Analysing and visualising callback modules of Erlang generic server behaviours

István Bozó
bozoistvan@caesar.elte.hu

Mátyás Béla Kuti
matyas.kuti@gmail.com

Melinda Tóth
tothmelinda@caesar.elte.hu

ELTE Eötvös Loránd University, Faculty of Informatics
Budapest, Hungary

Abstract

Understanding and maintaining the source code of industrial-scale software product is hard and time-consuming, and it is getting more difficult when the software implements parallel/concurrent/distributed computations and behaviours. Static source code analysis tools support program comprehension through detecting dependencies and relations among small (like functions and processes) or large (modules and components) software elements. For Erlang, which is a dynamic language, only an approximation of the real dependencies can be calculated statically. However it is possible to improve these analyses by adding application specific information, such as the most frequently used behaviours. In this paper we introduce an extension of the RefactorErl static source code analyser framework for Erlang that adds information about generic server behaviour implementations to the Semantic Program Graph of RefactorErl. In addition we define several views of the generic server based hidden communication.

1 Introduction

Tools that support the software development life-cycle (such as code comprehension tools, refactoring tools, visualisation tools, test selection tools, test coverage checkers, etc.) are frequently used by the industry. These tools can reduce the time of bug fixing, code change requests, new feature requests and they also help to decrease the number of faults in the system.

Static analyser tools extract information from the source code without executing the program. Static analysis is not straightforward for sequential programs and it is getting more difficult when we are to analyse complex concurrent or distributed software.

Erlang [13] is a concurrent functional programming language that was designed for developing telecommunication systems in the 1980s. RefactorErl [11] is a tool for Erlang, that aims to support semantics preserving source code

In: E. Vatai (ed.): Proceedings of the 11th Joint Conference on Mathematics and Computer Science, Eger, Hungary, 20th – 22nd of May, 2016, published at http://ceur-ws.org

transformations and code comprehension. It represents the source code in a so-called Semantic Program Graph (*SPG*). The *SPG* contains the abstract syntax tree of the source code, and in addition semantic nodes and edges representing semantic information. For instance, functions and function calls, and the connections generated by the usage of the built in communication primitives of the language, etc.

Erlang provides patterns (so called behaviours) for concurrent and distributed application design. There are several behaviours in the Erlang/OTP [16] library, such as generic servers, finite state machines, supervisors, etc.

The main focus of our paper is the extension of the Semantic Program Graph with application specific semantic information. We extend the communication and process semantic layer found in the *SPG* with the results of generic server analysis. We also define a separate graph, a view of a static interaction model (communication and message handling) of the generic server behaviour. This can be a vital tool when it comes to code comprehension and discovering defects or anomalies in the behaviour of a generic server.

The rest of this paper is structured as follows: Section 2 introduces the basic communication primitives of Erlang and the generic server behaviour; Section 3 introduces RefactorErl and the Communication Graph; Section 4 presents the Erlang generic server analysis and visualisation; Section 6 shows an example of the resulted model; Section 7 discusses related work and Section 8 concludes the paper and highlights future directions.

2 Processes in Erlang

Originally, Erlang was designed for developing concurrent and distributed software systems with soft real-time characteristics. Erlang supports process creation and communication between processes with built in language constructs.

To start processes one can use the `spawn*` functions with different options, for message sending the `!` operator and for receiving messages the `receive` construct. Processes can be accessed either with their process identifier (*pid*) or can be registered and accessed by the registered name.

2.1 Behaviours

Beside the communication primitives the Erlang OTP comes with a set of design patterns, so called *behaviours* [16]. Some well-known standard behaviours are:

- `gen_server` – Generic server pattern for client-server applications.
- `gen_fsm` – Generic finite state machine.
- `gen_event` – Generic event handler.

Besides the standard behaviours we can extend the above list by defining our own behaviours. For this, we have to define the required interface of the callback module and implement the general part of the behaviour.

2.2 Generic server behaviour

In this paper we focus only on static analysis of source codes implementing the `gen_server` behaviour. Let us introduce briefly the main features of the generic server behaviour, the additional features can be handled analogously. In Section 6 the reader can find an example source code.

2.2.1 Interface of the `gen_server` module

The interface functions of the `gen_server` module are:

- `start*/3,4` – The function starts the server with the provided callback module, arguments and options. The server can be accessed by its *pid* or, if it is registered, by its name. The visibility of the registered server can be local or global. For setting up the server process it uses the `init/1` function from the callback module.

- `call/3,4` – This function is for synchronous communication. The function sends the given request to the addressed server and waits for the response. The calling process is suspended until the reply arrives. Optionally it can be given a timeout value, that causes the function call to fail if there is no reply within that time.
- `cast/2` – This function is for asynchronous communication. The function sends the given message to the addressed server asynchronously.
- `reply/2` – With this function the server can send messages directly to the client. It can be used for delayed answers to the client.

2.2.2 Callback module

The callback module can be divided into two parts, the server API and the callback functions. The API functions depend on the purpose of the server: starting and stopping the server, interface functions to hide the communication with the server and forward correctly the requests to the server.

Let us describe only the subset of the callback functions that are the most relevant to our analysis:

- `init/1` – The function is called when the server is started, it initialises the server process using the input arguments.
- `terminate/2` – The function is evaluated if the server is stopping, it performs the necessary clean up operations according to the provided reason of stopping and state of the server.
- `handle_call/3` – The function is called when a synchronous request arrives to the server. The function gets the request data, the client id and the current state of the server. There are three different types of return values:
  - `reply` : The return encloses the reply message to the client request, the new server state and an optional timeout value or the `hibernate` atom. If timeout option is used, a timeout will occur if there is no new request within this time. If the `hibernate` option is used, the server goes into hibernation until the next request arrives.
  - `noreply` : There are situations that require to delay the answer to the client, this option can be used in this case. The return value encloses the new state and an optional timeout value or the `hibernate` atom. The delayed answer can be returned to client later on with the `reply/2` function.
  - `stop` : The return value must enclose the reason of stopping, the new state, and an optional reply to the request. If the request is not provided the reply must be handled with the `reply/2` function directly. This return value causes the server to stop, the `terminate/2` function is evaluated.
- `handle_cast/2` – The function is called when an asynchronous request arrives to the server. The function gets the request data and the current state of the server. There are two possible returns of this function:
  - `noreply` : The return value encloses the new server state and an optional timeout value or the `hibernate` atom.
  - `stop` : This return causes the server to stop. The return value encloses the reason of stopping and the new state of the server.
- `handle_info/2` – The function is called when a message arrives to the server (in other ways than the provided `gen_server` interface functions) or a timeout occurs. The server handles the message as asynchronous communication, the return values are the same as at the `handle_cast/2`.

3 RefactorErl

RefactorErl [11] is a source code analysis and transformation tool for Erlang. Besides the more than twenty refactoring steps it provides a variety of features to support software maintenance and development (e.g. code browsing and investigations controlled by semantic queries, duplicated code detection, dependence analysis and visualisation).

As the basis of further analysis RefactorErl uses a Semantic Program Graph (*SPG*) to represent the Erlang source code in a data structure designed to efficiently store and reuse the calculated lexical, syntactic and semantic information. The tool has an asynchronous semantic analyser framework and provides several built-in analyses: variable scoping, static and dynamic function call, data-flow, module, record, specification analyses.

3.1 Semantic Program Graph schema

The *SPG* can be described as the following hextuple: $SPG = (N, A_N, A_V, A, T, E)$,

where:

- N is the set of nodes,
- A_N is the set of attribute names,
- A_V is the set of possible attribute values,
- $A : N \times A_N \rightarrow A_V$ is a node labelling partial function,
- T is the set of edge labels,
- $E : N \times T \times \mathbb{N}_0 \rightarrow N$ a partial function that describes labelled, ordered edges between the nodes.

The set N corresponds to the three layers of the *SPG*, it is the union of the lexical, syntactic and semantic nodes:

$$N = N_{lex} \cup N_{syn} \cup N_{sem}$$

The set N_{syn} contains nodes of four distinct classes: file, form, clause and expression. Formally:

$$N_{syn} = N_{file} \cup N_{form} \cup N_{clause} \cup N_{expr}$$

The set N_{sem} contains nodes of several different classes, like module, function, variable, *pid*, behaviour, etc. Formally:

$$N_{sem} = N_{module} \cup N_{func} \cup N_{pid} \cup N_{behaviour} \cup ...$$

In further sections we use the $Node \in N_x$ notation which means that $Node$ is a node of class x.

3.2 Nodes in the *SPG*

We introduce the node classes that are necessary for the generic server analysis:

- **ROOT**: Root node of the *SPG*
- **Expr**: Syntactic nodes representing expressions. The `type` attribute of the node describes the type of the given expression, e.g. tuple, list, application, function parameter, etc.

- **Clause**: Syntactic nodes representing the clauses of functions and expressions.
- **Func**: Semantic nodes representing functions. The node describes the arity and the name of the function.
- **Module**: Semantic nodes representing modules.
- **Behaviour**: Semantic nodes of behaviours. Its `type` attribute describes the represented behaviour (e.g. `gen_server`).
- **Pid**: Semantic nodes representing processes. A node can either belong to a behaviour process or it can be an interface function that can be started as a separate process.

3.3 Data-Flow Reaching

To determine the identifiers of processes and the recipient of the messages we use data-flow analysis [22]. A Data-Flow Graph is built during the initial analysis and we use the first order data-flow reaching relation ($\overset{\mathbf{1f}}{\rightsquigarrow}$) to calculate the possible values of certain expressions. The notation $e_1 \overset{\mathbf{1f}}{\rightsquigarrow} e_2$ means that the value of expression e_1 can flow to expression e_2.

3.4 Behaviour nodes in the *SPG*

The behaviour analysis is performed when a file is being added to the database of RefactorErl. It extends the *SPG* with the behaviour information of modules. Whenever a behaviour attribute form is discovered in the source code, a new semantic node and a new edge with the label `beh` is inserted between semantic module node and the newly added behaviour node. Formally: $Module \overset{beh}{\rightarrow} Behaviour$

3.5 Process analysis

The process relations are represented in the Semantic Program Graph as described in [23]. The process analysis is a so called post analysis[1], therefore it has to be started by the user after the source code has been added to the database of RefactorErl.

The process discovery analysis examines the function applications which can start new processes, these are `erlang:spawn/1,2,3,4` and `erlang:spawn_link/1,2,3,4` and other `spawn_*` function applications. For each discovered process a new semantic `Pid` node is created in the *SPG*. The algorithm tries to extract as much information as possible about the started processes, such information are for example the registered names. Messaging primitives such as `send/2,3` function applications and `receive` expressions are examined as well to identify the communication between the processes.

The process analyser identifies `gen_server` processes and the synchronous and asynchronous requests from clients to servers [12].

3.5.1 *SPG* edges

Besides the semantic process identifier nodes several edges are created in the *SPG* to describe the communication between `gen_servers` and clients:

- `pid`: from the root node of the *SPG* to the process identifier nodes: $ROOT \overset{pid}{\rightarrow} Pid$
- `eval_in`: any expression that performs message passing (via communication primitives or `gen_server` interface functions) is connected to the process it is evaluated in: $Expr \overset{eval\_in}{\rightarrow} Pid$

[1]RefactorErl analyses the sources with its asynchronous incremental analyser framework, where each Erlang form is analysed in a separate Erlang process. When we rely our analysis on interfunctional information, such as data-flow, we have to run our analysis after the initial analysis is finished for the whole graph. These kinds of analyses are called post analyses.

- `sync_call`: process identifier nodes that perform `gen_server:call/2,3` are connected to the `Pid` node of the `gen_server` process: $Pid \overset{snyc\_call}{\rightarrow} Pid$
- `async_call`: process identifier nodes that perform `gen_server:cast/2` are connected to the `Pid` node of the `gen_server` process: $Pid \overset{async\_call}{\rightarrow} Pid$

4 Analysing generic server behaviours

The `gen_server` analysis extends the behaviour nodes with the gathered information and adds new edges that represent the communication between processes. The analysis uses the information from other analyses like behaviour, process and data-flow analyses.

The analyser first identifies the generic servers and for each server its possible registered name is determined. Next the algorithm examines the communication between processes. The algorithm looks for expressions that could potentially send messages to the `gen_server` process (using messaging primitives, synchronous or asynchronous message passing) and it determines the possible reply values based on the sent messages. The sent messages and possible replies are stored in the behaviour node. The next step is to insert new edges to the *SPG*. New edges are inserted between the behaviour node and functions that define, start or stop the server, and to functions that initiate message sending to the server process.

4.1 Semantic Program Graph extension

The *SPG* schema is extended to allow for storing communication information in behaviour nodes and the addition of new edges into the graph.

Behaviour nodes are extended with the `messages` attribute:

$$Messages \subseteq N_{pid} \times Term \times N_{expr} \times N_{clause} \times \mathcal{P}(Term)$$

In the described set, each message is described with a 5-tuple with the following elements:

1. $Pid \in N_{pid}$: The semantic process identifier node in the *SPG* of the message sender process.
2. $Message \in Term$: The sent message, an Erlang term.
3. $Expression \in N_{expr}$: The expression node in the *SPG* that performs the message sending.
4. $DestinationClause \in N_{clause}$: Syntactic node of the function clause that handles the message.
5. $Replies \in \mathcal{P}(Term)$: A list of Erlang terms that are possible replies to the message from the server side.

The message sending could be represented as edges from the process identifier nodes to the handling function clause with the message as the edge label. The reason why we introduced this complex attribute instead of edge labels was that the RefactorErl database has a fixed schema. This means every edge label has to be declared in advance, they cannot be changed dynamically. The advantage of this representation is that there is no need to perform costly queries in the graph, every piece of information is accessible directly from the semantic behaviour node.

The following new edges are introduced to the *SPG*:

- `gs_def`: Connects the `gen_server` behaviour semantic node to the `init/1` function in its callback module.
- `gs_call`: Connects the `gen_server` behaviour semantic node to those functions that apply the `gen_-server:call/2,3` function thus sending a synchronous request to the server process.
- `gs_cast`: Connects the `gen_server` behaviour semantic node those functions that apply the `gen_-server:cast/2` function thus sending an asynchronous request to the server process.

- `gs_start`: Points from the `gen_server` node to those function nodes that could start server process. These functions either call the `gen_server:start/3,4` or `gen_server:start_link/3,4` functions.
- `gs_stop`: Points from the `gen_server` node to those function nodes that could stop the server process. These functions call the `gen_server:stop/1,3` function.
- `gs_pid`: Connects the `gen_server` node to the `Pid` node, representing the server process.

4.2 Analysis of generic server behaviours

As the first step of the `gen_server` analysis it performs the process analysis. This step is shown in Algorithm 1, as the *analyse_processes*() subroutine call. After this we query the `gen_servers` from the graph by calling the `genservers()` subroutine and we apply the analysis for each found `gen_server`.

Algorithm 1 *analyse_genservers*()

```
1: $analyse\_processes()$
2: $GenServers \leftarrow genservers()$
3: for $Server \in GenServers$ do
4:     $analyse\_genserver(Server)$
5: end for
```

The algorithm for the `genservers()` subroutine is described in Algorithm 2. The subroutine filters and returns the set of behaviour nodes of type `gen_server`.

Algorithm 2 *genservers*()

```
1: $GenServers \leftarrow \emptyset$
2: for $Behaviour \in N_{behaviour}$ do
3:     if is_genserver($Behaviour$) then
4:         $GenServers \leftarrow GenServers \cup \{Behaviour\}$
5:     end if
6: end for
7: return $GenServers$
```

The abstract algorithm of extending the *SPG* with semantic knowledge about a `gen_server` is shown in Algorithm 3.

The routine gets a generic server semantic node as argument. As a first step the algorithm determines the possible names for the `gen_server` and updates the node. The second step is to determine the possible messages addressed to the examined server, the possible replies and update the node with the gathered information. The further steps are to insert edges between the examined `gen_server` node and its `Pid` node, `init/1` function, and functions that make either synchronous or asynchronous requests, start or stop the server.

Algorithm 3 *analyse_genserver*(*GenServer*)

```
1: $update\_with\_name(GenServer)$
2: $update\_with\_messages(GenServer)$
3: $create\_gs\_pid\_edge(GenServer)$
4: $create\_gs\_def\_edge(GenServer)$
5: $create\_gs\_call\_edges(GenServer)$
6: $create\_gs\_cast\_edges(GenServer)$
7: $create\_gs\_start\_edges(GenServer)$
8: $create\_gs\_stop\_edges(GenServer)$
```

5 The gen_server graph

The *SPG* itself contains an overwhelming amount of information, although it can be visualised with RefactorErl it is not the most appropriate form for human readers. To extract information produced by the `gen_server` analysis one can either generate views on different abstraction levels or query the information directly from the *SPG*.

In Section 4 we have presented how the *SPG* is extended with `gen_server` specific information. The most comprehensible presentation of this information is through visualisation, therefore we defined the `gen_server` communication graph to present the message passing between server processes and their clients.

We can generate the `gen_server` communication graph on three levels of abstraction, which correspond with the abstraction levels of the language itself (functions, function clauses, expressions):

- **Compact:** Server processes, callback modules and functions sending requests to server processes are displayed, along with the `gen_server:handle*` functions. Request and message contents are the labels of edges leading to the handle functions from the function initiating them.
- **Normal:** The compact view is extended with the clauses of the functions `gen_server:handle*`. The edges showing message passing are leading to the function clause handling them.
- **Detailed:** The normal view is extended with the actual expressions performing a request or sending a message, these are the sources of the request edges in the graph.

5.1 Graph definition

Let G_{gs} be the `gen_server` communication graph in the following way:

$$G_{gs} = (V_{gs}, E_{gs})$$
$$V_{gs} = V_{server} \cup V_{module} \cup V_{pid} \cup V_{handler} \cup V_{clause} \cup V_{expr}$$
$$E_{gs} \subseteq V_{gs} \times V_{gs} \times LABEL$$

Where V_{gs} and E_{gs} are the node and edge sets respectively. The set of nodes is composed of the following subsets:

- V_{server}: nodes of generic servers
- V_{module}: nodes representing modules
- V_{pid}: nodes of processes
- $V_{handler}$: nodes of `gen_server:handle*` functions of each server
- V_{clause}: nodes of clauses of the `gen_server:handle*` functions
- V_{expr}: expression nodes (that initiate a request)

Nodes in the communication graph have a *label* property, defining the visualised name of the node.

E_{gs} is the set of labelled edges, where each edge is represented with an ordered triple. The elements of the ordered triple are the head node, the tail node and the label of the edge. The elements of the $LABEL$ set are arbitrary Erlang terms.

5.2 Graph building rules

The graph building procedure can be described as declarative rules. The graph is built in a way that we match the rules on the *SPG* and build the corresponding genserver graph nodes and edges. There are several rules that are common for all three graph abstraction levels, and a few rules that are specific for the given abstraction level.

General rules

1. *`gen_server` node:* If a behaviour node represents a generic server in the *SPG*, it will be present in the communication graph.

$$\frac{is\_genserver(GenServer) \land callback\_module(Module, GenServer)}{GenServer' \in V_{server} \land GenServer'.label = Module.name}$$

2. *Callback module node:* Each `gen_server` callback module is present in the graph, with the module name as a label.

$$\frac{callback\_module(Module, GenServer)}{Module' \in V_{module} \land Module'.label = Module.name}$$

3. *Callback module edge:* From the `gen_server` node an edge with the label "*callback module*" leads to the callback module node.

$$\frac{callback\_module(Module, GenServer)}{(GenServer', Module', \text{"}callback\ module\text{"}) \in E_{gs}}$$

4. *`handle_call/3` message handler:* If a server callback module defines a `handle_call/3` function, it will be present in the graph.

$$\frac{\begin{array}{c} callback\_module(Module, GenServer) \land Module \overset{func}{\rightarrow} Function \land \\ Function.name = handle\_call \land Function.arity = 3 \end{array}}{\begin{array}{c} Function' \in V_{handler} \land \\ Function'.label = Module.name : Function.name/Function.arity \land \\ (Module', Function', \text{"}synchronous\ handler\text{"}) \in E_{gs} \end{array}}$$

5. *`handle_cast/2` message handler:* If a server callback module defines a `handle_cast/2` function, it will be present in the graph

$$\frac{\begin{array}{c} callback\_module(Module, GenServer) \land Module \overset{func}{\rightarrow} Function \land \\ Function.name = handle\_cast \land Function.arity = 2 \end{array}}{\begin{array}{c} Function' \in V_{handler} \land \\ Function'.label = Module.name : Function.name/Function.arity \land \\ (Module', Function', \text{"}asynchronous\ handler\text{"}) \in E_{gs} \end{array}}$$

6. *`handle_info/2` message handler:* If a server callback module defines a `handle_info/2` function, it will be present in the graph.

$$\frac{\begin{array}{c} callback\_module(Module, GenServer) \land Module \overset{func}{\rightarrow} Function \land \\ Function.name = handle\_info \land Function.arity = 2 \end{array}}{\begin{array}{c} Function' \in V_{handler} \land \\ Function'.label = Module.name : Function.name/Function.arity \land \\ (Module', Function', \text{"}message\ handler\text{"}) \in E_{gs} \end{array}}$$

7. *Pid nodes:* Process identifier nodes in the *SPG* will be present in the graph if they send a request to the server process.

$$\frac{\begin{array}{c} is\_genserver(GenServer) \land \\ (Pid, Message, Expression, DestClause, Replies) \in GenServer.messages \end{array}}{Pid' \in V_{pid} \land Pid'.label = Pid.module : Pid.function/Pid.arity}$$

8. *Module of Pid:* For process identifier nodes their module will be present in the graph as well.

$$\frac{\begin{array}{c} is\_genserver(GenServer) \land Module \in N_{module} \land \\ (Pid, Message, Expression, DestClause, Replies) \in g.messages \land \\ Module.name = Pid.module \end{array}}{Module' \in V_{module} Module'.label = Module.name \land (Module', Pid', \text{"}pid\text{"}) \in E_{gs}}$$

9. *Function of clause:*

$$\frac{Form \in N_{form} \wedge Clause \in N_{clause} \wedge Function \in N_{func} \wedge \\ Form \overset{funcl}{\to} Clause \wedge Form \overset{fundef}{\to} Function}{clause(Clause, Function)}$$

Compact graph rules

1. *Requests:* From every process that is a message source of a `gen_server` an edge leads to the proper message handling function, with the message as an edge label.

$$\frac{is\_genserver(GenServer) \wedge \\ (Pid, Message, Expression, DestClause, Replies) \in GenServer.messages \wedge \\ clause(DestClause, Function)}{(Pid', Function', Message) \in E_{gs}}$$

2. *Responses:* From the message handling functions an edge leads to the process that originally sent a request. The edge label is the response itself.

$$\frac{is\_genserver(GenServer) \wedge \\ (Pid, Message, Expression, DestClause, Replies) \in GenServer.messages \wedge \\ clause(DestClause, Function) \wedge Reply \in Replies}{(Function', Pid', Reply) \in E_{gs}}$$

Normal graph rules

1. *Function clauses:* Each function clause of the message handling functions of the server will be present in the graph.

$$\frac{is\_genserver(GenServer) \wedge \\ (Pid, Message, Expression, DestClause, Replies) \in GenServer.messages \wedge \\ clause(DestClause, Function)}{DestClause' \in V_{clause} \wedge \\ DestClause'.label = text(DestClause) \wedge \\ (Function', DestClause', "function\ clause") \in E_{gs}}$$

2. *Requests:* From every process that is a message source of a `gen_server` an edge leads to the proper message handling function clause, with the message as an edge label.

$$\frac{is\_genserver(GenServer) \wedge \\ (Pid, Message, Expression, DestClause, Replies) \in GenServer.messages}{(Pid', DestClause', Message) \in E_{gs}}$$

3. *Responses:* From the message handling function clauses an edge leads to the process that originally sent a request. The edge label is the response itself.

$$\frac{is\_genserver(GenServer) \wedge \\ (Pid, Message, Expression, DestClause, Replies) \in GenServer.messages \wedge \\ Reply \in Replies}{(DestClause', Pid', Reply) \in E_{gs}}$$

Detailed graph rules

1. *Function clauses:* Each function clause of the message handling functions of the server will be present in the graph.

$$\frac{\begin{array}{c} is\_genserver(GenServer) \land \\ (Pid, Message, Expression, DestClause, Replies) \in GenServer.messages \land \\ clause(DestClause, Function) \end{array}}{\begin{array}{c} DestClause' \in V_{clause} \land \\ DestClause'.label = text(DestClause) \land \\ (Function', DestClause', \text{"}function\ clause\text{"}) \in E_{gs} \end{array}}$$

2. *Message sending expressions:* Each expression that is a message source will be a node in the communication graph.

$$\frac{\begin{array}{c} is\_genserver(GenServer) \land \\ (Pid, Message, Expression, DestClause, Replies) \in GenServer.messages \end{array}}{\begin{array}{c} Expression' \in V_{expression} \land (Pid', Expression', \text{"}send\ expression\text{"}) \in E_{gs} \land \\ Expression'.label = text(Expression) \end{array}}$$

3. *Requests:* From every expression that is a message source of a `gen_server` an edge leads to the proper message handling function clause, with the message as an edge label.

$$\frac{\begin{array}{c} is\_genserver(GenServer) \land \\ (Pid, Message, Expression, DestClause, Replies) \in GenServer.messages \end{array}}{(Pid', Expression', Message) \in E_{gs}}$$

4. *Responses:* From the message handling function clauses an edge leads to the process that originally sent a request. The edge label is the response itself.

$$\frac{\begin{array}{c} is\_genserver(GenServer) \land \\ (Pid, Message, Expression, DestClause, Replies) \in GenServer.messages \land \\ Reply \in Replies \end{array}}{(DestClause', Pid', Reply) \in E_{gs}}$$

6 Evaluation

The introduced analysis technique has been applied to larger open-source project, such as RabbitMQ [6] and EMQTTD [3]. The results were published as a student research paper [17].

In this section we present a demonstrating example, a simplified chat client-server application, implemented with the generic server behaviour. We discuss the source code in detail and show the different views of the generic server graphs described in previous sections.

6.1 The server

The vital parts of the implemented chat server are introduced in Figure 1. There is a macro definition `SrvRef` to make it easier to refer the server as it is registered globally. The functions of the server module are divided into three groups: interface function for managing (starting, stopping) the server, interface functions for clients (connecting to the server, sending messages) and the callback functions of the server.

Server management

The server can be started with the `start/1` function. The argument of the function is the maximum number of simultaneous users. The function starts the server with the given name, callback module and initialising arguments.

The generic server can be terminated with the `stop/0` function.

Client interface

The module provides interface function to the clients to hide specific information of the server. Such information is the name of the server and how to interact with the server process.

The client can connect to the server through evaluating the `chatserver:connect/1` providing the nick as an argument. The function performs a synchronous request to connect to the server with the given nick.

The client can disconnect from the server by calling the `chatserver:disconnect/0`. The function performs an asynchronous request that sends the server the `quit` message.

The texting to other chat clients is performed by the `chatserver:send/1` function. The function sends the text and the *pid* of the sender to the server. Messages can be sent directly to the server with the `chatserver:other_-message/1` function.

Callback functions

The server receives requests and react to them using defined callback functions.

The `chatserver:init/1` function is called when the server is started, it performs the initialisation and returns the initial state of the server.

The synchronous requests sent with `gen_server:call/2,3` functions are handled with `handle_call/3` function defined in the callback module. In our implementation there is a synchronous request when a client initiates connection to the server process. When the request arrives the server checks whether the user limit has been reached. If there is no free space in the chat room, a `deny` message is replied to the caller. In this case the state remains unchanged. If the chat room is not full the user list is extended with the new member. The reply will be the `ok` atom, and the state of the server is updated with the new user.

The asynchronous requests sent with `gen_server:cast/2` function are handled with the `handle_cast/2` function defined in the callback module. In our example implementation there are three different cases. The first clause of the function handles the text broadcasting among the chat users in the chat room. It receives the message and the identifier of the sender. From the identifier the server determines the nick of the user (it is assumed that the sender is connected to the server) and sends the composed message for every member of the chat room. The second clause handles the disconnection request from the users. The server omits the requested user from the list of users and updates its sate. The third case is for stopping the server.

The requests/messages sent in a non standard way to the generic server are handled with the `handle_info/2` function definition of the callback module. In our case any unexpected message causes the server to stop.

6.2 The client

The client is composed of two interface function `start/1` and `send/2` and two auxiliary functions `loop/0` and `input/1`.

Interface functions

The function `start/1` initiates the client process by sending a request `connect` to the server. The client is terminated if a `deny` message is received. If the connection succeeded it spawns the input reading process and continues executing the `loop/0` function.

The `send/2` function is an interface function for the server process. With the help of this function the server can send easily messages to clients.

```
%% Macro definition of server reference
-define(SrvRef, {global, chatserver}).

%% Interface functions
start(Max) -> gen_server:start(?SrvRef, chatserver, [Max], []).

stop() -> gen_server:cast(?SrvRef, stop).

%% Interface functions for clients
connect(Nick) -> gen_server:call(?SrvRef, {connect, Nick}).

disconnect() -> gen_server:cast(?SrvRef, {quit, self()}).

send(Text) -> gen_server:cast(?SrvRef, {text, self(), Text}).

other_message(Message) -> chatserver ! Message.

%% Callback functions
init(Max) -> {ok, #state{users = [], max = Max}}.

handle_call({connect, Nick}, {Pid, _}, State = #state{users = Users}) ->
    if length(Users) >= State#state.max -> {reply, deny, State};
        true ->
            NewUsers = [#user{nick = Nick, pid = Pid} | Users],
            {reply, ok, State#state{users = NewUsers}}
    end.

handle_cast({text, From, Text}, State = #state{users = Users}) ->
    Nick = get_nick(From, Users),
    lists:foreach(
      fun(#user{pid = Pid}) ->
              chatclient:send(Pid, Nick ++ ": " ++ Text)
      end,
      Users),
      {noreply, State};
handle_cast({quit, From}, State = #state{users = Users}) ->
    NewUsers = omit_user(From, Users),
    {noreply,  State#state{users = NewUsers}};
handle_cast(stop, State) -> {stop, normal, State}.

handle_info(_, State) -> {stop, normal, State}.
```

Figure 1: Functions of the `chatserver` module

```
start(Nick) when is_list(Nick)->
    case chatserver:connect(Nick) of
        deny -> io:format("Connection failed~n",[]);
        ok   ->
            Loop = self(),
            spawn(fun() -> input(Loop) end),
            loop()
    end.
loop() ->
    receive
        quit -> chatserver:disconnect(), quit;
        {send, Text} -> chatserver:send(Text), loop();
        {text, Text} -> io:format("~s~n", [Text]), loop()
    end.
send(Pid, Text) -> Pid ! {text, Text}.
input(Loop) ->
    case string:strip(io:get_line('--> '), right, $\n) of
        "#q" -> Loop ! quit, ok;
        S    -> Loop ! {send, S}, input(Loop)
    end.
```

Figure 2: Functions of the `chatclient` module

Auxiliary functions

The input process iteratively reads the standard input and forwards the read text or command to the client process. The `loop/0` function receives messages continuously and perform actions based on these messages. If the loop receives the `quit` message the client disconnects from the server and exits looping. If a tuple with the `send` tag is received a message is sent to the server process. If a tuple with `text` tag is received, that is a message from the server with text of other chat users, it is printed to the standard output and continues looping.

6.3 Generic server views

The compact graph in Figure 3 shows the `chatserver` process, its callback module and the client module, `chatclient`. The message sending functions and the message handlers are featured as well, with the messages and responses as edges between them.

The normal graph (Figure 4) compared to compact graph contains the individual function clauses of the message handler functions. Each message is displayed as an edge from the sender to the handler clause.

The detailed graph in Figure 5 is the extension of the normal graph with the expressions that perform the message sending.

7 Related work

Supporting code comprehension by static analysis tool is not unique, several tools exist for different programming languages [2, 8, 5, 7, 4]. A few static analysers exist for Erlang as well, but none of them is focusing on `gen_server` based client-server implementation visualisation.

However the Erlang Verification Tool [10] is an interactive verification tool for proving correctness of distributed systems implemented in Erlang, also capable of reasoning about servers implemented with the `gen_server` behaviour.

The paper [14] introducing a message passing analysis for Erlang based on control flow graphs. The main purpose of this work was to detect some common message passing errors in Erlang source code. The paper is not focusing on the Erlang OTP behaviours. This tool is using some features of the static analyser tool, Dialyzer [18]. The main goal of Dialyzer is to identify software discrepancies and defects. In [9] the `-callback` attributes are introduced and used to analyse the possible misuses of OTP behaviours based on Dialyzer.

Researches have been done in the area of formal analysis (verification) of problems related to message passing. In paper [20] the authors defines race condition analysis, deadlock detection, etc. for *MPI* (Message Passing Interface).

Message passing analyses have been developed to build accurate data- [21] and control-flow [19, 21] graphs of *MPI* programs as well. The data-flow analysis technique presented in [21] is used for activity analysis and constant reaching analysis. The former analysis is used to reduce the computation and storage requirements of *MPI* programs.

Percept2 [15] is a profiler and tracer tool for Erlang software, with heavy focus on processes and process communication. It allows the visualisation of process hierarchies and the display of message passing between processes. Percept2 does not differentiate between processes thus has no specific features for generic server processes.

Akka [1] is a platform that allows for writing robust and fault tolerant software, using the same principles as Erlang, in Scala or Java. It provides the same functionality as a generic server behaviour, as it supports the actor model, but there is no specific static analysis tool available for Akka yet.

8 Summary

Under the development and maintenance of a concurrent software the developer has to properly understand the source code. In this paper we have introduced a static analyser tool that detects and visualise the communication among processes implemented using the generic server behaviour and their clients.

We have introduced a static source code analysis and transformation tool, RefactorErl. An extension of its existing process analysis is presented to examine source code that implements a client-server architecture with Erlang generic server behaviour.

We have described the visualisation of the communication between server processes and the clients that use them, in the form of the `gen_server` Communication Graph. We have evaluated our tool on several open source projects and it was able to produce a useful view of the server.

In a distributed Erlang environment the process replacement is explicit, therefore the developer has to deal with this information as well. Thus as a future work we aim to extend our analysis with distributed node analysis.

References

[1] Akka documentation. `http://doc.akka.io/docs/akka/snapshot/general/`. Accessed: 2016-06-30.

[2] CodeCompass. `https://github.com/Ericsson/codecompass`. Accessed: 2017-10-09.

[3] Erlang MQTTD Broker source. `https://github.com/emqtt/emqttd`. Accessed: 2017-10-01.

[4] Juliet. `http://infotectonica.com/`. Accessed: 2017-10-09.

[5] OpenGrok. `http://opengrok.github.io/OpenGrok/`. Accessed: 2017-10-09.

[6] RabbitMQ server source. `https://github.com/rabbitmq/rabbitmq-server`. Accessed: 2017-10-01.

[7] SourceInsight. `https://www.sourceinsight.com/`. Accessed: 2017-10-09.

[8] Understand. `https://scitools.com/features/`. Accessed: 2017-10-09.

[9] Stavros Aronis and Konstantinos Sagonas. Typed callbacks for more robust behaviours. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, pages 23–29, New York, NY, USA, 2011. ACM.

[10] Thomas Arts and Thomas Noll. *Verifying Generic Erlang Client—Server Implementations*, pages 37–52. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[11] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Kőszegi, Máté Tejfel, and Melinda Tóth. RefactorErl, Source Code Analysis and Refactoring in Erlang. In *Proceeding of the 12th Symposium on Programming Languages and Software Tools*, Tallin, Estonia, 2011.

[12] István Bozó and Melinda Tóth. Analysing and visualising Erlang behaviours. *AIP Conference Proceedings*, 1738(1), 2016.

[13] Francesco Cesarini and Simon Thompson. *Erlang Programming.*
O'Reilly Media, 2009.

[14] Maria Christakis and Konstantinos Sagonas. Detection of Asynchronous Message Passing Errors Using Static Analysis. In Ricardo Rocha and John Launchbury, editors, *Practical Aspects of Declarative Languages*, volume 6539 of *Lecture Notes in Computer Science*, pages 5–18. Springer Berlin Heidelberg, 2011.

[15] Huiqing Li and Simon Thompson. Multicore Profiling for Erlang Programs Using Percept2. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, pages 33–42, New York, NY, USA, 2013. ACM.

[16] Martin Logan, Eric Merritt, and Richard Carlsson. *Erlang and OTP in Action.* Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[17] Mátyás Kuti. Erlang viselkedések elemzése, 2016.

[18] Konstantinos Sagonas. Experience from Developing the Dialyzer: A Static Analysis Tool Detecting Defects in Erlang Applications. Talk presented at the 2005 Workshop on the Evaluation of Software Defect Detection Tools (Bugs'05), Chicago, June 2005.
`http://user.it.uu.se/~kostis/Papers/bugs05.pdf`. Accessed: 2016-06-30.

[19] Dale Shires, Lori Pollock, and Sara Sprenkle. Program Flow Graph Construction for Static Analysis of MPI Programs. In *Parallel and Distributed Processing Techniques and Applications*, pages 1847–1853, Jun 1999.

[20] Stephen F. Siegel and Ganesh Gopalakrishnan. Formal Analysis of Message Passing. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 2–18. Springer Berlin Heidelberg, 2011.

[21] Michelle Mills Strout, Barbara Kreaseck, and Paul D. Hovland. Data-Flow Analysis for MPI Programs. In *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, pages 175–184, Washington, DC, USA, 2006. IEEE Computer Society.

[22] Melinda Tóth and István Bozó. Static Analysis of Complex Software Systems Implemented in Erlang. In *Central European Functional Programming School*, volume 7241 of *Lecture Notes in Computer Science*, pages 440–498. Springer, 2012.

[23] Melinda Tóth and István Bozó. Detecting and Visualising Process Relationships in Erlang. *Procedia Computer Science*, 29(0):1525 – 1534, 2014. 2014 International Conference on Computational Science.

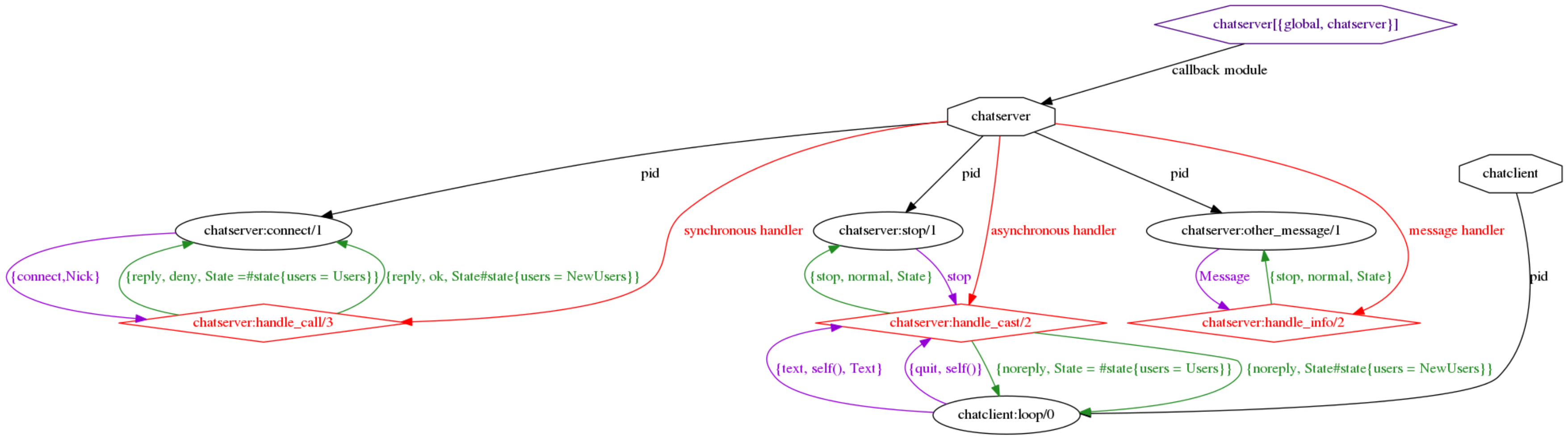

Figure 3: Compact `gen_server` communication graph

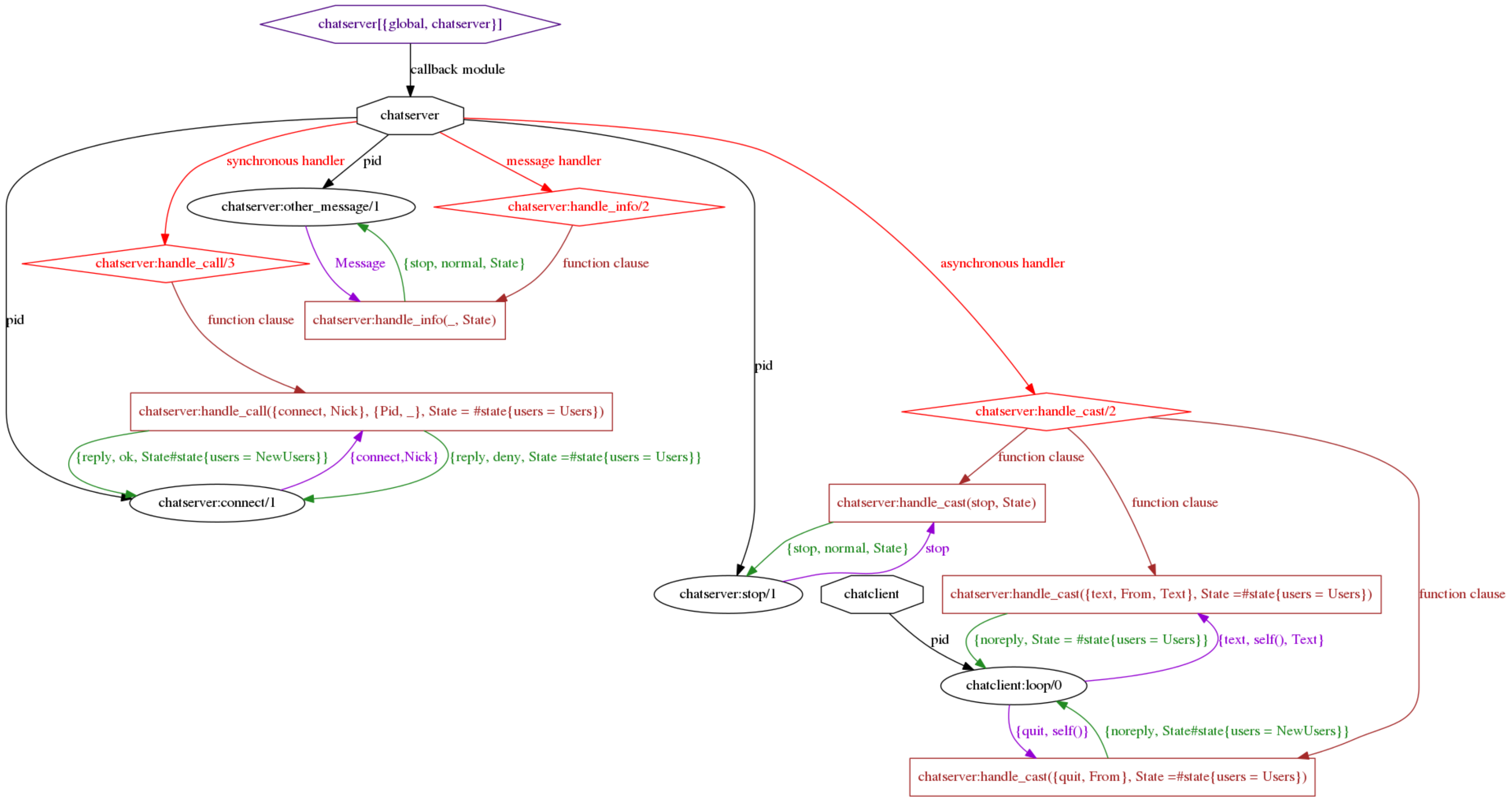
chatserver[{global, chatserver}]
callback module
chatserver
synchronous handler
pid
message handler
chatserver:other_message/1
chatserver:handle_info/2
chatserver:handle_call/3
Message
{stop, normal, State}
function clause
asynchronous handler
pid
chatserver:handle_info(_, State)
function clause
pid
chatserver:handle_call({connect, Nick}, {Pid, _}, State = #state{users = Users})
chatserver:handle_cast/2
{reply, ok, State#state{users = NewUsers}}
{connect,Nick}
{reply, deny, State =#state{users = Users}}
function clause
chatserver:connect/1
chatserver:handle_cast(stop, State)
function clause
{stop, normal, State}
stop
chatserver:stop/1
chatclient
chatserver:handle_cast({text, From, Text}, State =#state{users = Users})
function clause
pid
{noreply, State = #state{users = Users}}
{text, self(), Text}
chatclient:loop/0
{quit, self()}
{noreply, State#state{users = NewUsers}}
chatserver:handle_cast({quit, From}, State =#state{users = Users})

Figure 4: Normal `gen_server` communication graph

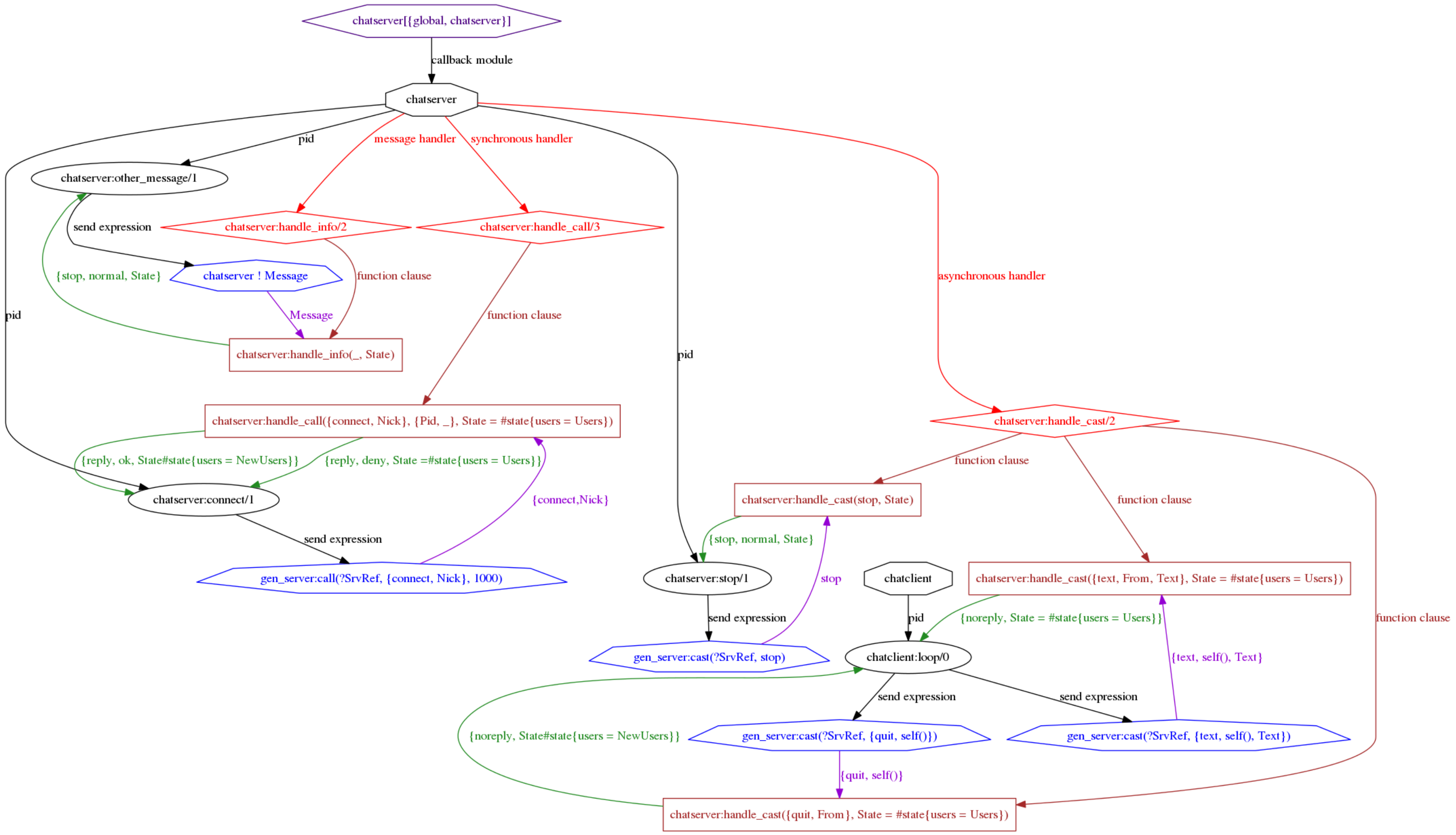
chatserver[{global, chatserver}]
callback module
chatserver
pid
message handler
synchronous handler
chatserver:other_message/1
send expression
chatserver:handle_info/2
chatserver:handle_call/3
asynchronous handler
{stop, normal, State}
chatserver ! Message
function clause
Message
function clause
pid
chatserver:handle_info(_, State)
pid
chatserver:handle_call({connect, Nick}, {Pid, _}, State = #state{users = Users})
chatserver:handle_cast/2
{reply, ok, State#state{users = NewUsers}}
{reply, deny, State =#state{users = Users}}
function clause
chatserver:connect/1
{connect,Nick}
chatserver:handle_cast(stop, State)
function clause
send expression
{stop, normal, State}
gen_server:call(?SrvRef, {connect, Nick}, 1000)
chatserver:stop/1
stop
chatclient
chatserver:handle_cast({text, From, Text}, State = #state{users = Users})
send expression
pid
{noreply, State = #state{users = Users}}
function clause
gen_server:cast(?SrvRef, stop)
chatclient:loop/0
{text, self(), Text}
send expression
send expression
{noreply, State#state{users = NewUsers}}
gen_server:cast(?SrvRef, {quit, self()})
gen_server:cast(?SrvRef, {text, self(), Text})
{quit, self()}
chatserver:handle_cast({quit, From}, State = #state{users = Users})

Figure 5: Detailed `gen_server` communication graph