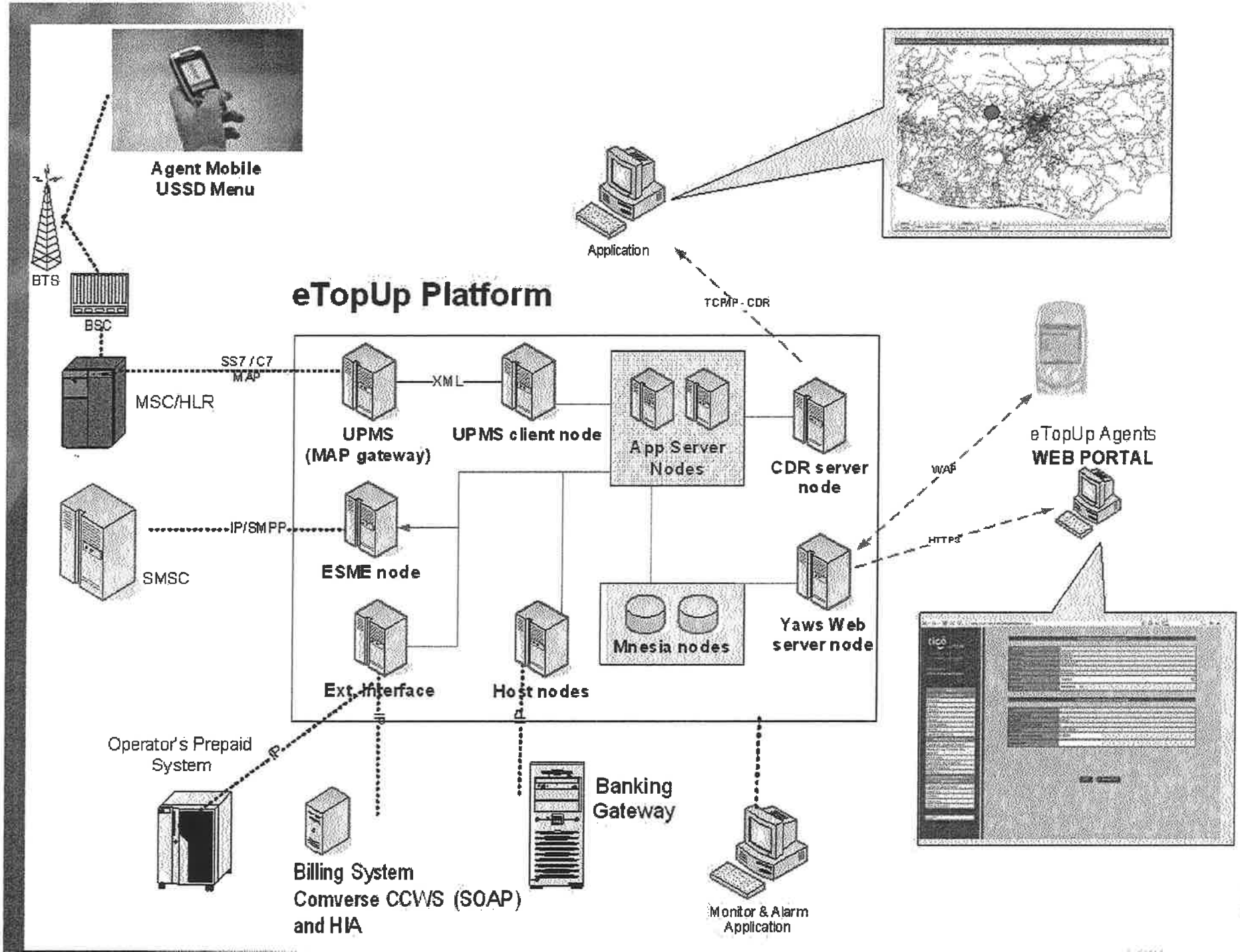
IN Switch solutions

TIGO Case Study



IN Switch solutions

5


Agent Mobile
USSD Menu
BTS
BSC
MSC/HLR
SS7 / C7
MAP
SMSC
IP/SMPP
eTopUp Platform
UPMS
(MAP gateway)
XML
UPMS client node
App Server
Nodes
CDR server
node
ESME node
Mnesia nodes
Yaws Web
server node
Ext. Interface
Host nodes
IP
IP
IP
Application
TCMP - CDR
WAP
HTTPS
eTopUp Agents
WEB PORTAL
Operator's Prepaid
System
Billing System
Comverse CCWS (SOAP)
and HIA
Banking
Gateway
Monitor & Alarm
Application

Yaws Web Server Pages – agent information

tigo ePIN

IN Switch Solutions
www.inswitch.us

Bienvenido
ePIN Administrator

Menú

Configuración
- Agentes
- Zonas
- Celdas
- Móviles
- TopUp
- Promociones
- Marcas
- Cambio de COS

Manejar Agentes
- Cambiar Cuenta Padre
- Manejar Cuenta del Agente
- Reversión
- Comisión de Agentes
- Transferencias
- Ordenes

Usuarios, perfiles y permisos
- Cambiar Contraseña
- Usuarios
- Perfiles
- Permisos
- Configuración

Logoff

Información del Agente

Id	29
Nombre	Distribuidor Demo
Anexo	010
Código de Cliente	502
Cuenta	
Nivel	Distribuidor
Zonas	Zona1
Marcas	Tigo

Datos de Distribuidor

Banco	Industrial Demo
Número de Cuenta	010010001
Estado	Activa
Descripción	Monetarios

Móviles del Agente

No. Celular	Estado				
55252998	Activo				

Agregar Móvil

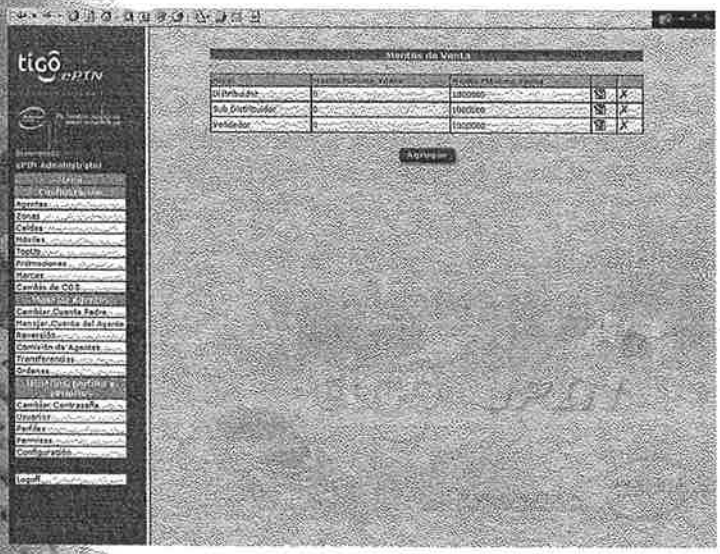
Hijos del Agente

Id	Nombre	Nivel			
42	Sub Distribuidor Demo	SubDistribuidor			

Agregar Hijo Cancelar

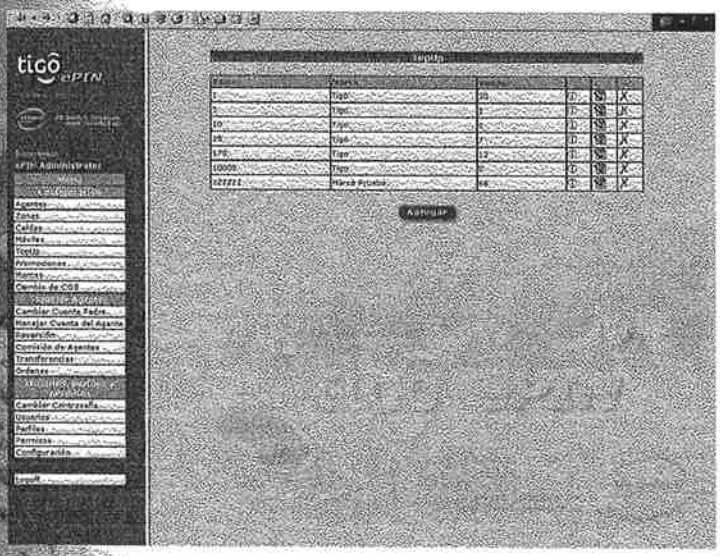
IN Switch solutions

Yaws Web Server Pages – money transfer



IN Switch solutions

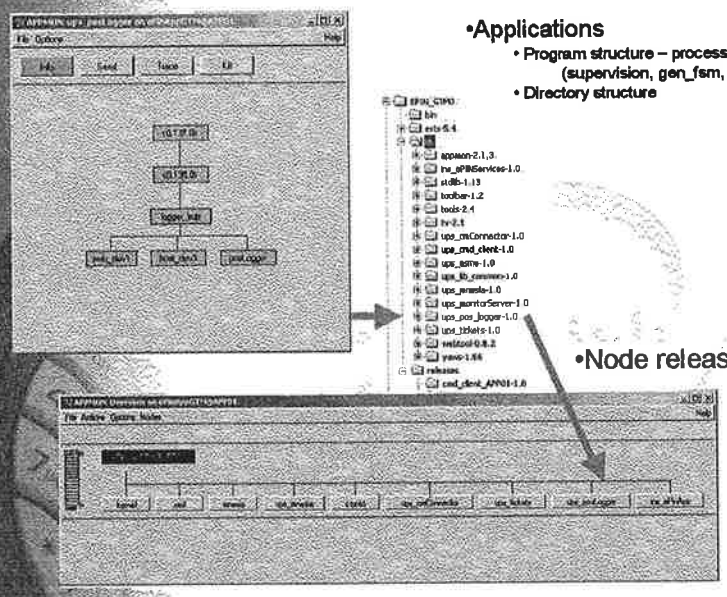
Yaws Web Server Pages – top up



IN Switch solutions

8

Erlang/OTP and eTopUp



- Applications
  - Program structure – processes behaviours (supervision, gen_fsm, ...)
  - Directory structure
- Node release

HW and SW technology used

Hardware:

- Servers HP DL 360 Pentium IV 3.2 GHz with 2 GB RAM
- Telesoft NS700 signaling card

Software:

- Windows 2000 Server
- Erlang 5.4 (release R10B) licensed

9

Why Erlang?

1. Light weight processes
2. OS independent (Windows / Linux) - portability
3. Fault tolerance mechanisms (supervision, 99.999,...)
4. Hot code loading
5. Distributed environment and DB
6. Support from Ericsson and live Erlang community

Advantages:

1. Mnesia DB stores Erlang data types
2. Easy management of a huge number of processes
3. Low cost implementation
4. Erlang OTP framework
5. Robust and scalable distributed software
6. Easy programming language and fewer lines of code

Not so good:

1. Non standard SQL interface to Mnesia
2. Needs more memory and processor power
3. Erlang SSL port *(ssl_sock)* with Yaws not working properly under Windows
4. No IDE and easy graphical debugger

Some of our products

- Land and Wireless Prepaid Platforms
- Calling Cards Platforms
- Prepaid Roaming
- Location Based Engines
- USSD Gateways
- SMSC
- Graphical Service Creation (IVR, SMS, USSD)
- Convergent VAS
- GPRS Charging
- Ring Back Tones
- Welcome Roamers
- Missed Call Alert
- Virtual HLR

11

Inswitch Erlang Team

1. Inswitch has a team of about 10 members speciallized in Erlang since 2002
2. We are thinking of increasing the team number

IN Switch solutions

Thank you !!

Please visit us at www.inswitch.us

IN Switch solutions

Teaching Functional Programming and Erlang: The Galician Experience*

Victor M. Gulias

MADS Group, Department of Computer Science
University of A Coruña, Spain
gulias@dc.fi.udc.es

Abstract. In this paper, we present the experience of teaching functional programming in the Computer Engineering programme of a Galician University, the University of A Coruña. Erlang is introduced as part of an optional course on functional programming as an example of application of functional paradigm to the real world and most of the students really appreciate the beauty of the language when describing concurrent systems. In fact, several students choose Erlang every year as development framework for their Master's Thesis or for conducting research in our doctorate programme.

1 Introduction

In this paper, we present our experience of teaching functional programming in the Computer Engineering programme of a Galician University, the University of A Coruña. The functional paradigm plays an important role in the whole programme and Erlang [AWWV96] has become a relevant part of an optional fourth-year course on functional programming.

Besides academic topics (such as λ-calculus, type systems and so on) and academic languages (OBJECTIVE CAML, HASKELL), Erlang is introduced as part of the course as an example of application of functional programming in the real world and most of the students really appreciate the beauty of the language when describing concurrent systems. In fact, several students choose Erlang every year as implementation language for their Master Thesis or for conducting research in our doctorate programme in front of more popular languages such as JAVA or C/C++.

The paper is structured as follows. In the next section, we present a general overview of the teaching environment, introducing University of A Coruña and its Faculty of Informatics as well as the role played by declarative programming in the studies. Section 3 presents the general outline of the functional programming course. Section 4 presents some remarks based on of personal experience teaching Erlang. Finally, we conclude.

* Partially supported by ERDF and Spanish MEC TIN2005-08986.

2 The Environment

2.1 University of A Coruña and its Faculty of Informatics

The University of A Coruña (UDC, `http://www.udc.es`) was established in 1989 as scission of an older one, the University of Santiago de Compostela. UDC holds more than 25.000 students spread over seven campuses in two different Galician cities (A Coruña and Ferrol). Faculty of Informatics (`http://www.fic.udc.es`) is located in A Coruña, a city just by the Atlantic coast in the north west of Spain with about 300.000 people in the metropolitan area. It is one of the most important and largest schools at UDC with about 2.500 students (both undergraduate and graduate). Curiously, Faculty of Informatics is older than UDC, being established in 1986 as part of University of Santiago de Compostela.

Nowadays, the Faculty of Informatics holds three different study programmes in computer science –Engineering (five years) and two Technical Engineerings (three years)– as well as several doctorate programmes for graduate students.

2.2 The Role of Declarative Programming in the Studies

Declarative paradigm plays a relevant role in the study programmes offered by Faculty of Informatics. Students of fourth (or fifth) year can choose a specific optional course on Functional Programming (described in detail in section 3). However, students get in touch with functional paradigm previously. There is a mandatory course on declarative programming in the second year (three hours in classroom and two in the laboratory per week during one semester). Here, the students learn two different programming models: logical (PROLOG) and functional (OBJECTIVE CAML, the french dialect of ML family).

Besides this mandatory course, students use OBJECTIVE CAML as the implementation language for several other courses such as *Programming Technology*, *Automata and Formal Languages*, *Artificial Intelligence*, *Compilers*, etc. In some courses, it is a student's choice the implementation language for practical exercises and people usually agree that it is easier to use a functional language than using other more popular languages such as JAVA, C/C++ or any of the other nine different languages that students must learn in the first three years. In fact, sometimes you have to smile when other lecturers explicitly forbid the use of functional languages for students' duties because *"that way the proposed exercises are just too easy"*.

2.3 Why Objective Caml?

At the end of the eighties, a research group (LFCIA, `http://www.lfcia.org`), with strong mathematical background and leaded by former dean Jose L. Freire, was exploring applications of category theory. At that time, there was an implementation of an ML dialect based upon this formalism, the *categorical abstract machine* [CCM85] used by the language CAML [Mau91]. Hence, this language was gradually introduced in lectures replacing other options at that time such

as Edinburgh's Standard ML or Miranda. Though the language evolved (Leroy's CAML LIGHT and finally OBJECTIVE CAML), dropping its categorical foundation, it was the language of choice for teaching functional programming.

OBJECTIVE CAML [Ler00] (O'CAML for short) has several nice features for teaching: it is efficient (with a native code compiler), portable (both Windows and particularly Linux are usual target platforms), with most of the state-of-the-art functional features (static typing, anonymous functions, parametric polymorphism, powerful module system, partial application, and so on), and a reasonable set of libraries for the development of interesting applications (from the teaching point of view).

3 The Functional Programming Course at UDC

Functional programming is an optional course that is usually taken by students in the fourth year (fall semester). The student load is three hours in the classroom and two in the laboratory per week during 14 weeks (equivalent to 5.0 ECTS). It is a popular optional course, with about 40 people per year (about 25% of fourth year students), even though it has a strange (crazy?) schedule according to European customs (for example, a laboratory session from 8pm to 10pm at friday night). The course is divided into five different parts:

1. **Quick review of functional programming concepts.** Using O'CAML, the first two weeks are used to refresh previous knowledge on functional programming. Topics of interest: (a) Values, types and expressions; (b) Identifiers, definitions and scope; (c) Predicates, conditional expressions and pattern-matching; (d) Recursion; (e) Lists; (f) High-order functions, partial application; (g) Polymorphism; (h) User-defined and abstract datatypes; (i) Modules and Functors.
2. **Introduction to λ-Calculus.** In order to better understand the fundamentals of functional programming, we present the λ-calculus in three-four weeks. Using O'CAML the student implements a small interpreter. After studying different evaluation models, a short presentation of HASKELL let us introduce a language with lazy evaluation. Topics: (a) Pure λ-calculus; (b) Substitution and reduction rules; (c) Normalization; (d) Lazy vs. eager evaluation; (e) Fix-point combinators and recursion; (f) Extending the λ-calculus.
3. **Type Systems.** We extend the pure λ-calculus introducing values and an static type system which helps to understand how type inference works. During three-four weeks, the students use O'CAML to add static typing to their own interpreters.
4. **Implementation Details of Functional Compilers.** Depending on students' interest and available lecturers, we spend one-two weeks with (some of) these topics:
   (a) Garbage collection algorithms
   (b) Internal representation and communication with low-level languages

(c) Pattern-matching implementation

5. **Functional programming in the real-world**. That means **Erlang**. At the beginning, it was a short seminar of two hours but students interest suggested to offer a two-three weeks course on concurrent functional programming in Erlang. This module of the course covers most of the topics proposed by basic and continuation Erlang courses, though with a quite different approach due to the student background.

Since 1999, Victor M. Gulias has been in charge of the course even though right now he only conducts the coordination of the whole course and the teaching of the Erlang part.

4 Experience Teaching Erlang

Some general impressions of the students when teaching Erlang:

- After learning a modern functional languages such as O'CAML, Erlang looks somehow primitive: no static typing, no modules, no partial application, etc.
- After a first contact with the compiler in the laboratory, they experiment the "type freedom" effect and start loving the language. Nevertheless, the use of a typed language before seems to be present in their minds: most of them use comments to "informally" state function signatures. In fact, as they get into more complex problems, they miss the type facilities and point out this as the more important problem of Erlang.
- Panic! They are frighten when first hear the term "message passing". They have experimented, in previous courses, the development of parallel applications using infamous low-level libraries such as MPI or PVM. They are amazed that Erlang's runtime system gets in charge of all the marshalling of data. World starts becoming a better place to program distributed applications.
- The main goal of the functional programming course is *abstraction*. They appreciate examples such as the generalization of a server, separating the imperative part (recursion loop and communication) from the definition of the particular services, or the definition of skeletoms such as an abstract divide-and-conquer algorithm.
- The notion of *behaviour* is very well accepted as they are learning (from the same lecturer and at the same time!) classical GoF design patterns [GHJV95] with JAVA implementation in a mandatory fourth-year *System Design* course. At this moment, students ask why Erlang is not being used in the system design course with (or instead of!) JAVA.
- Though in the course we try to use familiar design artifacts (such as UML's state or sequence diagrams), students feel that design tools and techniques do not have Erlang in mind.
- Erlang students, when have to implement simple concurrent programs in JAVA in the system design course, just miss the good days programming in Erlang... but at the end they get better and cleaner solutions than regular students. Even some of them use Erlang to quick implement a solution to better understand the problem.

5 Conclusions

Erlang is an excellent example of a real-world success of functional paradigm and, in the case of University of Coruña, is also a success case among students. The Erlang part of the functional programming course is really appreciated by students and they demand even more time dedicated to Erlang-related topics. "Ericsson", "complex real-time applications", "distributed programming", in summary "Real-world stuff", are by far the best advertisement for Erlang. Students that really get into the language feel that they are going backwards when returning to tradicional but more popular approaches such as JAVA. Some of the students continue working with Erlang by means of their Master's Thesis, where they can conduct a larger Erlang-based project, or joining the doctorate programme where they can access to additional lectures involving Erlang.

In order to measure the impact of the language in the students, we should count several Master's Thesis in the last few years (about 2-3 per year), at least one large research project related with Erlang (VoDKA project [GBF05]) and several on-the-way Ph.D. Thesis. In addition, at least three SMEs has been recently created using Erlang/OTP as platform for the development of their products, which is a notable success indicator if we take into account the poor industry development of Galicia region.

References

[AWWV96] J. L. Armstrong, M. C. Williams, C. Wikström, and S. R. Virding. *Concurrent Programming in Erlang.* Prentice Hall, 2nd edition edition, 1996.

[CCM85] G. Cousineau, P. Curien, and M. Mauny. The categorical abstract machine. In J-P. Jouannaud, editor, *Proceedings Functional Programming languages and Computer Architecture*, volume 201 of *LNCS*, pages 50–64. Springer-Verlag, 1985.

[GBF05] V. Gulias, M. Barreiro, and J. Freire. Vodka: Developing a video-on-demand server using distributed functional programming. *Journal of Functional Programming*, 15(3):403–430, may 2005.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software.* Addison Wesley, 1995.

[Ler00] X. Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.

[Mau91] M. Mauny. Functional programming using CAML. Technical Report RT-0129, Inria, Institut National de Recherche en Informatique et en Automatique, 1991.

Concurrent Erlang Flow Graphs

Manfred Widera
Fachbereich Informatik
FernUniversität in Hagen
58084 Hagen
Germany
manfred.widera@fernuni-hagen.de

ABSTRACT

Flow graphs are an important, and useful tool for testing programs or program components during software development. For imperative languages it is state of the art to use flow graph based coverage tools during the unit testing stage. Based on flow graphs for functional programming languages, that have to cope with higher order functions, a flow graph concept for Erlang needs a special treatment for the concurrent language constructs that are typical of Erlang. This paper presents a definition of flow graphs for Erlang programs that especially handles process generation and message passing, and describes how these flow graphs can be computed.

1. INTRODUCTION

Testing of software is a widely used method of detecting errors during the software development process. Every software is tested before being used in practice. Though testing can just prove the presence, but not the absence of errors, the passing of all tests given by an appropriate test set is often understood as an evidence for reaching a certain level of software quality. For imperative programming there are several approaches defining the appropriateness of a test set by coverage criteria based on the flow graph. Compared to the test case selection based on the specification of a program, these structure oriented criteria have the closest correspondence to the *actual implementation* under testing. Structure oriented testing is usually applied to small program fractions like single modules, and is an important part of the early stages of software development.

In the context of Erlang, the only available implementation of source code directed testing is an ad hoc approach that checks the individual *lines* of a program for coverage [6]. This tool works by an instrumentation of the the source code of a module that is focused on the executable lines; it is therefore not extendible to take into account relationships between distant parts of the program, e.g. the data flow, and especially the concurrent data flow in an Erlang program.

As systematic testing is an important task of professional software development, it is desirable to have more advanced source code oriented testing methods for Erlang available. It is important to note, that systematic testing cannot be replaced completely by employing the suitability of functional languages for verification. As the first main reason, verification is a quite expensive, and time consuming task, and cannot be applied to all the less critical components (which should nevertheless be as correct as possible). Second, verification is always done against an already formalized specification of the intended program behavior which itself is not guaranteed to be correct.

The aim of this paper is to give a definition of flow graphs of Erlang programs similar to the known flow graph definition for imperative programs, and to describe a system generating such a flow graph from a program's source code. Based on an preliminary approach for sequential Erlang programs [23], the approach described here covers the whole Erlang standard, especially handling process generation and message passing.

The rest of the paper is organized as follows. In Sec. 2 related work is described and the current paper is classified in this context. Section 3 presents the language under consideration which is essentially given by the full Erlang standard without some of the syntactic sugar. A definition of concurrent flow graphs is given in Sec. 4. Section 5 recalls and refines some definitions of data flow analysis which are necessary for the computation of concurrent flow graphs presented in Sec. 6. Conclusions, and possible areas of future work are presented in Sec. 7.

2. RELATED WORK

2.1 Flow Graphs and Sequential Testing

The work presented here is related to publications from several areas. In imperative programming languages, flow graphs are accepted as a standard tool for checking test case coverage during the unit testing stage [24]. In the context of functional programming there are already approaches on flow graphs that are, however, not focused on test case coverage. Van den Berg [19] uses flow graphs, and call graphs for software measurement on functional programs. The flow graphs used there consider function calls as atomic operations and are generated for each function independently.

Information on calls between functions is given by a call graph as separate structure.

A concept of generating flow graphs for higher order functional programs is described by Shivers [17] and further analyzed by Ashley/Dybvig [1]. Especially, the level 0CFA described there is very similar to our approach. Due to its use of continuation passing style (CPS) and the Y combinator, it is, however, not very adequate for presenting the analysis results to human programmers. The same holds for works based on Shivers approach [17]. They do not focus on the presentation of the generated flow graphs to the programmer.

Different approaches on testing and debugging functional programs have been proposed. QuickCheck [4] aims at automatically checking Haskell programs by generating input data on a random basis and checking the results with constraints on the expected output. In the WYSIWYT framework [14, 15, 16] flow graphs are used for judging the coverage of a functional program by a set of test inputs. This approach is, however, restricted to spreadsheets considered as first order functional programs without recursion.

Several approaches on declarative debugging and tracing functional languages (e.g. [8], [12], [3, 20]) describe how to trace down the programming errors causing an observed misbehavior of a program. These approaches, however, do not provide mechanisms for generating or judging the test sets that are used to provoke such a misbehavior.

The module cover that comes with the tools library of Erlang [6] implements a coverage test for Erlang modules, that analyzes the individual lines of the source code for coverage. It is, however, not able to distinguish between several computations coded within a single line, or to check non-local relationships, e.g. between calls, and called functions or between throws, and corresponding catches. Since the distant relationships between send operations, and receives in a program are also not considered by cover, the concurrent structure of Erlang programs is not an additional challenge for the system. In contrast, the preliminary approaches to flow graph based testing in sequential Erlang [23] need a nontrivial extension to handle concurrent Erlang constructs.

2.2 Testing Concurrent Programs

In testing concurrent programs, one is usually especially interested in certain interactions between the different processes or threads that lead to a number of specific errors like deadlocks and race conditions. The approaches to ensure the correctness of concurrent programs are divided into static and dynamic approaches.

Static analysis of concurrent programs is often done in the form of model checking. The underlying concept as well as the VeriSoft tool for performing model checking are described by Godefroid [9, 10].

The dynamic testing of concurrent systems is based on executing different schedules of synchronization events (i.e. events that are observable from outside the triggering process). Besides non-deterministic testing of schedules generated by chance, there are different systematic approaches for generating schedules and for enforcing their execution. Carver, and Tai [2] describe a deterministic testing approach by enriching the program code with special calls to a scheduler that is able to generate and repeat different schedules of interest. A similar approach of a scheduler function that is explicitly called is followed by Stoller [18] for Java Programs. Factor, Farchi, and Talmor [7] also address Java programs. Besides the schedule replay they focus on a coverage test for schedules. A further approach on systematically generating schedules is given by Hwang, Tai, and Huang [11]. For a given valid schedule new prefixes of schedules are generated by introducing minimal changes to the known schedules.

2.3 Classification of the Current Paper

The approach presented here stands in the tradition of testing sequential programs [24]: a flow graph oriented testing tool is applied to program parts that are usually too small for detecting the special forms of errors described in Subsec. 2.2. Concurrency and message passing are, however, very prominent parts of the Erlang design, such that a strategy in handling the concurrent language features to some extent by the structure oriented testing process is necessary. The aim of this work is to take into account the effects of message passing on the possible destinations of higher order function calls and on data flow oriented coverage criteria.

During the software development cycle the concurrent flow graphs described here, and the coverage criteria based on them can replace previously available structure oriented coverage approaches. Other stages of the testing process remain unchanged. This is especially the case for detecting synchronization errors, where tools as those described in Subsec. 2.2 can be employed.

3. PRELIMINARIES

The flow graph generation is defined (and is mostly implemented) for the whole Erlang standard [5]. The presentation in this paper is, however, restricted to the subset defined in Fig. 1 (ignoring the boxes around some expressions for the moment). Definitions consisting of a $\star$ are not of interest here, and are therefore omitted. Infix operators are considered as ordinary functions. Timeouts for receive expressions are omitted for simplification reasons. The BIFs *throw/1* and the binary operator *!* (which is denoted as ordinary function *send/2*) need a special treatment and are therefore considered as syntactic keywords in this work.

In the following when speaking of a first order function call we mean a call of the form $fn(e_1, e_2, \ldots, e_k)$ with a function name fn, and a higher order function call has the form $e_0(e_1, e_2, \ldots, e_k)$ with an expression e_0.

In the rest of this paper programs are assumed to fulfill the following *named definition property* which is easy to obtain by a preprocessing stage.

DEFINITION 1 (NAMED DEFINITION PROPERTY). *A program P fulfills the* named definition property *if*

- *Every expression in P whose position is that of a boxed expression in Fig. 1 consists of an instantiated variable.*

constants a: $\star$

variables X: $\star$

patterns p: $a|X|\{p_1,\ldots,p_k\}|[p_1|p_2]|[p_1,\ldots,p_k]$

guards g: $\star$

if clauses ic: $g \rightarrow l$

case clauses cc: p [when g] $\rightarrow l$

fun clauses fc: $(p_1,\ldots,p_k)$ [when g] $\rightarrow l$

function name fn: $\star$

expressions e: $a|X|\boxed{e_0}(\boxed{e_1},\boxed{e_2},\ldots,\boxed{e_k})|fn(\boxed{e_1},\boxed{e_2},\ldots,\boxed{e_k})|p = e|\{\boxed{e_1},\ldots,\boxed{e_k}\}|[\boxed{e_1}|\boxed{e_2}]|$
$[\boxed{e_1},\ldots,\boxed{e_k}]|$begin l end|if $ic_1; ic_2;\ldots ic_k$ end|case $\boxed{e}$ of $cc_1; cc_2;\ldots; cc_k$ end|
fun $fc_1; fc_2;\ldots; fc_n$ end|catch(e)|throw$(\boxed{e})$|send$(\boxed{e},\boxed{e})$|receive $cc_1; cc_2;\ldots; cc_k$ end

expression lists l: $e_1, e_2, \ldots, e_k$

functions f: $fn\ fc_1; fn\ fc_2;\ldots; fn\ fc_n$.

programs P: $f_1 f_2 \ldots f_k$

Figure 1: The Erlang Subset Under Consideration

- *Each function consists of a single clause with just variables as arguments.*[1]
- *The return value of the function is bound to a return variable on each branch of the function body. The return variable is unique for each function.*

The preprocessing stage enforcing the named definition property yields a name for each use of a value, a property that is useful for performing data flow analysis and for presenting the data flow results.

4. CONCURRENT FLOW GRAPHS

The definition of concurrent flow graphs is given in two stages. Subsection 4.1 defines the basic properties of a concurrent flow graph. The correspondence between a program P and its flow graph V_P is defined in Subsec. 4.2.

4.1 Basic Flow Graph Definition

As ordinary flow graphs known from literature [24], concurrent flow graphs are given by sets of nodes and edges. These sets are partitioned into a number of subsets. Their definition is given in the following Definitions 2 and 3.

Essentially, each expression in a program is represented by an individual node.[2] These nodes are labeled to express all information given by the expression itself. In order to assign specific labels for the different kinds of expressions, different kinds of nodes are necessary.

[1]To enforce this, the case distinction of the different function clauses, and the value decomposition by their patterns need to be performed by a case-expression inside the single clause. For functions with arity > 1 the arguments and the corresponding patterns are structured into tuples of equal element number.

[2]This also holds in the case of nested sub-expressions, which have been eliminated by the preprocessing stage enforcing the named definition property.

DEFINITION 2 (NODES). *The set V of nodes of a concurrent flow graph is divided into the following subsets.*

- $V_{match} \subset V$ *denotes the set of all match nodes. A match node is labeled by a pattern LHS and a further node RHS $\in V$.*
- $V_{call} \subset V$ *denotes the set of all call nodes. A call node is labeled with a function call $e_0(e_1,\ldots e_k)$ or $fn(e_1,\ldots e_k)$. Each call node occurs as label of a match node in the flow graph.*
- $V_{spawn} \subset V$ *denotes the set of all spawn nodes. A spawn node is labeled with a function (given by module name and function name) and a sequence $e_1,\ldots,e_k$ of argument expressions.*
- $V_{branch} \subset V$ *denotes the set of all branching nodes. A branching node is labeled with a sequence $e_1,\ldots e_k$ of $k \geq 0$ tests, and for each branch with a sequence of k patterns, and a set of guards.*
- $V_{block} \subset V$ *denotes the set of all block nodes. A block node is labeled with a set of nodes.*
- $V_{catch} \subset V$ *denotes the set of catch nodes. A catch node is labeled with a further concurrent flow graph node n.*
- $V_{throw} \subset V$ *denotes the set of throw nodes. A throw node is labeled with an expression e.*
- $V_{send} \subset V$ *denotes the set of send nodes. A send node is labeled with two expressions, the* destination expression *e_d and the* message expression *e_m.*
- $V_{receive} \subset V$ *denotes the set of receive nodes. A receive node contains a set of branches, and is labeled with a pattern, and a sequence of guards for each branch.*

- *$V_{fun} \subset V$ denotes the set of fun nodes. A fun node is labeled with a function name and an arity.*[3]
- *$V_{import} \subset V$ denotes the set of import nodes. An import node is labeled with a list of variables.*
- *$V_{context} \subset V$ denotes the set of context nodes. A context node is labeled with a set of pairs $(Var, Defs)$ where Var is a variable v and $Defs$ is a list of references to nodes potentially assigning a value to v.*
- *$V_{return} \subset V$ denotes the set of return nodes. A return node is labeled with a variable.*
- *$V_{compute} = V \setminus (V_{match} \cup V_{call} \cup V_{spawn} \cup V_{branch} \cup V_{block} \cup V_{catch} \cup V_{throw} \cup V_{send} \cup V_{receive} \cup V_{fun} \cup V_{import} \cup V_{context} \cup V_{return})$ denotes the set of all computation nodes. Each node $n \in V_{compute}$ is labeled with an expression, that is not a match, a call, a branch (i.e. if, case), a begin, a catch, a throw, a send, a receive, or a fun.*

All described subsets of V are pairwise disjoint.

For concurrent flow graphs several different kinds of edges are necessary. Usually an edge in a flow graph describes the (directed) control flow, and data flow between two nodes. In concurrent flow graphs we distinguish two kinds of edges with this property. *Neighborhood edges* connect nodes whose represented expressions are adjacent in the source code. The non-local returns given by the catch-throw mechanism in Erlang are expressed by *throw edges* in concurrent flow graphs.

Call edges express function calls, and are special in the sense that they are bidirectional and represent both the control and data flow during a function call, *and* during the return from the call. (For the generation, we distinguish first order call edges and higher order call edges, depending on the call represented by their source node.)

Two further forms of edges just represent a data flow, but no control flow. *Spawn edges* essentially represent the data flow during the process generation. They do not represent an ordinary control flow, because the new process generated by them forms a new independent instance of control. *Message edges* finally stand for the data flow performed by the message passing mechanism between a send expression and a receive expression. No control flow occurs between the processes connected by a message edge.

The formal definitions of these kinds of edges are given by the following Def. 3.

DEFINITION 3 (EDGES). *The set E of edges of a concurrent flow graph is divided into the following subsets.*

- *$E_{call} \subset E$ denotes the set of call edges. Source of a call edge is a call node $n_s \in V_{call}$ with call arguments $a_1, \ldots a_k$; destination is an import node $n_d \in V_{import}$ with parameters $p_1, \ldots, p_n$ such that $k = n$. A call edge is labeled with the following information:*
  - *A set of assignments $p_i = a_i$ for $i \in \{1, \ldots, k\}$ called* parameter assignments.
  - *An assignment $u = r$ where r is the return variable of the function starting with n_d, and u is the pattern, occurring besides n_s as a label of a match node. This is called the* result assignment.

  Call edges are divided into first order call edges and higher order call edges, depending on the call represented by their source call node.
- *$E_{spawn} \subset E$ denotes the set of all spawn edges. Source of a spawn edge is a spawn node $n_s \in V_{spawn}$ labeled with the function expression f and the argument expressions $e_1, \ldots, e_k$; destination is the import node $n_d \in V_{import}$ of a function with parameters $p_1, \ldots, p_n$ fulfilling $k = n$ such that there is at least one execution of the containing program P with f denoting the destination function. A spawn edge is labeled with a set of parameter assignments $p_i = e_i$ for $i \in \{1, \ldots, k\}$.*
- *$E_{throw} \subset E$ denotes the set of all throw edges. A throw edge has a throw node as source, and a catch node as destination.*
- *$E_{message} \subset E$ denotes the set of all message edges. A message edge runs from a send node to a receive node.*
- *$E_{neighbor} = E \setminus (E_{call} \cup E_{spawn} \cup E_{throw} \cup E_{message})$ denotes the set of neighborhood edges. Neighborhood edges emerging from a branching node or a receive node are labeled with a clause number.*

The described subsets of E are pairwise disjoint.

Combining the definitions of nodes and edges, we get the following definition of a concurrent flow graph.

DEFINITION 4 (CONCURRENT FLOW GRAPH). *A concurrent flow graph is a pair $G = (V, E)$ where*

- *The set V is divided into subsets V_{match}, V_{call}, V_{spawn}, V_{branch}, V_{block}, V_{catch}, V_{throw}, V_{send}, $V_{receive}$, V_{fun}, V_{import}, $V_{context}$, V_{return}, and $V_{compute}$ according to Def. 2.*
- *The set E of edges is divided into the subsets E_{call}, E_{throw}, $E_{message}$, and $E_{neighbor}$ according to Def. 3.*

EXAMPLE 1 (CONCURRENT FLOW GRAPH). *A graphical example of a concurrent flow graph is given in Fig. 2. For simplicity reasons, the presentation is simplified as follows.*

- *The context nodes are empty for all the functions and are omitted in the graphical representation.*
- *Match nodes are marked with dotted lines inside the nodes they are labeled with.*

Call and spawn edges are marked by rounded corners. Receive nodes are marked by a triangle which is connected to a number of rows, each containing the pattern and the guards

[3]Note that a fun node does not correspond to a fun expression directly: the node is labeled with a name for the function instead of the function's clauses.

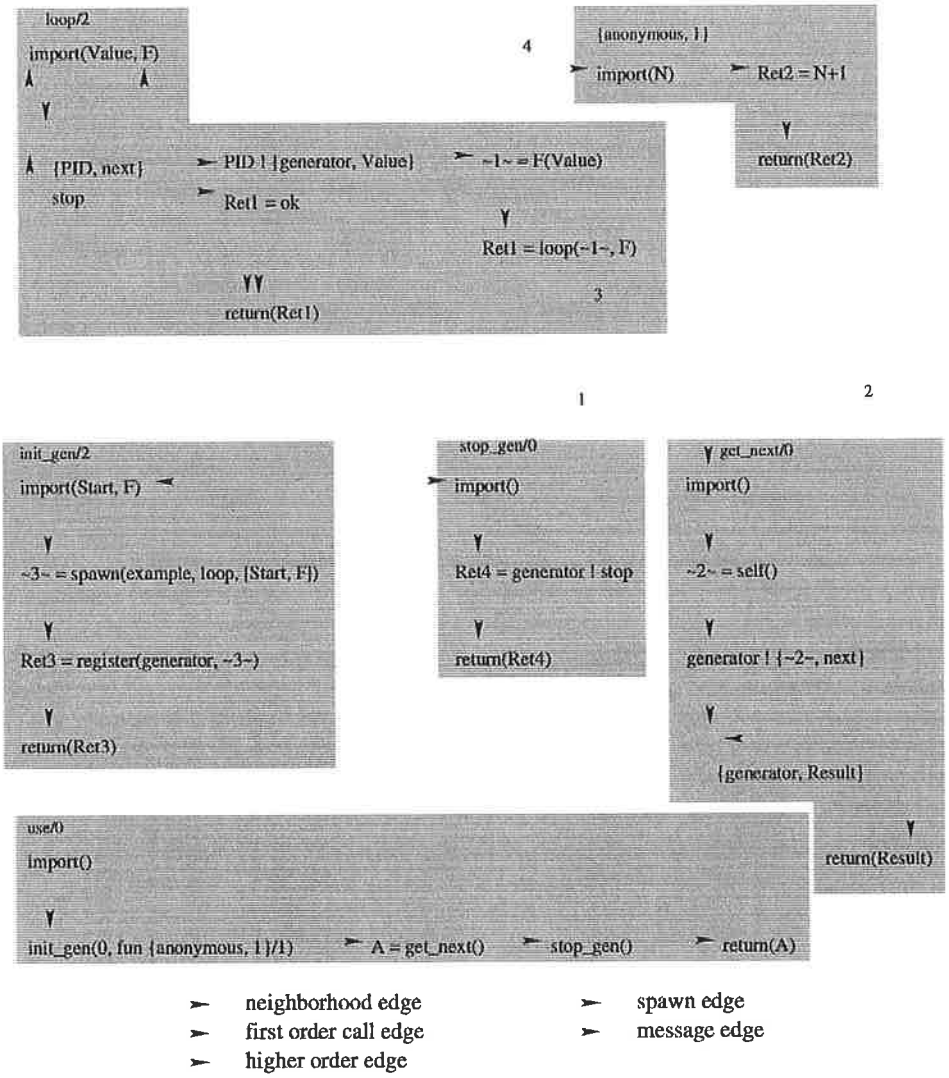


Figure 2: Flow graph of example.erl

for one clause. In order to simplify the identification of functions, the nodes of each function are contained in a gray box.

The numbers marking some of the edges are not of interest for the moment. They will be used later for describing the iterated edge computation by our implementation.

The graphical representation of the node forms not occurring in Ex. 1 is identical to sequential Erlang flow graphs [23].

4.2 The Concurrent Flow Graph of a Program

Given an Erlang program P fulfilling the named definition property, the following Definitions 5, 6, and 7 describe the concurrent flow graph G_P corresponding to P, i.e. the flow graph that can be used to represent P.

Note that the definition of G_P given here is not intended to provide an algorithm for computing G_P. Indeed, an algorithm will just be able to compute an approximation of G_P instead of G_P itself. The presentation of the implementation in Sec. 6 will discuss sources of inaccuracy in the computation, and the effect of the necessary approximations on the higher order call edges, spawn edges, throw edges, and message edges.

We start the presentations with the correspondence between the nodes in G_P, and the program expressions in P. Essentially, for each expression in the program a node is generated. The kind of node chosen depends on the structure of the expression. Additionally, each function definition is extended by an import node representing a local definition of the parameters of the function, a context node representing local definitions of the variables taken from the context of

the fun clause (this just applies to funs), and a return node representing the return to the calling program part.

DEFINITION 5 (CORRESPONDING NODES). *Let P be a program fulfilling the named definition property. The set V_P of corresponding nodes for P is generated by the following rules.*

- *For each expression e denoting a function call of the form* $\mathtt{e_0(e_1,\ldots e_k)}$ *or* $\mathtt{fn(e_1,\ldots e_k)}$ *V_P contains a call node $v \in V_P$ labeled with e.*
  In the special case of a call $\mathtt{fn(e_1,\ldots e_k)}$ *with* `fn` *denoting the BIF* ***spawn/3*** *or* ***spawn/4****, a spawn node is generated instead, which is labeled with the function and the call arguments given by the arguments of* ***spawn*** *at the corresponding positions.*
- *For each expression of the form* `if` $\mathtt{ic_1;\ldots;ic_k}$ *there exists a branching node $n \in V_P$ which is labeled with the guards of the individual clauses.*
- *For each expression of the form* `case e of` $\mathtt{cc_1;\ldots;cc_k}$ *there exists a branching node $n \in V_P$ which is labeled with the test e, and with patterns, and guards of the individual clauses.*
- *For expressions of the form* `begin l end` *a block node is introduced which is labeled with the nodes generated for the expressions in l.*
- *For each expression of the form* `catch(e)` *a catch node is generated and labeled with the node for the subexpression e.*
- *For each expression of the form* `throw(e)` *a throw node is generated and labeled with the subexpression e.*
- *For each expression of the form* $\mathtt{send(e_1,e_2)}$ *a send node is generated and labeled with e_1 as destination expression and e_2 as message expression.*
- *For each expression of the form* `receive` $\mathtt{cc_1;\ldots;cc_k}$ *there exists a receive node $n \in V_P$ which is labeled with the patterns and guards of the individual clauses.*
- *For every function definition in P and every expression of the form* `fun` $\mathtt{fc_1;\ldots;fc_k}$ *the following nodes are in V_P.*
  - *An import node labeled with the formal parameters of the function.*
  - *A context node labeled with all variables v, and references to the defining nodes n such that v is defined outside the function,*[4] *and the definition of v in n reaches a use within the function.*
  - *A branching node labeled with the patterns and guards of the individual function clauses.*
  - *A return node labeled with the return variable of the function.*

  For expressions of the form `fun` $\mathtt{fc_1;\ldots;fc_k}$ *a fun node is generated which is labeled with a generated function name and the function arity.*

[4]This only applies to funs. For named functions the context node is empty.

- *For every expression e of the form* $\mathtt{p = e'}$ *there is a match node $v \in V_P$ labeled with the pattern p and the node v' generated for e'.*
- *For each expression e of the form* `a`, `X`, $\mathtt{\{e_1,\ldots,e_k\}}$, $\mathtt{[e_1|e_2]}$, *or* $\mathtt{[e_1,\ldots,e_k]}$ *a computation node is generated and labeled with the expression e.*

Edges represent a control or data flow between the individual nodes. Their definition is based on the runtime behaviour of the program (which will be approximated for the computation of flow graphs).

DEFINITION 6 (CORRESPONDING EDGES). *Let P be a program fulfilling the named definition property, and V_P the set of corresponding nodes for P.*
Now let $n_1, n_2 \in V_P$ be nodes, and let e_1 and e_2 be the expressions in P the nodes n_1 and n_2 correspond to, respectively.[5]
The set E_P of corresponding edges for P consists of all edges generated by one of the following rules.

- *There exists a neighborhood edge from n_1 to n_2 in E_P if e_1 and e_2 belong to the same function f, and one of the following conditions holds.*
  - *n_1 is the import node, and n_2 the context node of f.*
  - *n_1 is the context node, and e_2 the first expression of f.*
  - *e_2 is the direct successor of e_1 in a sequence of expressions.*
  - *n_1 is a branching node or a receive node, and e_2 is the first expression in one of the clauses belonging to e_1. In this case the edge is labeled with the clause, e_2 belongs to.*
  - *e_1 is the last expression in a clause, and e_2 is the expression following the if, case or receive expression containing e_1.*
  - *e_1 is the last expression of one of the clauses of f (if the last expression e' is an if, case or receive expression, e_1 is the last expression in the body of one of the clauses of e'), and n_2 is the return node of f.*
- *There exists a call edge from n_1 to n_2 in E_P if n_1 is a call node, n_2 is the import node of a function f, and there exists an execution of P such that e_1 performs a call to f.*
- *There exists a spawn edge from n_1 to n_2 in E_P if e_1 is a spawn node, e_2 is the import node of a function f, and there exists an execution of P such that e_1 spawns a process starting its execution with f.*
- *There exists a throw edge from n_1 to n_2 in E_P if n_1 is a throw node, n_2 is a catch node, and there exists an execution of P such that e_1 throws a value that is caught by e_2.*

[5]If n_i is an import node, a context node, or a return node then e_i is undefined.

```
-module(example).
-export([loop/2, use/0]).

loop(Value, F) ->
    receive
        {PID, next} ->
            PID ! {generator, Value},
            loop(F(Value), F);
        stop -> ok
    end.

init_gen(Start, F) ->
    register(
        generator,
        spawn(example, loop, [Start, F])).

stop_gen() -> generator ! stop.

get_next() ->
    generator ! {self(), next},
        receive
            {generator, Result} -> Result
        end.

use() ->
    init_gen(0, fun(N) -> N + 1 end),
    A = get_next(),
    stop_gen(),
    A.
```

Figure 3: Erlang source code of example module

- *There exists a message edge from n_1 to n_2 in E_P if n_1 is a send node, n_2 is a receive node, and there exists an execution of P such that e_1 sends a message that is received by e_2.*

Combining corresponding nodes and corresponding edges of a program P, we get the concurrent flow graph G_P of P.

DEFINITION 7 (CORRESPONDING FLOW GRAPH). *Let P be a program fulfilling the named definition property. The corresponding flow graph for P is defined by $G_P = (V_P, E_P)$ where V_P is the set of corresponding nodes for P and E_P is the set of corresponding edges for P.*

EXAMPLE 2 (CORRESPONDING FLOW GRAPH). *Consider the module `example.erl` that is presented in Fig. 3, and that contains the following functions.*

- *`loop/2` forms a sequence generator that is meant to reside in an own process. It is initialized with the initial value, and the successor function of the sequence. For every message `{PID,next}` it sends the next sequence element to the process `PID`.*
- *`init_gen/2`, `stop_gen/0`, and `get_next/0` are the accessor functions for initializing, stopping and accessing the generator process.*
- *`use/0` is an example user of the sequence generator. It initializes the generator with the sequence of all natural numbers starting from 0, queries the first sequence element, stops the generator process and returns the element.*

The concurrent flow graph corresponding to `example.erl` is the one presented in Fig. 2.

5. DATA FLOW ANALYSIS

As stated by Shivers [17], the control flow given by a higher order program can depend on the data flow of the funs from their generation to their application in the program. In this section we therefore give a definition (adapted towards the use for concurrent flow graphs) of some base notions of data flow analysis, that are known for imperative languages [13]. For a definition of a variable v we write $def(v)$, for a use of v we write $use(v)$. The precise definitions of these notions are as follows.

DEFINITION 8 (DEFINITIONS). *Let G be a concurrent flow graph, and v a variable. A node n in G contains a definition of v if one of the following conditions holds.*

- *n is an import node, and v is one of the variables defined in n.*
- *n is a context node, and v is one of the variables defined in n. This is called an f-definition (denoted by f-def(v)).*
- *n is a match node denoting a matching LHS = RHS, v occurs in LHS, and there is at least one path w from the beginning of the function containing n to n itself such that v is not defined on w.*
- *n is a branching node, v occurs in at least one pattern p in n, and there is at least one path w from the beginning of the function containing n to n itself such that v is not defined on w.*
- *n is a receive node, v occurs in at least one pattern p in n, and there is at least one path w from the beginning of the function containing n to n itself such that v is not defined on w. This is called an m-definition (denoted by m-def(v)).*

A definition binding v to a value selected from a structure (either by pattern matching or the corresponding selection BIFs) is called an s-definition and denoted by s-def(v).

The opposite of the definition of a variable is given by its use. Intuitively, a use of a variable v is given by every expression that needs the value of v to be evaluated.

DEFINITION 9 (USES). *Let G be a concurrent flow graph, and v a variable. A node n in G contains a use of v if one of the following conditions holds.*

- *n is a node representing the expression E where*
  - $E = v$
  - $E = \{v_1, \ldots, v_k\}$, $E = [v_1|v_2]$, *or* $E = [v_1, \ldots, v_k]$ *with* $v = v_i$ *for some* i. *This is called an s-use and denoted by s-use*(v).
  - $E = v_0(v_1, \ldots, v_k)$, *or* $E = fn(v_1, \ldots, v_k)$ *with* $v = v_i$ *for some* $i \in \{0, \ldots, k\}$.
- *n is a branching node with a test given by v.*
- *n is a fun node, and there is at least one path w from the beginning of the function containing n to n itself such that v is defined on w. This is called an f-use and denoted by f-use*(v).
- *n is a match node denoting a matching LHS = RHS, v occurs in LHS, and there is at least one path w from the beginning of the function containing n to n itself such that v is defined on w.*
- *n denotes a branching node, or a receive node, v occurs in at least one pattern p in n, and there is at least one path w from the beginning of the function containing n to n itself such that v is defined on w.*
- *n is a send node and v occurs either as destination expression or as message expression. If v is the message expression, this is called an m-use and denoted by m-use*(v).

Some special information is added to the specification of a definition or use of a variable v inside a pattern p of a branching or receive node. Besides the branching/receive node n the number of the clause, the pattern p belongs to is stored. Occurrences of v in the patterns of several clauses of n are treated independently.

For the pairs, f-definition/f-use, s-definition/s-use, and m-definition/m-use we need to define the notion of *corresponding* uses and definitions: each of these kinds of use can *hide* a value from the data flow analysis. The corresponding definition makes the value available again (possibly under a different name).

- Corresponding f-uses and f-definitions express the situation of using a definition for a freezing it in a fun-generation. It is defrosted by the corresponding definition in the context node of the function.
- Corresponding s-uses and s-definitions express the situation of using a value to store it in a structure, and selecting it from there in the definition of a variable.
- Corresponding m-uses and m-definitions express the use of a value for sending it as a message, and the definition of a variable by receiving this message.

The precise definitions are as follows.

DEFINITION 10 (CORRESPONDING *f-use, f-def*). *Let v be a variable, u an f-use of v, and d an f-definition of v. u and d correspond to each other if the fun containing d is the one defined in u.*

DEFINITION 11 (CORRESPONDING *s-use, s-def*). *Let v be a variable, and u an s-use of v, generating a structure c. A selection d defining a variable v′ is an s-definition of v′ corresponding to u if there exists at least on run of the containing program P such that the structure decomposed in d is c, and the selected element position is the one containing the value of v.*

DEFINITION 12 (CORRESPONDING *m-use, m-def*). *Let v be a variable, and u an m-use of v in a node* n_u. *An m-definition d of some v′ in a node* n_d *corresponds to u if there is a message edge from* n_u *to* n_d *in the concurrent flow graph.*

Note that for an s-use, and the corresponding s-definition or for an m-use, and the corresponding m-definition the variable names usually differ.

The following main definition of this section states the situations under which a definition d reaches a use u.

DEFINITION 13. *Let d be a definition of a variable v, and u a use of a variable v′. Then d reaches u if one of the following properties holds.*

- $v = v'$ *and there is a path in the flow graph from d to u that does not contain a definition of v different from d. In this case d reaches u* directly.
- *There is a copy expression e of the form* $\tilde{v} = \tilde{v}'$ *such that d reaches the use of* $\tilde{v}'$ *in e and the definition of* $\tilde{v}$ *in e reaches u.*
- *d reaches an f-use of some* $\tilde{v}$ *and there is a corresponding f-definition of* $\tilde{v}$ *that reaches u.*
- *d reaches an s-use of v and there is a corresponding s-definition of some* $\tilde{v}$ *that reaches u.*
- *d reaches an m-use of v and there is a corresponding m-definition of some* $\tilde{v}$ *that reaches u.*

Besides the data flow coded in some of the nodes, the *labels of edges* can contain data flow information. This is the case for the parameter assignments given by call edges and spawn edges, and the result assignments of the call edges. These labels have to be taken into account for the data flow analysis. They are processed analogously to a sequence of nodes containing copy expressions when the edge is followed in the corresponding direction.

6. COMPUTATION OF CONCURRENT FLOW GRAPHS

For a program P fulfilling the named definition property the generation of the concurrent flow graph G_P consists of the following stages.

1. Generation of the set V_P of nodes according to Def. 5.

2. Computation of the set E_{neighbor} of neighborhood edges.

3. Computation of the call edges for first order function calls.

4. Computation of the call edges for higher order calls, the throw edges, and the message edges by an iterated process.[6]

Step (4) contains the computation of all edges that depend on the data flow in the program. It is necessary to iterate over all these edges because each new edge adds new opportunities for data flow in the graph.

The steps (1), (2), and (3) consist of a direct transfer of the corresponding definitions. They can be implemented in a straightforward manner. A detailed description of Step (4) is given in the following Subsec. 6.1.

6.1 Iterated Edge Computation

The generation of edges is implemented in form of three functions.

1. The computation of higher order call edges is done by a function

   $$\texttt{ho\_call\_edges(Graph1)} \rightarrow \{\texttt{Graph2}, \texttt{Bool}\}$$

2. For computing the throw edges in the concurrent flow graph, we use a function

   $$\texttt{throw\_edges(Graph1)} \rightarrow \{\texttt{Graph2}, \texttt{Bool}\}$$

3. The computation of the message edges is done by a function

   $$\texttt{message\_edges(Graph1, Process)} \rightarrow \{\texttt{Graph2}, \texttt{Bool}\}$$

Each function expects a flow graph as input and returns a tuple containing a new flow graph and a boolean flag whether any new edges have been introduces. For the computation of the message edges in `message_edges/2` a description of the initial process, essentially given by its initial call (or a list of potential initial calls), must be provided as additional argument.

The main loop `introduce_edges/2`, which is presented in Fig. 4, loops over the three functions until no change was made by any of them in one step.

In the remaining presentation of `ho_call_edges/1`, `throw_edges/1`, and `message_edges/2` we omit the boolean flag for changes in the return value in order to simplify the presentation. In the following, the high level structure of the functions is presented, but we omit several of the called functions. The names of the omitted functions are chosen to represent their general semantics.

[6]Spawn edges behave very similar to higher order call edges during the computation. To simplify the following presentation, spawn edges are not discussed explicitely. Their creation is done together with the higher order call edges in an analogous manner.

```
introduce_edges(Graph, InitProcess) ->
   % generate new edges
   {GraphWithCall, CallChangeFlag} =
      ho_call_edges(Graph),
   {GraphWithThrow, ThrowChangeFlag} =
      throw_edges(GraphWithCall),
   {GraphWithMessage, MessageChangeFlag} =
      message_edges(GraphWithThrow, InitProcess),
   % check whether a further step is necessary
   if
      CallChangeFlag;
      ThrowChangeFlag;
      MessageChangeFlag ->
          % next iteration with new graph
          introduce_edges(
              GraphWithMessage, InitProcess);
      true ->
          % return graph with computed edges
          GraphWithMessage
   end. % if
```

Figure 4: computation of data flow dependent edges

- `ho_call_edges/1` is presented in Fig. 5. For each higher order function call it analyzes the variable in the function position. For each reaching definition which denotes a function within the graph,[7] a call edge is introduced.

- `throw_edges/1` is presented in Fig. 6. For each catch node c, it determines the set of throw nodes t such that there exists a path from c to t without another catch node in-between. For each such t a throw edge from t to c is introduced.

- `message_edges/1` is presented in Fig. 7. First, it calculates the set of processes from the initial process. This is done by an iterated analysis of the spawn nodes reachable from the already known processes. For each send node s in the graph, the following steps are performed afterwards.

  1. The first argument of the send node is analyzed to determine the potential destination processes from the set of all processes. This is done by data flow analysis and partial evaluation on the reaching definitions.

  2. The variable v in the second argument of the send node is tested against the receive statements of the destination processes: if one of the definitions reaching the use of v in the send node matches one of the patterns of the receive r then a message edge from s to r is inserted into the graph.

EXAMPLE 3. *We reconsider the module* `example.erl` *presented in Fig. 3, and its corresponding concurrent flow graph*

[7]This property is checked using partial evaluation.

```
ho_call_edges(Graph) ->
    % extract higher order calls
    Sources = ho_calls(Graph),
    % compute and insert edges for each call independently
    foldl(fun(Call, GraphAcc) -> edges_from_call(Call, GraphAcc) end, Graph, Sources).

edges_from_call(Call, Graph) ->
    % compute all reaching definitions of function position in call that denote a function
    Dest = filter(fun is_function/1, reaching_definitions(extract_called_function(Call))),
    % insert edges from Call to each element of Dest into Graph
    foldl(fun(SingleDest, GraphAcc) -> insert_call_edge(Call, SingleDest, GraphAcc) end, Graph, Dest).
```

Figure 5: Computation of higher order call edges

```
throw_edges(Graph) ->
    % extract catch nodes from graph
    Dest = catch_nodes(Graph),
    % compute and insert the throw edges for each catch node independently
    foldl(fun(Catch, GraphAcc) -> edges_to_catch(Catch, GraphAcc) end, Graph, Dest).

edges_to_catch(Catch, Graph) ->% compute all throw nodes that are reachable from the current
                                % catch without a further catch on the path
    Source = filter(fun(PossibleSource) ->
                        % source is throw node, no other catch between Catch and trow node
                        is_throw_node(PossibleSource)
                        and not(catch_on_each_path(Catch, PossibleSource))
                    end, nodes_reached_from(Catch)),
    % insert an edge from each element of Source to Catch into Graph
    foldl(fun(SingleSrc, GraphAcc) -> insert_throw_edge(SingleSrc, Catch, GraphAcc) end, Graph, Source).
```

Figure 6: Computation of throw edges

```
message_edges(Graph, InitProcess) ->
    % compute processes starting from InitProcess
    Processes = compute_all_processes(Graph, InitProcess),
    % extract send nodes from graph
    Source = send_nodes(Graph),
    % compute and insert the throw edges for each send node independently
    foldl(fun(Send, GraphAcc) -> edges_from_send(Send, GraphAcc, Processes) end, Graph, Source).

edges_from_send(Snd, Graph, Processes) ->
    % compute all processes that can be the destination of the current send
    DestProc = filter(fun(PossibleDest) -> member(PossibleDest, Processes) end,
                      reaching_definitions(extract_send_destination(Snd))),
    % compute set of sent messages
    Messages = reaching_definitions(extract_send_message(Snd)),
    % compute all receives in all destination processes that match an element of Messages
    Dest = filter(fun(Receive) -> receive_matches_value(Receive, Messages) end,
                      % test all receives of all destination processes
                      append(map(fun get_process_receive_nodes/1, DestProc)))),
    % insert edges from Snd to each element of Dest into Graph
    foldl(fun(SingleDest, GraphAcc) -> insert_message_edge(Snd, SingleDest, GraphAcc) end, Graph, Dest).
```

Figure 7: Computation of message edges

presented in Fig. 9. The graph generation in this case consists of the following steps.

1. *The nodes, the neighborhood edges, and the first order call edges (including the spawn edge) are generated.*
2. *During the first iteration of edge generation the following edges are introduced.*
   - *The higher order call edge marked with* 4.
   - *The message edges marked with* 1 *and* 2.
3. *During the second iteration the message edge marked with* 3 *is introduced. It is delayed to this step, because it makes use of the data flow of variable* `~2~` *to* `PID` *on the message edge marked with* 2.
4. *The third iteration does not introduce any further edges. Therefore the iteration process terminates.*

6.2 Sources of Inaccuracy

The algorithm presented previously in this section cannot compute the concurrent flow graph according to Def. 7 exactly because of a number of sources of inaccuracy during the computation. All these inaccuracies just affect the sets of higher order call edges, throw edges, and message edges. The computation of the nodes, the neighborhood edges, and the first order call edges does not depend on the data flow analysis, and can be done in a precise manner.

In detail the data flow analysis is affected by the following effects.

- Distant conditionals in a program can correspond in a way, that only certain combinations of subpaths can occur in a path. For instance, consider two functions f, g that are called with the same argument n, and both contain different branches for even and for odd values of n. A path through the program that takes the even branch in f but the odd branch in g for the same n is not possible. Such situations are not recognized by the system.
- The control flow is based on 0CFA [17], i.e. different function closures sharing the same source code are identified. When distinguishing these functions, several nodes are necessary for what is represented by one node in our approach. Especially, for higher order function calls, each of these nodes could be more specific in the sense of having less outgoing call edges than the one node being the source for the union of all these call edges in our concurrent flow graphs.
- In Erlang the function position of a higher order function call can be given by a fun, or a representation of the *name* of the called function. The second case is problematic because partial evaluation is necessary to identify the called functions. This partial evaluation must, however, be approximate, especially if the needed information is not completely available before runtime. (Example: the name of a called function is read from the keyboard at runtime.)
- Parts of the data flow can escape the scope of the flow graph. In that case approximating assumptions must be used. For example this is the case when a function f is passed to a function g as argument, but g is not part of the code under testing. In this case we cannot be sure whether f is called from g, or not.
- The test whether a definition matches a pattern of a receive statement must approximate the value of the definition. Therefore the computation of the message edge destinations is approximate.

Approximation is done according to the following policy. Whenever there is any doubt whether an edge is necessary, the edge is introduced. This guarantees the concurrent flow graph to contain as much control and data flow alternatives as possible.

If there is, however, no information about the alternatives at a certain point (e.g. when reading a name of the function to be called from the keyboard) we do not insert edges to all possible destinations (e.g. to all functions of the right arity occurring in the flow graph).

6.3 Implementation

The computation of concurrent flow graphs for Erlang programs is implemented based on OTP R9C-2, i.e. both the implemented code, and the programming constructs expected in the analyzed program are based on this language standard.[8]

The flow graph structure is essentially based on the parse result of the OTP library module `epp.erl`. This module is used to read the source code modules under testing, and the result format is preserved during the preprocessing stage enforcing the named definition property. The flow graph consists of a list of modules, each given by the result of `epp:parse_file/3` which is modified in the following steps to form the flow graph.

The node generation consists of a traversal of the code performing the following tasks.

- In each expression representing a node, the line number entry of the `epp` output is changed to a tuple additionally holding a node number, and some local data flow information.
- For each fun generation consisting of function clauses, a new function name is generated, which replaces the clauses in the definition. The clauses are taken to represent the generated function in the flow graph.
- Some of the structures denoting special expressions are replaced or extended.
  - Calls to the BIF `throw/1` are replaced by a new kind of tuple structure.

[8] At the moment, the implementation lacks the handling of list comprehensions, that are a bit tedious to cope with, but do not provide any new problems or insight. Code for handling them will be added, once the restricted prototype is finished.

Matrices for function calls are extended by a field for the call edge information.

  - Receive entries are extended by a field for information on the message edges.

- For each function, the additional import, context and return nodes are introduced, which are represented by tuples similar to those returned by epp.

Some further information is pre-calculated and stored for future use. Among others, a list of all call nodes (and other important node types) is stored for each function, and an index given by a balanced tree is generated for each module for accessing the individual nodes by their number.

In a next step, the call nodes are divided into first order calls and higher order calls. For the first order calls, the destinations can be computed easily, and the resulting edge information is stored in the prepared field of the call.

For the higher order call edges, the spawn edges, the throw edges, and the message edges, the computation is done as described in Subsec. 6.1, and the computation results are coded into the prepared fields of the structures.

7. CONCLUSION AND FUTURE WORK

By adapting the notion of flow graphs to functional programs written in Erlang, especially containing the concurrent constructs integrated in the Erlang standard, we have made a large step towards having the wide area of source code directed testing (which is heavily used in industry) accessible for functional programming.

As already stated for flow graphs for sequential Erlang [23], function calls have a strong influence on the control flow in functional programs comparable to the looping constructs in imperative languages. A refined definition of call edges is given here, providing a notion of expressing the whole control flow of a function call, namely the jump to a distant piece of code and the return to the calling code piece after processing the function call.

When considering higher order functional programs, we get the additional problem that we need data flow analysis in order to determine the set of functions that is possibly called at a certain program point [23]. Throw edges and message edges depend (like higher order call edges and spawn edges) on the possible control flow in the program and therefore on the computed higher order call edges. They, however, cause additional data flow opportunities and can therefore extend the set of higher order call edges in a program. An iteration looping over the generation of higher order call edges (with spawn edges), throw edges and message edges has been described that computes all these edges, and terminates when a fixpoint is reached.

Future work towards a coverage system based on concurrent flow graphs consists of two steps. First, a tracing tool storing the control flow through the tested modules during a test case execution [22] must be extended to handle several processes, and to store information on the data flow generated by passing messages between tested modules. As a second step, a tool analyzing given traces for their coverage level must be implemented. While a simple node coverage test is already finished, we expect data flow oriented coverage [21] to be of special use in the context of concurrent Erlang programs, because data flow coverage is the only level of coverage analysis that is able to consider messages passed around the program.

8. REFERENCES

[1] J. M. Ashley and R. K. Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 20(4):845–868, July 1998.

[2] R. H. Carver and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Software*, 8(2):66–74, 1991.

[3] O. Chitil. A semantics for tracing. In *Draft Proceedings of the 13th International Workshop on Implementation of Functional Languages, IFL*, 2001.

[4] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP'00)*, volume 35.9 of *ACM Sigplan Notices*, pages 268–279, N.Y., Sept. 18–21 2000. ACM Press.

[5] Ericsson Utvecklings AB. *Erlang Reference Manual, Version 5.3*, 2003.

[6] *Tools version 2.3*. Documentation of Erlang/OTP R9C.

[7] M. Factor, E. Farchi, and Y. Talmor. Timing dependent bugs. In *Software Testing Analysis and Review (STAR98)*, May 1998.

[8] A. Gill. Debugging Haskell by observing intermediate data structures. In *Proceedings of the 4th Haskell Workshop. Technical report of the University of Nottingham*, 2000.

[9] P. Godefroid. Model checking for programming languages using VeriSoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[10] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2), Mar. 2005.

[11] G.-H. Hwang, K.-C. Tai, and T.-L. Huang. Reachability testing: An approach to testing concurrent software. In *Proc. First Asia-Pacific Software Engineering Conference (APSEC)*, Tokyo, Japan, Dec. 1994.

[12] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[13] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[14] G. Rothermel, M. Burnett, L. Li, C. DuPuis, and A. Sheretov. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Methodology*, 10(1):110–147, 2001.

[15] G. Rothermel, L. Li, C. DuPuis, and M. Burnett. What you see is what you test: A methodology for testing form-based visual programs. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 198–207. IEEE Computer Society Press/ACM Press, 1998.

[16] K. J. Rothermel, C. R. Cook, M. M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel. WYSIWYT testing in the spreadsheet paradigm. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 230–239. ACM Press, June 2000.

[17] O. Shivers. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[18] S. D. Stoller. Testing concurrent java programs using randomized scheduling. In *Proceedings of the Second Workshop on Runtime Verification (RV)*, volume 70 of *Electronic Notes in Theoretical Computer Science*, 2002.

[19] K. van den Berg. *Software Measurement and Functional Programming*. 1995.

[20] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: a new hat. In *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop, Firenze, Italy*, pages 151–170, 2001.

[21] M. Widera. Data flow considerations for source code directed testing of functional programs. In H.-W. Loidl, editor, *Draft Proceedings of the Fifth Symposium on Trends in Functional Programming*, Nov. 2004.

[22] M. Widera. Flow graph interpretation for source code directed testing of functional programs. In C. Grelck and F. Huch, editors, *Implementation an Application of Functional Languages, 16th International Workshop, IFL'04*, Technischer Bericht 0408. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, 2004.

[23] M. Widera. Flow graphs for testing sequential Erlang programs. In *Proceedings of the 3rd ACM SIGPLAN Erlang Workshop*. ACM Press, 2004.

[24] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.

ERICSSON

FIFO Run-to-completion Event-based Programming Considered Harmful

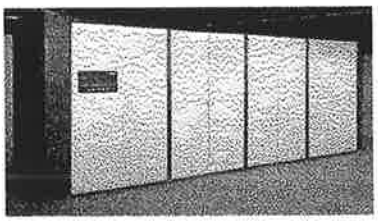
Structured Network Programming

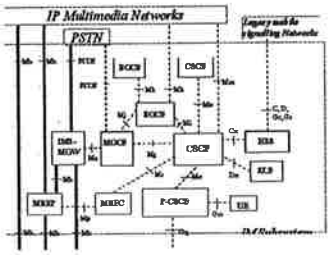
Ulf Wiger
Senior Software Architect
Ericsson AB, IMS Gateways
<ulf.wiger@ericsson.com>

EUC 2005
10 November 2005

Trend: monoliths ⇒ networks of loosely coupled components.

- ⇒ stateful multi-way communication, delay issues and partial system failures
- No common insight yet into how this affects SW complexity (suspect that most people think it simplifies things...)





http://www.3gpp.org/ftp/Specs/archive/23_series/23.002/

1

Claims

1. Ability to filter messages with implicit buffering ("selective receive") is vital for proper state encapsulation.
   - Otherwise, complexity explosion is inevitable (in certain situations.)
2. Inline selective receive keeps the logical flow intact – no need to maintain your own "call stack".

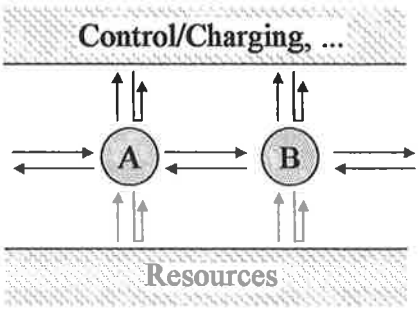
(1) is more important than (2).

The ability to implement complex state machines well will most likely become a key competitive edge.

Example Scenario

- Each "session" is represented by one or more stateful processes (as in CSP)
- Each control process interacts with multiple uncoordinated message sources
- Message sequences may (and invariably will) interleave

Traditional "Half-Call" model



A = originating side
B = terminating side

2

FIFO, Run-To-Completion (RTC) semantics:

- Thread of control owned by central event loop
- For each message, an associated method is called
- The method executes, then returns control to the main loop
- Typically, the event loop dispatches messages for multiple "process" instances
  => an instance may never block.
- Examples: UML, common C++ pattern, OHaskell

Selective Receive semantics

- Each process instance specifies a subset of messages that may trigger method dispatch at any given time
- If the process owns the thread of control, this is done in a blocking "system call" (e.g. the 'receive ... end' language construct in Erlang, or the select() function in UNIX.

Selective receive is not a new concept

- The select() system call first appeared in 4.2BSD 1982
  - Allowed for blocking wait on a set of file descriptors.
  - (Needs to be coupled with e.g. getmsg() in order to fetch the message.)
  - Now supported by all unices.
- MPI* has API support for blocking selective receive.
- Erlang was first presented in 1987.

*http://www-unix.mcs.anl.gov/mpi/

Asynchronous Programming still dominates – why?

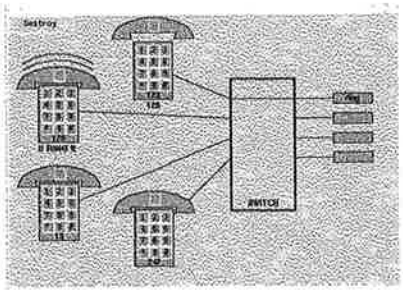
- Synchronous programming is considered slow.
- Reasoning about event-based programming seems easier.
- Easy to build a fast, simple event-based prototype.
- It's not clear what you give up by not programming synchronously!
- *(and blocking RPC is not the whole secret – selective receive is the powerful enabler.)*

4

Programming Experiment

- Demo system used in Ericsson's Introductory Erlang Course (assignment: write a control program for a POTS subscriber loop)
- We will re-write the control loop using different semantics.
- Note well: we don't handle errors in our example (usually the most complex part.)

"POTS": Plain Ordinary Telephony System – Trivial schoolbook example of telephony (as simple as it gets)



Demo...

POTS Control Loop – Original Impl. (1/3)

```
start() -> idle().

idle() ->
  receive
     {?lim, offhook} ->
         lim:start_tone(dial),
         getting fi
     {?lim, {di
         idle()
     {?hc, {req
         Pid !
         lim:st
         ringin
     Other ->
         io:for
         idle()
    end.
```

inline selective receive

Synchronous HW control

```
start_tone(Tone)->
  call({start_tone, Tone}).

call(Request) ->
  Ref = make_ref(),
  lim ! {request, Request, Ref, self()},
  receive
    {?lim, Ref, {_ReplyTag, Reply}} ->
        Reply
  end.
```

5

POTS Control Loop – Original Impl. (2/3)

```
getting_first_digit() ->
   receive
     {?lim, onhook} ->
         lim:stop_tone(),
         idle();
     {?lim, {digit, Digit}} ->
         lim:stop_tone(),
         getting_number(Digit,
             number:analyse(Digit, number:valid_sequences()));
     {?hc, {request_connection, Pid}} ->
         Pid ! {?hc, {reject, self()}},
         getting_first_digit();
     Other ->
         io:format("Unknown message ...: ~p~n", [Other]),
         getting_first_digit()
   end.
```

POTS Control Loop – Original Impl. (3/3)

```
calling_B(PidB) ->
    receive
     {?lim, onhook} ->
         idle();
     {?lim, {digit, _Digit}} ->
         calling_B(PidB);
     {?hc, {accept, PidB}} ->
         lim:start_tone(ring),
         ringing_A_side(PidB);
     {?hc, {reject, PidB}} ->
         lim:start_tone(busy),
         wait_on_hook(true);
     {?hc, {request_connection, Pid}} ->
         Pid ! {?hc, {reject, self()}},
         calling_B(PidB);
     Other ->
         io:format("Got unknown message...: ~p~n",[...]),
         calling_B(PidB)
    end.
...
```

Experiment:
Rewrite the program using
an event-based model

Event-based vsn, blocking HW control (1/3)

```
%% simple main event loop with FIFO semantics
event_loop(M, S) ->
   receive
      {From, Event} ->
          dispatch(From, Event, M, S);
      {From, Ref, Event} ->
          dispatch(From, Event, M, S);
      Other ->
          io:format("Unknown msg: ~p~n", [Other]),
          exit({unknown_msg, Other})
    end.

dispatch(From, Event, M, S) when atom(Event) ->
   {ok, NewState} = M:Event(From, S),
   event_loop(M, NewState);
dispatch(From, {Event, Arg}, M, S) ->
   {ok, NewState} = M:Event(From, Arg, S),
   event_loop(M, NewState).
```

7

Event-based vsn, blocking HW control (2/3)

```
offhook(?lim, #s{state = idle} = S) ->
   lim:start_tone(dial),
   {ok, S#s{state = getting_first_digit}};
offhook(?lim, #s{state = {ringing_B_side, PidA}} = S) ->
   lim:stop_ringing(),
   PidA ! {?hc, {connect, self()}},
   {ok, S#s{state = {speech, PidA}}};
offhook(From, S) ->
   io:format("Unknown message in ~p: ~p~n",
             [S#s.state, {From, offhook}]),
   {ok, S}.
```

Synchronous HW control

Event-based vsn, blocking HW control (3/3)

```
onhook(?lim, #s{state = getting_first_digit} = S) ->
   lim:stop_tone(),
   {ok, S#s{state = idle}};
onhook(?lim,#s{state={getting_number,{_Num,_Valid}}} = S) ->
   {ok, S#s{state = idle}};
onhook(?lim, #s{state = {calling_B, _PidB}} = S) ->
   {ok, S#s{state = idle}};
onhook(?lim, #s{state = {ringing_A_side, PidB}} = S) ->
   PidB ! {?hc, {cancel, self()}},
   lim:stop_tone(),
   {ok, S#s{state = idle}};
onhook(?lim, #s{state = {speech, OtherPid}} = S) ->
   lim:disconnect_from(OtherPid),
   OtherPid ! {?hc, {cancel, self()}},
   {ok, S#s{state = idle}};
...
```

A bit awkward
(FSM programming "inside-out"),
but manageable.

Add the non-blocking restriction

(first, naive, implementation)

Now, assume we are not allowed to block (common restriction, 1/3)

Asynchronous HW control

```
offhook(?lim, #s{state = idle} = S) ->
   lim_asynch:start_tone(dial),
   {ok, S#s{state = {{await_tone_start,dial},
                     getting_first_digit}}};
offhook(?lim, #s{state = {ringing_B_side, PidA}} = S) ->
   lim_asynch:stop_ringing(),
   PidA ! {?hc, {connect, self()}},
   {ok, S#s{state = {await_ringing_stop, {speech, PidA}}}};
offhook(?lim, S) ->
   io:format("Got unknown message in ~p: ~p~n",
       [S#s.state, {lim, offhook}]),
   {ok, S}.
```

9

... not allowed to block (2/3)

```
digit(?lim, Digit, #s{state = getting_first_digit} = S) ->
    %% CHALLENGE: Since stop_tone() is no longer a synchronous
    %% operation, continuing with number analysis is no longer
    %% straightforward. We can either continue and somehow log that
    %% we are waiting for a message, or we enter the state await_tone_stop
    %% and note that we have more processing to do. The former approach
    %% would get us into trouble if an invalid digit is pressed, since
    %% we then need to start a fault tone. The latter approach seems more
    %% clear and consistent. NOTE: we must remember to also write
    %% corresponding code in stop_tone_reply().
    lim_asynch:stop_tone(),
    {ok, S#s{state = {await_tone_stop,
                      {continue, fun(S1) ->
                                     f_first_digit(Digit, S1)
                                 end}}}};
```

...not allowed to block (3/3)

```
start_tone_reply(?lim, {Type, yes},
   #s{state = {{await_tone_start, Type}, NextState}} = S) ->
   {ok, S#s{state = NextState}}.

stop_tone_reply(?lim,_,#s{state={await_tone_stop,Next}} =S) ->
   %% CHALLENGE: Must remember to check NextState. An alternative would
   %% be to always perform this check on return, but this would increase
   %% the overhead and increase the risk of entering infinite loops.
   case NextState of
      {continue, Cont} when function(Cont) ->
          Cont(S#s{state = Next});
      _ ->
          {ok, S#s{state = Next}}
   end.
```

Quite tricky, but the program still isn't timing-safe. (Demo...)

Global State-Event Matrix

FIFO semantics, asynchronous hardware API

	idle	getting first digit	getting number	calling B	ringing A-side	speech	ringing B-side	wait on-hook	await tone start	await tone stop	await ringing start	await ringing stop	await pid with telnr	await conn-ect	await dis-connect
offhook	O	X	X	X	X	X	O	X	X	X	D	X	X	X	X
onhook	X	O	O	O	O	O	O	O	D	D	D	D	D	D	D
digit	—	O	O	—	—	—	—	—	D	D	D	D	D	D	—
connect	—	—	—	—	O	—	—	—	D	X	X	X	X	X	X
request connection	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
reject	—	—	—	O	—	—	—	—	X	X	X	X	X	X	X
accept	—	—	—	O	—	—	—	—	X	X	X	X	X	X	X
cancel	—	—	—	—	—	—	—	—	X	D	D	D	X	D	X
start tone reply	X	X	X	X	X	X	X	X	O	X	X	X	X	X	X
stop tone reply	X	X	X	X	X	X	X	X	X	O	X	X	X	X	X
start ringing reply	X	X	X	X	X	X	X	X	X	X	O	X	X	X	X
stop ringing reply	X	X	X	X	X	X	X	X	X	X	X	O	X	X	X
pid with telnr reply	X	X	X	X	X	X	X	X	X	X	X	X	O	X	X
connect reply	X	X	X	X	X	X	X	X	X	X	X	X	X	O	X
disconnect reply	X	X	X	X	X	X	X	X	X	X	X	X	X	X	O

Apparent Problems

- The whole matrix needs to be revisited if messages/features are added or removed.
- What to do in each cell is by no means obvious – depends on history.
- What to do when an unexpected message arrives in a transition state is practically never specified (we must invent some reasonable response.)
- Code reuse becomes practically impossible.

Non-blocking version, using message filter (1/2)

```
digit(?lim, Digit, #s{state = getting_first_digit} = S) ->
   %% CHALLENGE: ...<same as before>
   Ref = lim_asynch:stop_tone(),
   {ok, S#s{state = {await_tone_stop,
                     {continue, fun(S1) ->
                                   f_first_digit(Digit, S1)
                                end}}},
    #recv{lim = Ref, _ = false}};
```

Accept only msgs tagged with Ref, coming from 'lim'; buffer everything else.

The continuations are still necessary, but our sub-states are now insensitive to timing variations.

Non-blocking version, using message filter (2/2, the main event loop)

```
event_loop(M, S, Recv) ->
   receive
      {From, Event} when element(From, Recv) == [] ->
          dispatch(From, Event, M, S);
      {From, Ref, Event} when element(From, Recv) == Ref ->
          dispatch(From, Event, M, S);
      {From, Ref, Event} when element(From, Recv) == [] ->
          dispatch(From, Event, M, S)
   end.

dispatch(From, Event, M, S) when atom(Event) ->
    handle(M:Event(From, S), M);
dispatch(From, {Event, Arg}, M, S) ->
    handle(M:Event(From, Arg, S), M).

handle({ok, NewState}, M) ->
    event_loop(M, NewState);
handle({ok, NewState, Recv}, M) ->
    event_loop(M, NewState, Recv).
```

Properties of filtered event loop

- Can be implemented in basically any language (e.g. extending existing C++ framework.)
- Solves the complexity explosion problem.
- Doesn't eliminate the need for continuations (this affects readability – not complexity)

Real-Life Example

Code extract from the AXD301-based "Mediation Logic" (ML)

```
%% We are waiting to send a StopTone while processing a StartTone and now
%% we get a ReleasePath. Reset tone type to off and override StopTone
%% with ReleasePath since this will both clear the tone and remove connection.
cm_msg([?CM_RELEASE_PATH,TransId,[SessionId|Data]] = NewMsg,
       HcId, #mlgCmConnTable{
                 sessionId = SessionId,
                 sendMsg = ?CM_START_TONE_RES,
                 newMsg = {cm_msg,
                           [?CM_STOP_TONE|Msg]}} = HcRec,
       TraceLog) ->
   NewHcRec = HcRec#mlgCmConnTable{
                      newMsg = {cm_msg, NewMsg},
                               toneType = off},
   NewLog = ?NewLog({cm_rp, 10}, {pend, pend}, undefined),
   mlgCmHccLib:end_session(
       pending, NewHcRec, [NewLog | TraceLog], override);
```

Real-Life Example

Code extract from the AXD301-based "Mediation Logic" (ML)

```
%% If we are pending a Notify Released event for a Switch Device, override
%% with ReleasePath.
cm_msg([?CM_RELEASE_PATH,TransId,[SessionId|Data]] = NewMsg,
       HcId,
       #mlgCmConnTable{
           sessionId = SessionId,
           newMsg = {gcp_msg, [notify, GcpData]},
           deviceType = switchDevice,
           path2Info = undefined} = HcRec,
       TraceLog) ->
   NewHcRec = HcRec#mlgCmConnTable{newMsg= {cm_msg, NewMsg}},
   NewLog = ?NewLog({cm_rp, 20}, {pend, pend}, undefined),
   mlgCmHccLib:end_session(
        pending, NewHcRec, [NewLog | TraceLog], override);
```

Real-Life Example

Code extract from the AXD301-based "Mediation Logic" (ML)

```
%% Getting a ReleasePath when pending a Notify Released event is a bit
%% complicated. We need to check for which path the ReleasePath is for and
%% for which path the notify is for. If they are for different paths we are
%% in a dilemma since we only can be in pending for one of them. As a simple
%% way out we just treat this as an abnormal release for now.
cm_msg([?CM_RELEASE_PATH,TransId,[SessionId|Data]] = NewMsg,
       HcId,
       #mlgCmConnTable{
           sessionId = SessionId,
           newMsg = {gcp_msg, [notify, GcpData]},
           deviceType = switchDevice} = HcRec,
       TraceLog) ->
   mlgCmHcc:send_cm_msg(?CM_RELEASE_PATH_RES,
                        ?MSG_SUCCESSFUL, TransId, SessionId),
   NewHcRec = HcRec#mlgCmConnTable{newMsg = abnormal_rel},
   NewLog = ?NewLog({cm_rp, 30}, {pend, pend}, undefined),
   mlgCmHccLib:end_session(pending, NewHcRec,
                           [NewLog | TraceLog], override);
```

Observations

- Practically impossible to understand the code without the comments
- Lots of checking for each message to determine exact context (basically, a user-level call stack.)
- A nightmare to test and reason about
- (The production code has now been re-written and greatly simplified.)

ML State-Event Matrix (1/4)

State / Triggers	Null	Setup	Add	Connected	Release	ModifyConn	Modify	ToneActive	CotActive	Override	Pending	Seized	Move	Prepare	Broken
EstablishPath	1,2	y	y	40, 41, 42, 43, 44	84	y	y	123, 124	129, 130	y	y	213	220, y	y	235, 236, 237, 259
ModifyPath	y	y	y	45, 46	y	y	y	y	y	y	y	y	y	y	y
ReleasePath	4	13	y	47, 48, 49, 50, 51, 52	85, 86	13	13	y	131	134, 135, 136	150, 151, 152	y	13	13	238
StartTone	5	y	y	53	87	y	y	125	y	y	y	y	y	y	239
StopTone	5	y	y	54	88	y	111, 112, 113	126	y	137	y	y	y	y	240
PreparePath	254	y	y	55	y	y	y	y	y	y	y	y	y	y	y
BreakPath	y	y	y	56	y	y	y	y	y	y	y	y	y	y	y
SeizeDevice	6	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
ReleaseDevice	7	13	13	57, 58	89	13	13	NA	NA	138	153, 154, 155, 156	214	13	NA	241

Action procedures:

N/A	Not applicable
x	No action, ignore the error
y	Return protocol error, remain in same state
A	Anomaly, log

Alternative execution paths depending on context

ML State-Event Matrix (2/4)

State / Triggers	Null	Setup	Add	Connected	Release	ModifyConn	Modify	ToneActive	CotActive	Override	Pending	Seized	Move	Prepare	Broken
AddReply	A	A	29, 30, 31, 32, 33, 256	A	A	A	A	A	A	A	157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167	A	221	A	A
SubtractReply	A	A	A	A	90, 91	A	A	A	A	139	168, 169, 170, 171	A	A	A	A
ModifyReply	A	A	A	A	A	105, 106	114, 115, 116, 117, 118, 119	A	A	A	172, 173, 174, 175, 176, 177, 178, 179	A	A	A	A
MoveReply	A	A	A	A	A	A	A	A	A	140, 141	180	A	222, 223, 224, 225, 226	A	A
Notify - establish	x	14	34, 35, 36	59, 60	92, 93	A	A	A	A	A	181	215	227, 228	A	260
Notify - release	x	15	15	61, 62, 63, 64, 65, 66	94, 95	15	15	A	A	142	182, 183, 184, 185, 186	216	15	15	242, 243

ML State-Event Matrix (3/4)

State / Triggers	Null	Setup	Add	Connected	Release	ModifyConn	Modify	ToneActive	CotActive	Override	Pending	Seized	Move	Prepare	Broken
hc_msg - setup	8	A	A	67, 68, 69, 70, 71	96	A	A	127	132	A	187	217, 218	229, 230	A	244, 245, 246, 247, 248, 249, 250, 261, 262
hc_msg - setup_res	A	16, 17, 18, 19, 20, 21, 22, 255	A	A	A	A	A	A	A	A	188, 189, 190, 191, 192, 193, 194, 195	A	A	A	A
hc_msg - modify	A	A	A	72	A	A	A	A	A	A	196	A	A	A	A
hc_msg - modify_res	A	A	A	A	A	107, 108	120	A	A	A	A	A	A	A	A
hc_msg - release	9, 10	23	23	73, 74, 75, 76, 77	97, 98	23	23	A	A	143	197, 198, 199, 200, 201, 202, 203	A	23	23	251
hc_msg - release_res	x	A	A	A	99, 100	NA	NA	NA	NA	144	A	A	A	A	A
hc_msg - prepare	11	A	A	257	101	A	A	A	A	A	204, 205, 206, 207, 208	A	A	A	A
hc_msg - prepare_res	A	A	A	A	A	A	A	A	A	A	A	A	A	232, 233, 258	A
hc_msg - break	A	A	A	78	A	A	A	A	A	A	A	A	A	A	A
hc_msg - break_res	A	A	A	79	A	A	A	A	A	A	A	A	A	A	A

ML State-Event Matrix (4/4)

State / Triggers	Null	Setup	Add	Connected	Release	ModifyConn	Modify	ToneActive	CutActive	Override	Pending	Seized	Move	Prepare	Broken
hc_timeout	NA	24, 25, 26, 27	NA	NA	102	109	121	NA	NA	145	209	NA	NA	234	NA
gcp_timeout	A	A	37, 38, 39	A	103	110	122	A	A	146, 147	210	A	231	A	A
abnormal_rel	x	28	28	80, 81, 82, 83	104	28	28	128	133	148, 149	211, 212	219	28	28	252, 253

Observations...

Observations re. ML

- Still, only the external protocol is handled this way (the state machine uses synchronous calls towards internal APIs) – otherwise, it would *really* be bad.
- This is the semantics offered by UML(*) as well (UML gives lots of abstraction support, but only for the sequential parts – not for state machines.)
- This seems analogous to
  - Dijkstra's "Goto considered harmful", and
  - local vs. global variables.

(*) Only partly true – see 'deferrable event', UML 1.5, part 2, pp 147

Questions?

wxErlang

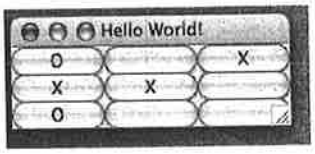
Mats-Ola Persson

wxErlang

- GUI library for Erlang
  - write GUI applications
  - cross platform
  - cross platform look and feel

Example

- A "stupid" *tic-tac-toe* application
- No intelligent behavior is implemented
- Just the (good) looks



Creating a window

```
create_window() ->
        wx:start(), % initialize wxErlang

        % create a frame (window) with no parent
        Frame = wx:frame(?NULL, ?wxID_ANY, "Hello World!"),

        wx:show(Frame). % make the frame visible
```

Creating a window

- Not very exciting: an empty window

Event-driven programming

- Things "react" to mouse clicks, mouse movements, etc.
- Event-handlers, callback functions

Events

```
create_window() ->
        wx:start(),
        Frame = wx:frame(?NULL, ?wxID_ANY, "Hello World!"),

        % event-handler that reacts to close clicks
        wx:connect(Frame, ?wxEVT_CLOSE_WINDOW,
                                fun(_,_) -> wx:quit() end),

        wx:show(Frame).
```

Layouts

- Platform independent layouts
  - Sizers
    - No fixed sized widgets, etc.
    - Arbitrary complex layouts
- Let's add the buttons!

Sizers

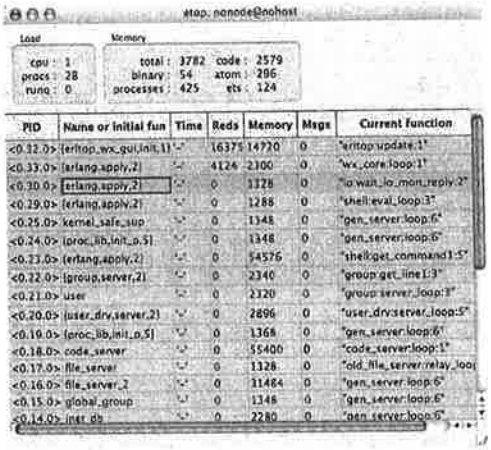
```
create_grid(Frame) ->    Grid = wx:grid_sizer(3), % a grid sizer with 3 cols
create_buttons(Grid, Frame, 9), % create 9 buttons    Grid.
create_buttons(_, _, 0) ->    ok;create_buttons(Grid, Frame, N) ->    Button =
wx:button(Frame, N), %a button without label

   wx:add(Grid, Button), % add button to the sizer    create_buttons(Grid, Frame,
N - 1).
```

The result

- A "shell" of a tic-tac-toe application

wxEtop

- Port of the old "etop"
- New features
  - context menues
  - view running code
  - ...



From the programmers point of view

- wxErlang is verbose - like most GUI libraries
- Trial-and-error - like most GUI libraries
- Interface designers
  - XRCed, DialogBlocks, etc.

Design decisions

- Binding to the C ++ GUI library wxWidgets
- Get a lot for free
  - "free" features from wxWidgets
  - reduced maintenance work
- wxErlang interface resembles wxWidgets C++ interface
  - free documentation(!)

Translation scheme

- Easy
  - Functions
    - Multiple return values => tuples
    - ...
  - Constants

Translation scheme

- Not as easy
  - Classes
  - Overloading and overriding functions
  - Type system

Safety

- Checks arguments
- Types
  - Primitive values
  - Objects
- Sanity

Implementational details

- Most of the code is generated from wxWidgets headers
- Implemented as a "port program"
- Has a lot of bugs

Wrap up

- Current status - a prototyp
  - wxEtop
  - Perhaps 10% implemented
- Future

Yet Another GUI for Erlang

Like a Fish needs a Bicycle.

Why did I do it?

- GUI tools for testing and troubleshooting
- Wanted to write some C

Why Yet Another?

- GS/Tcl is
  - slow
  - ugly
  - bizarre
  - no GUI builder
- ErlGTK
  - no longer supported

Why GTK2?

- Elegant design
- Open Source de facto standard (with Qt)
- Runs on Unices and Windows, but not MacOS
- Looks good
- Excellent GUI builder (Glade)
- Strangely familiar...
  - Garbage collection
  - Runtime type checking
  - Introspection

gtkNode

- C-node
- Each widget appears as a registered Erlang process
- Behaviour is specified by the config file from Glade
- Communicates with the Erlang node through messages

Glade

- Defines and names widgets
- Specifies properties of widgets
- Layout
- Events
  - Ignore
  - Handle by GTK callback
  - Send Erlang message

OO

- GTK is object oriented, but implemented in C
- Each method is a C function call

```
gtk_class_method(widget,args...);
```

- From Erlang, it looks like this (conceptually);

```
Widget ! {class_method, [Args]},
```

- It is implemented thusly;

```
GtkNode ! {class_method, [Widget|Args]},
```

Type safety

When gtkNode receives a message it checks;

- Widget exists
- Method exists
- Widget is of correct class for method
- Right number of args
- Right type of args

If one of these fails, gtkNode sends an error tuple.

Otherwise it calls the method and sends the result to the Erlang node

Typical

GTK's basic types are;	Mapped to Erlang;
• floats	• float
• integers	• integer
• strings	• list
• widgets (boxed pointer)	• atom
• structs (boxed pointer)	• atom
• booleans	• true/false
• macros	• atoms

Code generator

- The code generator decides which GTK functions can be safely called from Erlang (about half)
- For each safe function it generates C wrapper code that does the type checking
- The code generator works by analyzing the C header files.
- Written in Erlang and Python (stolen from PyGtk)

Odds and Ends

- messages can be stacked for efficiency
- recommended usage is through gtkNode.erl

available from jungerl:

http://jungerl.sourceforge.net

UPPSALA UNIVERSITET

Bit-level Binaries and Generalized Comprehensions in Erlang

Per Gustafsson and Kostis Sagonas
Dept of Information Technology
Uppsala University

UPPSALA UNIVERSITET

Binaries as we know them

Introduced in 1992 as a container for object code

Used in applications that do I/O, networking or protocol programming

A proposal for a binary datatype and a syntax was made in 1999 and a revised version was adopted in 2000

Since then, binaries have been used extensively, often providing innovative solutions to common telecom programming tasks

UPPSALA UNIVERSITET

Binaries are not so flexible

Some limitations:

- Binaries are byte streams, not bit streams
- Segment sizes cannot be arbitrary arithmetic expressions

Both undermine the use of the binary syntax for writing high level specifications

This work:

We show how to lift these limitations while maintaining backward compatibility

UPPSALA UNIVERSITET

Make binaries as flexible as lists

- In lists:
  - deconstructing a list always yields valid terms
  - can be constructed using list comprehensions
- In binaries:
  - deconstructing a binary sometimes yields terms which cannot be represented as Erlang binaries
  - no binary comprehensions are available

- **This work:**
  - allows binaries to represent bit streams
  - introduces binary comprehensions
  - introduces extended comprehensions to make conversions between lists and binaries simpler

UPPSALA UNIVERSITET

Flexible bit-level binaries

- The multiple-of-eight size restriction is lifted
- The size field of a segment can contain an arbitrary arithmetic expression
- No type specifier is needed in binary construction

UPPSALA UNIVERSITET

Pros and cons of bit-level binaries

+ Allows natural representation of bit fields
  - `<<BitSize:8, BitField:BitSize/binary, ...`
+ Helps avoid padding calculations
  - `Pad = (8 - ((X + Y) rem 8)) rem 8,`
+ Makes binary matching as easy for bit streams as it was for byte streams

– Introduces a speed trade-off

UPPSALA UNIVERSITET

Pattern Matching
- byte streams vs bit streams

```
keep_0XX(<<0:8,X:16,Rest/binary>>) ->
    <<0:8,X:16,keep_0XX(Rest)/binary>>;
keep_0XX(<<_:24,Rest/binary>>) ->
    keep_0XX(Rest);
keep_0XX(<<>>) ->
    <<>>.
```

This function only keeps the <u>byte</u> triples whose first byte is 0.

But what if me want to keep the <u>bit</u> triples whose first bit is 0?

UPPSALA UNIVERSITET

Pattern Matching
- byte streams vs bit streams

```
keep_0XX(<<0:1,X:2,Rest/binary>>) ->
    <<0:1,X:2,keep_0XX(Rest)/binary>>;
keep_0XX(<<_:3,Rest/binary>>) ->
    keep_0XX(Rest);
keep_0XX(<<>>) ->
    <<>>.
```

This is how it ought to look!

UPPSALA UNIVERSITET

Pattern Matching
- byte streams vs bit streams

```
keep_0XX(Bin) -> keep_0XX(Bin, 0, 0, <<>>).

keep_0XX(Bin, N1, N2, Acc) ->
  Pad1 = (8 - ((N1+3) rem 8)) rem 8,
  Pad2 = (8 - ((N2+3) rem 8)) rem 8,
  case Bin of
    <<_:N1, 0:1, X:2, _:Pad1, _/binary>> ->
      NewAcc =
      <<Acc:N2/binary-unit:1, 0:1, X:2, 0:Pad2>>,
      keep_0XX(Bin, N1+3, N2+3, NewAcc);
    <<_:N1, _:3, _:Pad1, _/binary>> ->
      keep_0XX(Bin, N1+3, N2, Acc);
    <<_:N1>> -> Acc
  end.
```

This is how you have to write it today!

UPPSALA UNIVERSITET

Allowing arithmetic expressions in the size field

Consider this classic example of the bit syntax:

```
case IP_Packet of
  <<4:4, Hlen:4, SrvcType:8, TotLen:16,
    ID:16, Flgs:3, FragOff:13, TTL:8, Proto:8,
    SrcIP:32, DestIP:32,
    RestDgrm/binary>> ->
      OptsLen = Hlen - 5,
      <<Opts:OptsLen/binary-unit:32,
        Data/binary>> = RestDgrm,
      ...
end
```

UPPSALA UNIVERSITET

Allowing arithmetic expressions in the size field

Using flexible binaries it could be written in the following manner:

```
case IP_Packet of
  <<4:4, Hlen:4, SrvcType:8, TotLen:16,
    ID:16, Flgs:3, FragOff:13, TTL:8,
    Proto:8, SrcIP:32, DestIP:32,
    Opts:((Hlen - 5)*32)/binary,
    Data/binary>> ->  ...
end,
```

UPPSALA UNIVERSITET

No need for a type-specifier in binary construction

Consider the following code:

```
X = <<1, 2, 3>>,
B = <<X, 4, 5>>
```

It causes a runtime exception. To avoid this you must explicitly specify the type

```
X = <<1, 2, 3>>,
B = <<X/binary, 4, 5>>
```

We want to lift this restriction, the type should default to the type of the variable.

UPPSALA UNIVERSITET

Binary Comprehensions

Analogous to List Comprehensions

- List Comprehensions represent a combination of map and filter

Comprehensions require a notion of an element

- For binary comprehensions the user must specify what they consider as an element

UPPSALA UNIVERSITET

Binary Comprehensions:

Introductory Example, invert

Using list comprehension:

```
invert(ListOfBits) ->
  [bnot(X) || X <- ListOfBits]
```

Using binary comprehension:

```
invert(Binary) ->
  <<bnot(X):1 || X:1 <- Binary>>
```

If your binary is byte-sized:

```
invert(Binary) ->
  <<bnot(X):8 || X:8 <- Binary>>
```

UPPSALA UNIVERSITET

Binary Comprehensions:

UU-decode

Using a binary comprehension UU-decode basically becomes a one-liner in Erlang

```
uudecode(UUBin) ->
  <<(X-32):6 || X:8 <- UUBin, 32=<X, X=<95 >>
```

Note the filter expressions which make sure that inserted characters such as line-breaks are dropped

UPPSALA UNIVERSITET

Extended comprehensions

Can we use list generators in binary comprehensions?

```
convert_to_binary(ListofWords) ->
  <<X:32 || X <- ListofWords>>.
```

YES !

UPPSALA UNIVERSITET

Extended comprehensions

Can we use binary generators in list comprehensions?

```
convert_to_listofwords(Binary) ->
  [X || X:32 <- Binary].
```

YES !

UPPSALA UNIVERSITET

Generators

Note that we need to be able to separate list generators from binary generators.

List generators:

```
P <- E_L
```

Binary generators:

```
S_1 ... S_n <= E_B
```

P	–	a pattern
E_L	–	an Erlang expression which evaluates to a list
S_i	–	a binary segment
E_L	–	an Erlang expression which evaluates to a binary

UPPSALA UNIVERSITET

Implementation of extended binary comprehensions

- We present a simple translation of extended comprehensions into Erlang in the form of rewrite rules in the paper

- Using these simple rules the cost of building the resulting binary is quadratic in the number of segments

- We present another set of rewrite rules which gives linear complexity, but the rules are slightly less straight-forward

UPPSALA UNIVERSITET

Implementation of extended binary comprehensions

When the size of the resulting binary can be calculated as a function of a generator binary, the translation can be very efficient

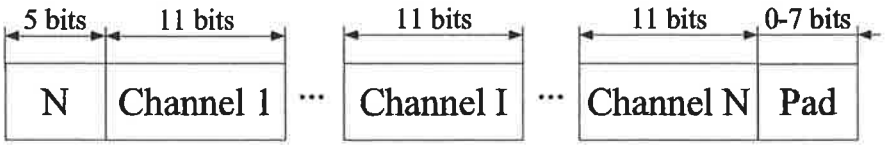
```
Res = << X:16 || X:8 <= Bin>>.
                =>
bitsize(Res)  == (bitsize(Bin) / 8) * 16
```

This allows us to preallocate the memory that is needed for the resulting binary

UPPSALA UNIVERSITET

Example: IS-683 PRL

Data Structure



Task:
Create a list of Channels

UPPSALA UNIVERSITET

First "Padding" Solution:

```
decode(<<NumChans:5, _Pad:3, _Rest/binary>> = Bin) ->
   decode(Bin, NumChans, NumChans, []).

decode(_, _, 0, Acc) ->
   Acc;
decode(Bin, NumChans, N, Acc) ->
   SkipBefore = (N - 1) * 11,
   SkipAfter = (NumChans - N) * 11,
   Pad = 8 - ((NumChans * 11 + 5) rem 8),
   <<_:5, _:SkipBefore, Chan:11,
     _:SkipAfter,_:Pad>> = Bin,
   decode(Bin, NumChans, N - 1, [Chan | Acc]).
```

Buggy calculation of padding

UPPSALA UNIVERSITET

Correct "Padding" Solution:

```
decode(<<NumChans:5, _Pad:3, _Rest/binary>> = Bin) ->
    decode(Bin, NumChans, NumChans, []).

decode(_, _, 0, Acc) ->
    Acc;
decode(Bin, NumChans, N, Acc) ->
    SkipBefore = (N - 1) * 11,
    SkipAfter = (NumChans - N) * 11,
    Pad = (8 - ((NumChans * 11 + 5) rem 8)) rem 8,
    <<_:5, _:SkipBefore, Chan:11,
      _:SkipAfter,_:Pad>> = Bin,
    decode(Bin, NumChans, N - 1, [Chan | Acc]).
```

UPPSALA UNIVERSITET

Expanded solution:

```
decode(Channels) ->
  case Channels of
    <<0:5, :5>> ->
      []:
    <<1:5,X1:11,_:6>> ->
      [X1];
    <<2:5,X1:11,X2:11,_:5>> ->
      [X1,X2];
    <<3:5,X1:11,X2:11,X3:11,_:2>> ->
      [X1,X2,X3];
    <<4:5,X1:11,X2:11,X3:11,X4:11, :5>> ->
      [X1,X2,X3,X4];
    <<5:5,X1:11,X2:11,X3:11,X4:11,X5:11,_:4>> ->
      [X1,X2,X3,X4,X5];
    <<6:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,_:1>> ->
      [X1,X2,X3,X4,X5,X6];
    <<7:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11, :6>> ->
      [X1,X2,X3,X4,X5,X6,X7];
    <<8:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,_:7>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8];
    <<9:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,_:8>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9];
    <<10:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,_:9>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10];
    <<11:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11, :2>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11];
    <<12:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,_:7>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12];
    <<13:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,_:6>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13];
    <<14:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,_:5>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14];
    <<15:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11, :4>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15];
    <<16:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,_:3>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16];
    <<17:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,_:2>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17];
    <<18:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,_:1>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18];
    <<19:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,_:0>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19];
    <<20:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,_:7>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20];
    <<21:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11, :6>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21];
    <<22:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,_:5>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22];
    <<23:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,_:4>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23];
    <<24:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,_:3>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24];
    <<25:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,X25:11, :6>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24,X25];
    <<26:5,X1:11,X2:11,X7:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,X25:11,X26:11,_:5>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24,X25,X26];
    <<27:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,X25:11,X26:11,X27:11,_:4>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24,X25,X26,X27];
    <<28:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,X25:11,X26:11,X27:11,X28:11,_:3>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24,X25,X26,X27,X28];
    <<29:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,X25:11,X26:11,X27:11,X28:11,X29:11,_:2>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24,X25,X26,X27,X28,X29];
    <<30:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,X25:11,X26:11,X27:11,X28:11,X29:11,X30:11,_:1>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24,X25,X26,X27,X28,X29,X30];
    <<31:5,X1:11,X2:11,X3:11,X4:11,X5:11,X6:11,X7:11,X8:11,X9:11,X10:11,X11:11,X12:11,X13:11,X14:11,X15:11,X16:11,X17:11,X18:11,X19:11,X20:11,X21:11,X22:11,X23:11,X24:11,X25:11,X26:11,X27:11,X28:11,X29:11,X30:11,X31:11,_:0>> ->
      [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24,X25,X26,X27,X28,X29,X30,X31]
end.
```

<<3:5,X1:11,X2:11,X3:11,_:2>> ->
[X1,X2,X3];

UPPSALA UNIVERSITET

Smart, but inefficient solution

```
decode(<<N_channels:5, Alignment_bits:3, Tail/binary>>) ->
  decode2(N_channels, <<Alignment_bits:3, Tail/binary, 0:5>>).

decode2(0, _) ->
  [];
decode2(N, <<C:11, A:5, T/binary>>) ->
  [C|decode2(N-1, <<A:5, T/binary, 0:3>>)].
```

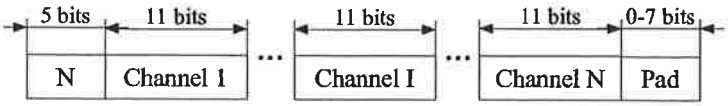
Avoids complicated padding calculations,
at the cost of recreating the binary in each iteration.

UPPSALA UNIVERSITET

Using Flexible binaries

Since flexible binaries can represent bit streams properly and leads to a natural solution

```
decode(<<N:5, Channels:(11*N)/binary,_/binary>>) ->
  decode2(Channels).

decode2(<<C:11, T/binary>>) ->
  [C|decode2(T)];
decode2(<<>>) ->
  [].
```

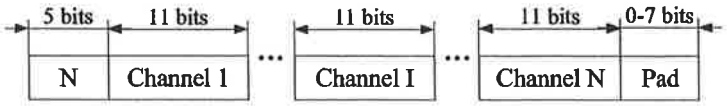


UPPSALA UNIVERSITET

Extended comprehensions

Using extended comprehensions and flexible binaries we can solve the problem in two lines:

```
decode(<<N:5, Channels:(11*N)/binary,_/binary>>) ->
  [X || X:11 <= Channels].
```



UPPSALA UNIVERSITET

Succintness of flexible binaries

- as measured in line counts

Program in	C	Java	Erlang (R10B)	Erlang (this)
keep 0XX	51	33	14	2
μ-law encode	30	25	25	13
UU-decode	19	14	10	2

μ-law encode	- Compresses sound
keep 0XX	- Keeps bit-triples that start with 0
UU-decode	- Decodes UU-encoded binaries

UPPSALA UNIVERSITET

Conclusion

- Introducing bit-level binaries makes it easy to represent bit streams as binaries
- This makes it possible to write high level specifications of operations on bit streams
- Extended comprehensions allow for powerful manipulation of binaries
- Together these extensions make binaries as easy to use as other datatypes in Erlang such as tuples and lists
- The extensions we propose are backwards compatible
- They will probably be included in the R11 release of Erlang/OTP

UPPSALA UNIVERSITET

Future Work

- A standard library for dealing with binaries
- A better representation of binaries to avoid quadratic complexity when appending binaries
- New compilation techniques which allow for in-place updates of binaries

UPPSALA UNIVERSITET

Adapting BIF:s to bit-level binaries

`size(Bin)`

- should return the minimal number of bytes needed to represent the binary.

`bitsize(Bin)`

- new bif which returns the size in bits

`binary_to_list(Bin)`

> the following should hold:

```
Bin == list_to_binary(binary_to_list(Bin))
```

UPPSALA UNIVERSITET

binary_to_list(Bin)

Desired property:

```
Bin == list_to_binary(binary_to_list(Bin))
```

```
binary_to_list(<<X:8,Rest/binary>>) ->
  [X|binary_to_list(Rest)];
binary_to_list(<<>>) ->
  [];
binary_to_list(Bin) when is_binary(Bin) ->
  [Bin].
```

gives:

```
[0,0,<<0:4>>] == binary_to_list(<<0:20>>)
```

ERLANG

OTP Development update

ERICSSON
TAKING YOU FORWARD

ERLANG

Highlights during 2005

- Released R10B-2 .. R10B-8, will be a R10B-9 before end of year.

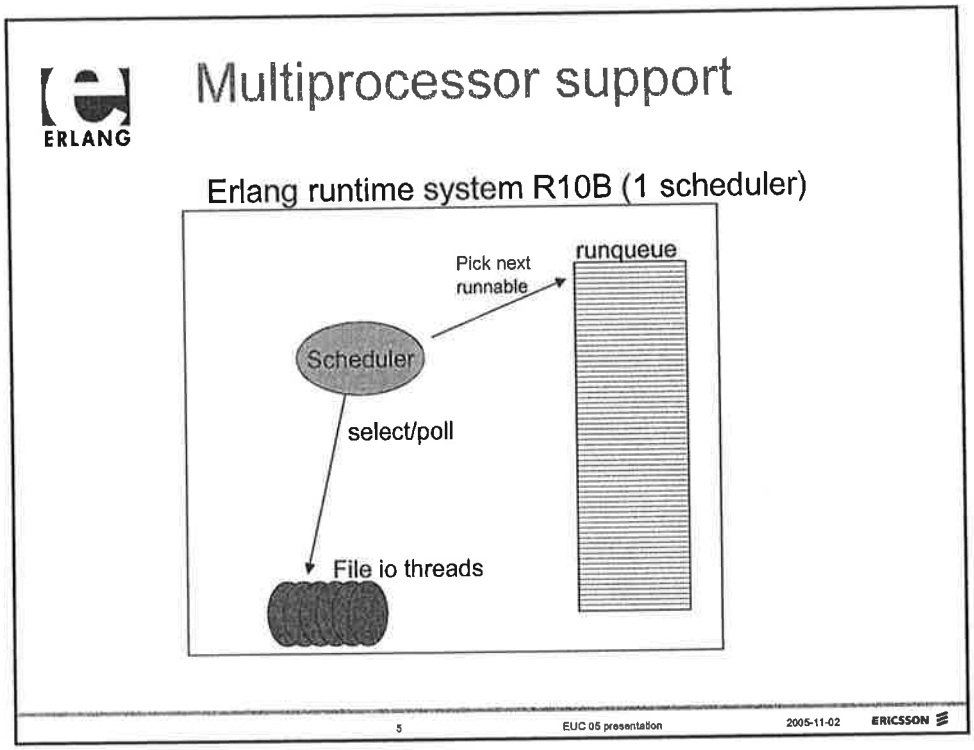
News in R10B-8

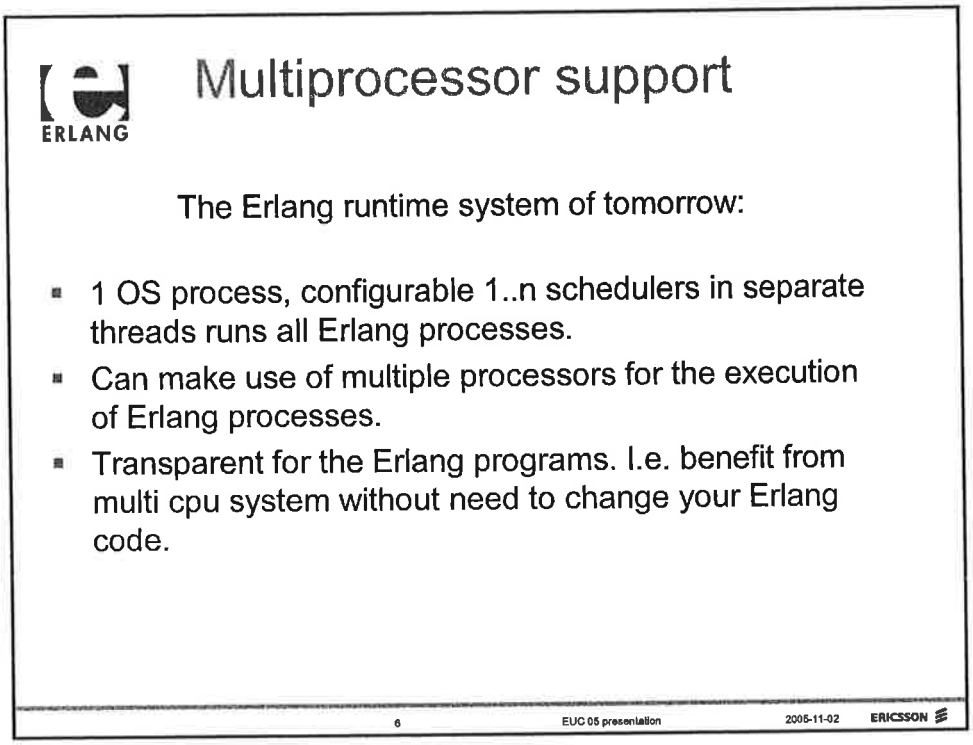
- Improvements of global
- new application SSH (both server and client) beta status, nice way to implement CLI for an application.
- Debugger now with support for try catch
- New version of Edoc (thanks to Richard Carlsson)

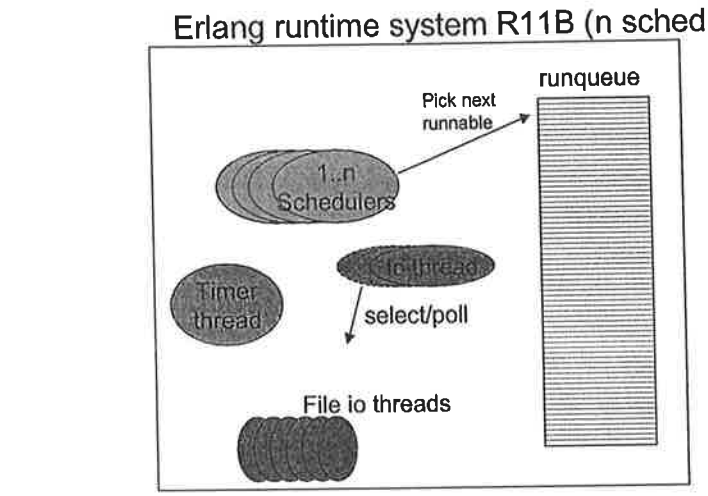
Multiprocessor support

The Erlang runtime system of today (R10B)

- 1 OS process, 1 thread runs all Erlang processes.
- Can not make use of more than 1 processor for the execution of Erlang processes.


ERLANG
Multiprocessor support
Erlang runtime system R10B (1 scheduler)
Pick next runnable
runqueue
Scheduler
select/poll
File io threads
5
EUC 05 presentation
2005-11-02
ERICSSON


ERLANG
Multiprocessor support
The Erlang runtime system of tomorrow:
1 OS process, configurable 1..n schedulers in separate threads runs all Erlang processes.
Can make use of multiple processors for the execution of Erlang processes.
Transparent for the Erlang programs. I.e. benefit from multi cpu system without need to change your Erlang code.
6
EUC 05 presentation
2005-11-02
ERICSSON



Multiprocessor support

some additional characteristics

- Linux, Solaris (MacOS x) first, (posix threads)
- No changes in compiler
- Support "traditional" separate heap per E-process first
- Hybrid heap later.

Multiprocessor support

Current status and benchmarks

- Can run quite a lot of the regular test suites
- A small benchmark ran:dom([1500,15000]) (2 processes sorting lists) on a 2-cpu machine
- maskin) give the following result:

	Time	Relation
Pre 11B without smp support	750 ms	base for comparision
Pre 11B with smp and 1 scheduler	845 ms	13% slower
Pre 11B with smp and 2 schedulers	520 ms	31% faster

Multiprocessor support
way forward

First step (ongoing)

- Add locking wherever needed
- Make system stable (Linux, Solaris) with multiple schedulers on multiple and single cpu systems.
- Include in R11B (as beta status)

Next step

- Benchmarking, profiling and optimizations
- Other platforms (MacOSx, Windows)
- Include in update release for R11B (end of 2006)

Multiprocessor support
joint development

The SMP support for Erlang is a joint development effort between the Ericsson OTP team, Uppsala University and Synapse

Many thanks to

- Mikael Pettersson, Uppsala University
- Tony Rogvall, Synapse

TAKING YOU FORWARD

Erlang/OTP User Conference 2005

Speakers

Simon	Aurell	Erlang Training and Consulting	London	England	simon@erlang-consulting.com
Gábor	Bátori	Ericsson	Budapest	Hungary	Gabor.Batori@ericsson.com
Mats	Cronqvist	Ericsson	Stockholm	Sweden	mats.cronqvist@ericsson.com
Vlad	Dumitrescu	HiQ	Göteborg	Sweden	vlad.xx.dumitrescu@ericsson.com
Eduardo	Figoli	IN Switch Solutions Inc	Miami	USA	
Victor M.	Gulias	University of A Coruña	Coruña	Spain	gulias@dc.fi.udc.es
Per	Gustafsson	University of Uppsala	Uppsala	Sweden	pergu@it.uu.se
Kenneth	Lundin	Ericsson	Stockholm	Sweden	kenneth.lundin@ericsson.com
Chandru	Mullaparthi	T-Mobile	London	England	chandrashekhar.mullaparthi@t-mobile.co.uk
Bernardo	Paroli	IN Switch Solutions Inc	Miami	USA	
Mats-Ola	Persson	Chalmers univ of Technology	Göteborg	Sweden	md1matso@mdstud.chalmers.se
Mickaël	Rémond	Process-one	Paris	France	mickael.remond@erlang-fr.org
Kostis	Sagonas	University of Uppsala	Uppsala	Sweden	kostis@user.it.uu.se
Corrado	Santoro	University of Catania	Catania	Italy	csanto@diit.unict.it
Carlos E.	Silva	IN Switch Solutions Inc	Miami	USA	carlos@inswitch.us
Michal	Slaski	Erlang Training and Consulting	London	England	michal.slaski@gmail.com
Zoltan	Theisz	Ericsson	Budapest	Hungary	
Manfred	Widera	FernUniversität Hagen	Hagen	Germany	Manfred.Widera@fernuni-hagen.de
Ulf	Wiger	Ericsson	Stockholm	Sweden	ulf.wiger@ericsson.com

Participants

Ola	Andersson	Ericsson	Stockholm	Sweden	ola.a.andersson@ericsson.com
Peter	Andersson	Ericsson	Stockholm	Sweden	
Gunilla	Arendt	Ericsson	Stockholm	Sweden	
Joe	Armstrong	Ericsson	Stockholm	Sweden	joe.armstrong@ericsson.com
Thomas	Arts	IT-university of Göteborg	Göteborg	Sweden	thomas.arts@ituniv.se
Johan	Bevemyr	Tail-F	Stockholm	Sweden	jb@tail-f.com
M. Harris	Bhatti		Linlithgow	Scotland	harrisbhatti@gmail.com
Éva	Bihari	Ericsson	Budapest	Hungary	eva.bihari@ericsson.com
Martin	Björklund	Tail-F	Stockholm	Sweden	mbj@tail-f.com
Johan	Blom	Mobile Arts	Stockholm	Sweden	johan.blom@mobilearts.com
Hans	Bolinder	Ericsson	Stockholm	Sweden	
Urban	Boquist	Ericsson	Göteborg	Sweden	urban.boquist@ericsson.com
Pascal	Brisset	Cellicium	Paris	France	pascal.brisset@cellicium.com

Participants cont.					
Mikael	Bylund	Telia Sonera	Uppsala	Sweden	
Göran	Båge	Mobile Arts	Stockholm	Sweden	goran.bage@mobilearts.com
Martin	Carlson	Erlang Training and Consulting	London	England	
Richard	Carlsson	University of Uppsala	Uppsala	Sweden	richardc@comhem.se
Jakob	Cederlund	Ericsson	Stockholm	Sweden	jakob@erix.ericsson.se
Francesco	Cesarini	Erlang Training and Consulting	London	England	francesco@erlang-consulting.com
Bjarne	Däcker		Segeltorp	Sweden	bjarne@cs-lab.org
Niclas	Eklund	Ericsson	Stockholm	Sweden	nick@erix.ericsson.se
Thomas	Elsgaard	Ericsson	Copenhagen	Denmark	thomas.elsgaard@ericsson.com
Morgan	Eriksson	Nortel	Stockholm	Sweden	
Michael	Fogeborg	Telenor	Oslo	Norway	micke@online.no
Magnus	Fröberg	Nortel	Stockholm	Sweden	magnus@bluetail.com
Francesca	Gangemi	Erlang Training and Consulting	London	England	
Joakim	Grebenö	Tail-F	Stockholm	Sweden	jocke@tail-f.com
Rickard	Green	Ericsson	Stockholm	Sweden	
Scott	Green	T-Mobile	London	England	Scott.Green@t-mobile.co.uk
Dag	Gruneau	Nortel	Stockholm	Sweden	dag@bluetail.com
Dan	Gudmundsson	Ericsson	Stockholm	Sweden	
Björn	Gustavsson	Ericsson	Stockholm	Sweden	bjorn@erix.ericsson.se
Gordon	Guthrie		Edinburgh	Scotland	gordon_guthrie@hotmail.com
Niklas	Hanberger	Nortel	Stockholm	Sweden	
Siri	Hansen	Ericsson	Stockholm	Sweden	
Dale	Harvey	Herriot Watt University	Edinburgh	Scotland	harveyd@gmail.com
Dragan	Havelka	Mobile Arts	Stockholm	Sweden	dragan.havelka@mobilearts.com
Per	Hedeland	Nortel	Stockholm	Sweden	per@bluetail.com
Pekka	Hedqvist	Optimobile AB	Stockholm	Sweden	pekkahedqvist@yahoo.se
Sean	Hinde	Synap.se	London	England	sean.hinde@gmail.com
John	Hughes	Chalmers univ of Technology	Göteborg	Sweden	john.hughes@swipnet.se
Håkan	Huss	Ericsson	Stockholm	Sweden	huss01@gmail.com
Rikard	Johansson	Mobile Arts	Stockholm	Sweden	rikard.johansson@mobilearts.com
Thomas	Johnsson	Gatespacetelematics	Göteborg	Sweden	thomas@gatespacetelematics.com
	Kannan	Nortel	Stockholm	Sweden	
Bertil	Karlsson	Ericsson	Stockholm	Sweden	
Mikael	Karlsson	Creado Systems	Stockholm	Sweden	mikael.karlsson@creado.com

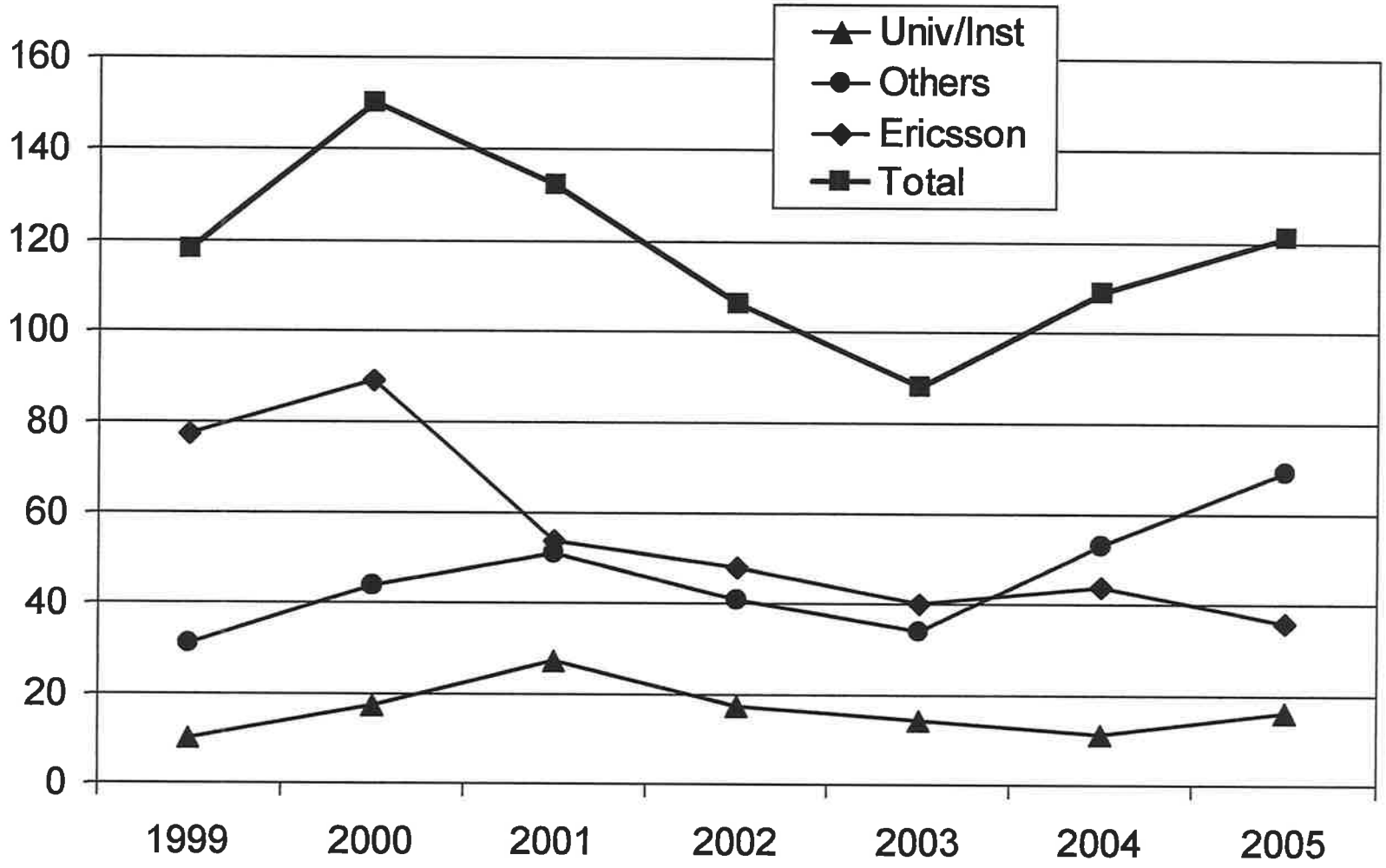
Participants cont.					
Peter	Karlsson	Ericsson	Stockholm	Sweden	
Martin	Kjellin	Mobile Arts	Stockholm	Sweden	martin.kjellin@mobilearts.com
Bengt	Kleberg	Ericsson	Stockholm	Sweden	bengt.kleberg@ericsson.com
	Krishna	Nortel	Stockholm	Sweden	
Lukas	Larsson	Erlang Training and Consulting	London	England	
Petter	Larsson	Ericsson	Linköping	Sweden	
Tord	Larsson	Nortel	Stockholm	Sweden	tlarsson@nortel.com
Conrad	Levitt	Herriot Watt University	Dunblane	Scotland	benefitsdragon@gmail.com
Tobias	Lindahl	University of Uppsala	Uppsala	Sweden	Tobias.Lindahl@it.uu.se
Thomas	Lindgren	Millpond Services Ltd	London	England	thomasl_erlang@yahoo.com
Daniel	Luna	Erlang Training and Consulting	London	England	luna@Update.UU.SE
Peter	Lund	Synap.se	Stockholm	Sweden	peter.lund@lundata.se
Matthias	Läng	Corelatus	Stockholm	Sweden	matthias@corelatus.se
Ann-Marie	Löf	Sjöland & Thyselius	Stockholm	Sweden	Ann-Marie.Lof@st.se
Peter-Henry	Mander		Thame	England	erlang@manderhanyu.plus.com
Thomas	Mattisson	Mobile Arts	Stockholm	Sweden	thomas.mattisson@mobilearts.com
Håkan	Mattsson	Ericsson	Stockholm	Sweden	hakan@erix.ericsson.se
Håkan	Millroth	Tail-F	Stockholm	Sweden	hakan@tail-f.com
Peter	Nagy	Ericsson	Budapest	Hungary	peter.nagy@ericsson.com
Vincenzo	Nicosia	University of Catania	Catania	Italy	vnicosia@diit.unict.it
Patrik	Nilsson	Nortel	Stockholm	Sweden	
Raimo	Niskanen	Ericsson	Stockholm	Sweden	raimo@erix.ericsson.se
Patrik	Nyblom	Ericsson	Stockholm	Sweden	
Jan Henry	Nyström	Herriot Watt University	Edinburgh	Scotland	jann@macs.hw.ac.uk
Göran	Oettinger	Mobile Arts	Stockholm	Sweden	goran.oettinger@mobilearts.com
Lars	Petterson-Fink	Nortel	Stockholm	Sweden	
Mikael	Pettersson	University of Uppsala	Uppsala	Sweden	mikpe@csd.uu.se
Laurent	Picouleau	Erlang Training and Consulting	London	England	
Anders	Ramsell	Telia Sonera	Uppsala	Sweden	Anders.Ramsell@teliasonera.com
Tony	Rogvall	Synap.se	Stockholm	Sweden	tony@rogvall.com
Staffan	Skogvik	Ericsson	Linköping	Sweden	staffan.xq.skogvik@ericsson.com
Håkan	Stenholm	Tail-F	Stockholm	Sweden	hokan@kreditor.se
Erik	Stenman	University of Uppsala	Uppsala	Sweden	happi.stenman@gmail.com
Sebastian	Strollo	Synap.se	Stockholm	Sweden	seb@strollo.org

Participants cont.

Per Einar	Strömme		Stockholm	Sweden	stromme@telia.com
Ulf	Svarte Bagge	Corelatus	Stockholm	Sweden	ulf@corelatus.se
Gunnar	Sverredal	Telia Sonera	Uppsala	Sweden	
Taavi	Talvik	Elisa	Tallinn	Estonia	taavi.talvik@elisa.ee
Marcus	Taylor	Erlang Training and Consulting	London	England	
Lars	Thorsén	Ericsson	Stockholm	Sweden	lars@erix.ericsson.se
Magnus	Thoäng	Ericsson	Stockholm	Sweden	
Fredrik	Thulin	University of Stockholm	Stockholm	Sweden	ft@it.su.se
Zoltan Peter	Toth	Ericsson	Budapest	Hungary	zoltan.peter.toth@ericsson.com
Torbjörn	Törnkvist	Nortel	Stockholm	Sweden	tobbe@nortel.com
Jane	Walerud	Tail-F	Stockholm	Sweden	jane@walerud.com
Gillan	Ward	Erlang Group	London	England	http://www.erlanggroup.com/
Carlos	Varela Paz	University of A Coruña	Coruña	Spain	cvarela@dc.fi.udc.es
Esko	Vierumäki	Ericsson	Stockholm	Sweden	esko.vierumaki@ericsson.com
Claes	Wikström	Tail-F	Stockholm	Sweden	klacke@tail-f.com
Chris	Williams	Ericsson	Stockholm	Sweden	chris.williams@ericsson.com
Mike	Williams	Ericsson	Stockholm	Sweden	michael.williams@ericsson.com
Wen	Xu	Royal Institute of Technology	Stockholm	Sweden	wenx@kth.se
Erik	Åckander	Ericsson	Stockholm	Sweden	
Lennart	Öhman	Sjöland & Thyselius	Stockholm	Sweden	lennart.ohman@st.se
Göran	Östlund		Stockholm	Sweden	goran.may@chello.se

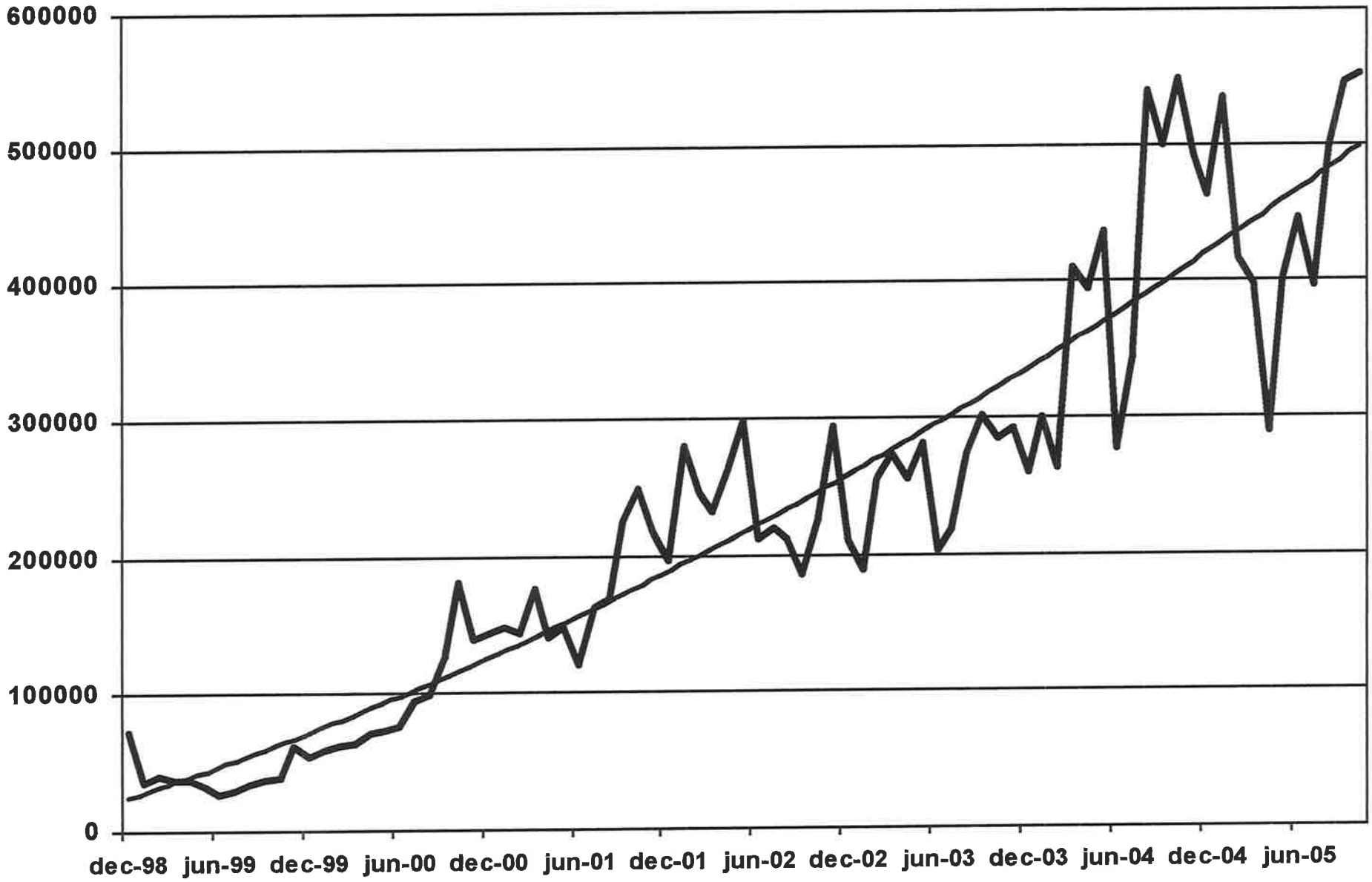
Updated 2005-11-02

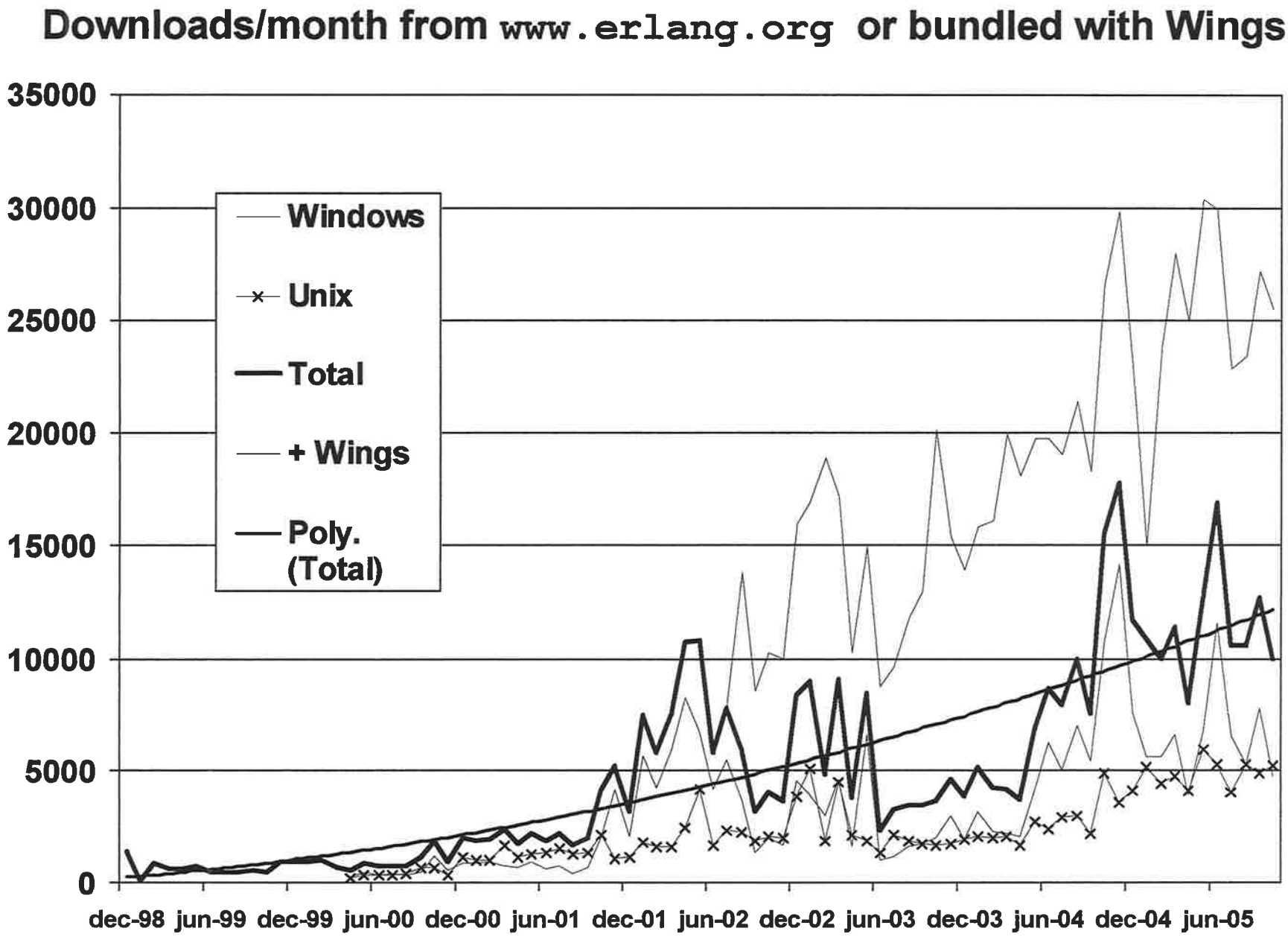
EUC participation



Updated 2005-11-02

Requests/month to www.erlang.org




Downloads/month from www.erlang.org or bundled with Wings
35000
30000
25000
20000
15000
10000
5000
0
Windows
Unix
Total
+ Wings
Poly. (Total)
dec-98
jun-99
dec-99
jun-00
dec-00
jun-01
dec-01
jun-02
dec-02
jun-03
dec-03
jun-04
dec-04
jun-05