Choreography-Based Analysis of Distributed Message Passing Programs

Ramsay Taylor
Dept. of Computer Science, University of Sheffield

Emilio Tuosto
Dept. of Computer Science, University of Leicester

Neil Walkinshaw
Dept. of Computer Science, University of Leicester

John Derrick
Dept. of Computer Science, University of Sheffield

***Abstract*—Distributed programs are vulnerable to subtle communication faults (e.g. deadlocks, lost messages). Although the behaviour of individual processes can be relatively straightforward, faults can still be hard to detect because the macroscopic behaviour emerging from interactions can easily become very complex. This paper reports the analysis of `gen_server`, an Erlang popular library to build client-server applications. Our analysis uses a tool based on choreographic models of communicating finite state machines (CFSMs). We discuss how, once the library has been modelled in terms of CFSMs, an automated analysis can be used to detect potential communication errors. The results of our analysis suggest how to properly use `gen_server` in order to guarantee the absence of communication errors.**

1. Introduction

The emergence of concurrent and distributed architectures, from multi-core processors to web-services, has had a substantial impact on software development. Languages with core support for concurrency and distribution such as Erlang, Elixir, Scala, and Go have all gained increasing prominence beyond their traditional domains of telecoms and finance. For example Erlang (which we focus on in this paper) now powers WhatsApp[1], Facebook[2], and Basho Riak - a distributed database that was recently chosen to underpin the UK National Health Service IT infrastructure[3].

This raises significant challenges from a verification perspective. Aside from ensuring that individual processes within the system behave correctly, it also becomes necessary to ensure that the macroscopic behaviour of multiple processes does not lead to 'distributed' communication faults, such as loss of messages, messages being sent to processes that are not expecting them, or deadlocks due to miscommunications.

Existing techniques to verify these systems can be broadly divided into three camps: Model-based verification, static source-code analysis, and test-driven analysis. Whilst these approaches all have their merits, none has yielded a scalable methodology that can systematically identify all such communication faults.

Recently *choreographies* have been advocated as a suitable basis for the analysis of distributed applications [1], [2], [3], [4]. A choreography models interactions among processes from a global point of view and provides a tractable basis for detecting the presence (or conversely guaranteeing the absence) of communication faults.

Our contributions are:

- A methodology to model communication behaviour in Erlang systems as communicating state machines.
- A choreography-based analysis to detect communication faults or to guarantee their absence.
- We show an application of our methodology to analyse the popular Erlang `gen_server` library.
- A few important vulnerabilities of `gen_server` that can undermine the communication behaviour and lead to deadlocks and loss of information.

Structure Section 2 surveys the Erlang's communication facilities, introduces the various message-passing faults using simple Erlang programs, and recalls *global graphs* and *communicating finite state machines*, which we use as a basis for detecting these faults. Section 3 provides a technique by which to construct communicating state machine models for Erlang systems. Section 4 presents the `gen_server` case study. Section 5 presents our conclusions and future work.

2. Background

2.1. Erlang by Examples

Figure 1 illustrates our first simple running example, and shows the core concepts of Erlang relevant to this paper. It uses a simple, self-contained module containing the implementation of client and server of the ping-pong protocol. Erlang processes communicate via message passing, and each process has a *mailbox* – a FIFO queue where messages sent by other processes are kept.

Erlang has a extensive built-in mechanisms for concurrency. New processes can be executed using the `spawn` command, which returns a process ID (PID) corresponding to the spawned process. For example, lines 27 and 28 of Figure 1 spawn processes that execute the `pong`

1. https://www.erlang-solutions.com/about/news/erlang-powered-whatsapp-exceeds-200-million-monthly-users
2. https://www.facebook.com/note.php?node_id-14218138919
3. http://basho.com/nhs-to-deploy-riak-for-it-backbone-with-quality-of\-care-improvements-in-sight/

```
1  -module(example).
2  -export([start/0, ping/2, pong/0]).
3
4  ping(0, Pong_PID) ->
5    Pong_PID ! finished,
6    io:format("ping finished~n", []);
7
8  ping(N, Pong_PID) ->
9    Pong_PID ! {ping, self()},
10   receive
11     pong ->
12       io:format("Ping received pong~n", [])
13   end,
14   ping(N - 1, Pong_PID).
15
16 pong() ->
17   receive
18     finished ->
19       io:format("Pong finished~n", []);
20     {ping, Ping_PID} ->
21       io:format("Pong received ping~n", [])
22       Ping_PID ! pong,
23       pong()
24   end.
25
26 start() ->
27   Pong_PID = spawn(example, pong, []),
28   spawn(example, ping, [3, Pong_PID]).
```

Figure 1. Erlang ping-pong message-passing example.

and `ping` functions. In its `receive` statement, `pong` inspects its mailbox for a message, which can either be the atom '`finished`' (line 18), or be the tuple '`{ping, Ping_PID}`' (line 20). If `finished` is received it will terminate, but if the tuple `{ping, Ping_PID}` is received it will send the atom `pong` back to the process identified by `Ping_PID` and then recurse. This recursion allows the process executing the `pong` function to receive as many pings as are sent before a `finished` message.

Variables in Erlang are bound to values by 'pattern-matching'. For example, the function `ping` is defined with two clauses and takes two parameters. If the first parameter matches the literal value `0` then the first clause is executed with variable `Pong_PID` bound to the second actual parameter. Otherwise the second clause is executed, with `N` and `Pond_PID` assigned to the first and second actual parameters respectively. Process `ping` is parametrised on `N`: If `N=0`, `ping` sends `pong` a '`finished`' message (line 5) and terminates, otherwise it sends `pong` a '`ping`' message, waits for a '`pong`' message (lines 10-13), and recurses with `N-1` (line 14).

2.2. Motivation: Communication Errors

In languages such as Erlang, message-passing makes programs vulnerable to *communication errors*. There are a variety of ways to characterise such errors, here we use that provided in [5], building on a similar characterisation in [6], as follows:

- **Deadlocks**: A process (or collection of processes) end up waiting for a message that will never arrive.
- **Orphaned messages**: A process sends a message, but this message is never consumed (perhaps because the target process has terminated).
- **Unspecified receptions**: A process receives a message, but is not in a state suitable to process it.

We show how communication errors can be easily introduced using the example in Figure 1.

Deadlock If the programmer types

```
Ping_PID ! poong,
```

instead of line 22 in Figure 1, a deadlock occurs because `ping` will not consume the misspelt message and wait indefinitely for a response.

Orphan messages Replacing the `start` function with

```
start() ->
  Pong_PID = spawn(example, pong, []),
  spawn(example, ping, [3, Pong_PID]),
  spawn(example, ping, [5, Pong_PID]).
```

triggers three processes: Two `ping`s and one `pong`; one `ping` will finish before the other and send a `finished` message terminating `pong`, so leaving the messages of the other process in `pong`'s mailbox.

Unspecified reception If the programmers omits the `finished` case from the `receive` clause of `pong`

```
receive
  {ping, Ping_PID} ->
    io:format("Pong received ping~n", []),
    Ping_PID ! pong,
    pong()
end.
```

the `finished` message will never be consume preventing the program from ever terminating.

It is important to note that such faults are impossible to detect by looking at the individual processes in isolation. They *emerge* from the interactions between processes, which demands a macroscopic overview of the system. This is what makes them especially difficult to detect, and their absence difficult to verify.

There have been attempts to detect specific errors in Erlang programs in the past, albeit with limited success. Modelling asynchronous Erlang programs as CCS or CSP, which presume synchronous communication, is impractical because it requires the explicit modelling of associated buffer and carrier processes [7]. Several static source code analysis approaches have been devised to detect deadlocks [8], [9], however these can be highly inaccurate. More recently, several testing approaches have emerged such as PULSE [10] and Concuerror [11], however these can only be used to reason about the executed program behaviour (i.e. they cannot, in general, guarantee to detect all instances of the above errors).

2.3. Choreographies

Recently *choreographies* have emerged as a promising basis for modelling the behaviour of distributed applications (see e.g., [1], [2], [3], [4]). A choreography models distributed interactions of several *participants* from a *global point of view* (cf. [2]).

We model the participants of a choreography as *communicating finite state machines* (CFSMs) [6]:

Definition 1 (Communicating FSMs) *Let* p, q *range over (machine) names,* Σ *be a set of messages, and* Act *be a set of* actions*, where an action* $\ell \in Act$ *is a* sending *action* $\ell = \mathsf{pq!a}$*, namely* p *writes message* $\mathsf{a} \in \Sigma$ *in the buffer to* q*) or a* receiving *action* $\ell = \mathsf{pq?a}$ *from channel* pq *(namely* q *inputs message* a *from the channel from* p*). A* communicating finite state machine *is a finite transition system given by a 4-tuple* $M = (Q, q_0, \Sigma, \rightarrow)$ *where*

- Q *is a finite set of* states*,*
- $q_0 \in Q$ *is the initial state, and*
- $\rightarrow \subseteq Q \times Act \times Q$ *is a set of* transitions*; we write* $q \xrightarrow{\ell} q'$ *for* $(q, \ell, q') \in \rightarrow$*.*

Given a CFSM $M_{\mathsf{p}} = (Q_{\mathsf{p}}, q_{0\mathsf{p}}, \Sigma, \rightarrow_{\mathsf{p}})$ *for each* $\mathsf{p} \in \mathcal{P}$*, the tuple* $\vec{S} := (M_{\mathsf{p}})_{\mathsf{p}\in\mathcal{P}}$ *is a* communicating system*.*

The semantics of communicating systems is defined in terms of *transition systems*, which keeps track of the state of each machine and the content of each buffer.

Definition 2 (Transition systems) *Let* $\mathbf{S} = (M_{\mathsf{p}})_{\mathsf{p}\in\mathcal{P}}$ *be a* communicating system*. A* configuration *of* $\mathbf{S}$ *is a pair* $s = \langle \mathbf{q} \ ; \ \mathbf{w} \rangle$ *where* $\mathbf{q} = (q_{\mathsf{p}})_{\mathsf{p}\in\mathcal{P}}$ *with* $q_{\mathsf{p}} \in Q_{\mathsf{p}}$ *and where* $\mathbf{w} = (w_{\mathsf{pq}})_{\mathsf{pq}\in C}$ *with* $w_{\mathsf{pq}} \in \Sigma^*$*;* $\mathbf{q}$ *is* initial *when* q_{p} *is the initial state of the corresponding CFSM and all buffers are empty.*

A configuration $s' = (\mathbf{q}'; \mathbf{w}')$ *is* reachable *from another configuration* $s = (\mathbf{q}; \mathbf{w})$ *by* firing transition ℓ*, written* $s \overset{\ell}{\Rightarrow} s'$ *if there is* $a \in \Sigma$ *such that either (1) or (2) below hold:*

1. $\ell = \mathsf{sr!a}$ *and* $(q_{\mathsf{s}}, \ell, q'_{\mathsf{s}}) \in \delta_{\mathsf{s}}$ *and*
   a. $q'_{\mathsf{p}} = q_{\mathsf{p}}$ *for all* $\mathsf{p} \neq \mathsf{s}$
   b. *and* $w'_{\mathsf{sr}} = w_{\mathsf{sr}}.a$ *and* $w'_{\mathsf{pq}} = w_{\mathsf{pq}}$ *for all* $\mathsf{pq} \neq \mathsf{sr}$

2. $\ell = \mathsf{sr?a}$ *and* $(q_{\mathsf{r}}, \ell, q'_{\mathsf{r}}) \in \delta_{\mathsf{r}}$ *and*
   a. $q'_{\mathsf{p}} = q_{\mathsf{p}}$ *for all* $\mathsf{p} \neq \mathsf{r}$
   b. *and* $w_{\mathsf{sr}} = a.w'_{\mathsf{sr}}$ *and* $w'_{\mathsf{pq}} = w_{\mathsf{pq}}$ *for all* $\mathsf{pq} \neq \mathsf{sr}$.

The reflexo-transitive closure of $\rightarrow$ *is* $\rightarrow^*$*. A sequence of transitions is* k-bounded *if no channel of any intermediate configuration on the sequence contains more than* k *messages. The set of* reachable configurations of S *is* $\mathbf{RS}(S) = \{s \mid s_0 \rightarrow^* s\}$*. The* k-reachability set of S *is the largest subset* $\mathbf{RS}_k(S)$ *of* $\mathbf{RS}(S)$ *within where configurations can be reached by a* k*-bounded execution from* s_0*.*

Condition (1) in Definition 2 puts a on channel sr, while (2) gets a from channel sr.

In [5] a condition is given to guarantee the absence of communication errors, which enables the automated transformation from transition systems into *global graphs* [12].

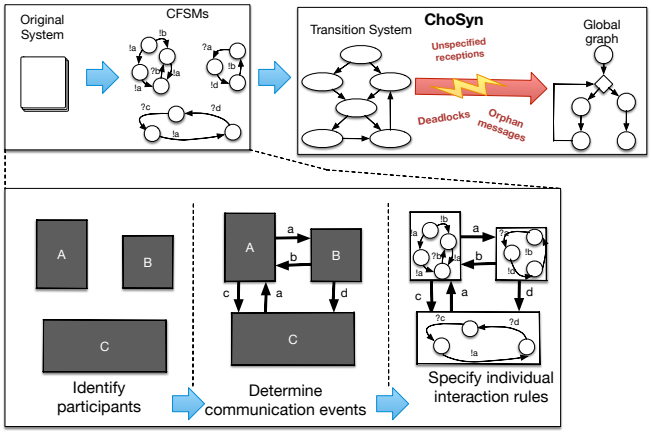


Figure 3. The process of extracting and analysing choreographies. The top-half shows the essential, high-level process. The bottom half provides an outline for our process of extracting CFSMs.

As a formal model of choreography, global graphs capture the overall behaviour of participants. The global graph in Figure 2(a) represents the ping-pong protocol: Following the path from the initial node, the pinger chooses whether to send `ping` or `finished`; in the former case the protocol loops after the participants exchange the `pong` message. The global graph in Figure 2(a) corresponds to the transition system in Figure 2(b) (more are given details in Section 3.4); both models are automatically computed by ChoSyn [13], the tool we will us here for the analysis of `gen_server`.

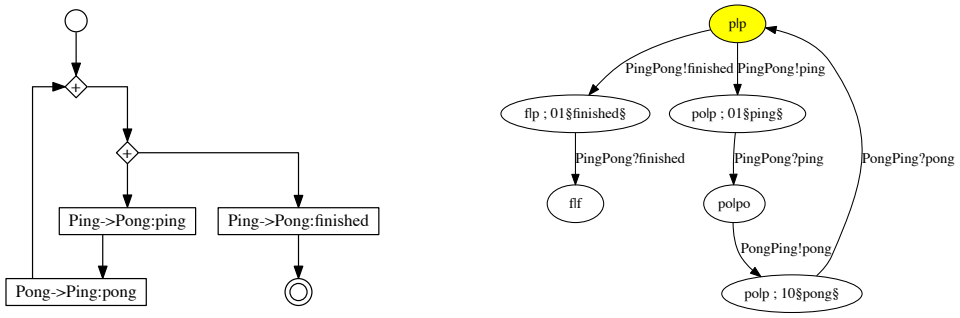
3. Modelling and Analysis

The process of modelling Erlang programs as choreographies is illustrated in the top half of Figure 3. There are two basic phases: (1) build CFSMs from the program under analysis and (2) apply the global graph construction and analysis (cf. [5]) to identify communication errors.

Ideally, automatic model extraction approaches should be applied in phase (1); however, we note that it is not always possible or necessary to do so. For instance, (i) for the analysis of `gen_server` the code of clients or servers may not be available, (ii) in some application domains it could be readily available from informal specifications, or (iii) it could be hard to identify the right level of abstraction at which a choreography faithfully represents the communication pattern of the program. Phase (1) requires some ingenuity due to function calls and scoping issues.

Function calls: In Erlang, the behaviour of a process can be contingent upon the outcome of internal function calls as well as incoming and outgoing messages. This means that, in order to fully model communication behaviour, such internal events must be explicitly represented within the model.

Scoping: A model incorporating every atomic process would be cluttered. The behaviour of individual runtime processes is often very simple, with meaningful components consisting of a selection of simple processes. Additionally, library modules can spawn processes for various tasks that



(a) (b)

Figure 2. Global graph (a) and global transition system (b) for the Ping Pong example.

are deliberately hidden from the client of the library's API. Accordingly, it can be preferable to aggregate the overall behaviour of collections of processes into single CFSMs.

Rather than modelling every Erlang process in the CFSM model, we segment the system into *participants* – the key abstract components of the system. The decision as to what forms a participant will depend on the analysis objectives (as discussed in Section 3.1), and there may be some iteration if a particular choice of participants results in an analysis that is too cluttered or vague.

The process of producing models of an Erlang system is as follows:

1) The basic participants of the choreography are identified according to the analysis objectives. These may be entire processes, or source code modules (e.g. library modules) that can influence the behaviour of the process to which they belong.
2) Communication events are identified that constitute the communication between the identified participants. These may be messages sends or receives, function calls, or returns from (synchronous) function calls. These form the alphabet of the CFSMs.
3) For each participant a CFSM is constructed in terms of the communication events.
4) The CFSMs are analysed using ChoSyn to identify communication errors. In the absence of these errors a global graph can be constructed, which provides a succinct specification of the system that can be guaranteed to be free of deadlocks, orphan messages, or unspecified receptions.

The rest of this section illustrates this process in terms of our running ping-pong example.

3.1. Identifying Participants

Identifying the participants of a choreography could be a non-trivial task, as it tends to depend upon both the structure of the system and the goals of the analysis. The complexity of typical applications includes the use of a mixture of libraries, third-party code, off-the-shelf components (the code of which may not be available for analysis). The example in Section 4 makes use of library code that might contain many individual processes internally, but the developer is only able to control their interaction with the library's API in their own source code. If this is the case, identifying participants is accomplished in two steps:

1) Consider each 'principal' process belonging to the source code under analysis as a participant.
2) Identify calls from these processes to external (e.g. library) modules. For each process that calls an external module, add a separate participant representing that external module and repeat this step for the newly identified participants.

This procedure may be iterative. Participants can initially be chosen from well-defined Erlang components such as processes. However, if this fails to fully capture the problem then some participants can be split further (as will be illustrated in Section 4 for our case study). Alternately, if the analysis is obscured by excessive details of irrelevant communication some participants can be merged and only their external interfaces considered.

Ultimately, the choice of participants identifies the *boundaries* across which communication events will take place. This informs the process of identifying the communication *alphabet*, which is the next step in building a model.

3.2. Identifying the Communication Events

The identification of participants enables the identification of the communication events that (could) take place among the participants (i.e. across the previously defined boundaries). This step amounts to determine the set of actions Act of the CFSMs (see Definition 1). Communication events fall under the following categories:

1) A message sent to an identified participant.
2) The receipt of a message from a participant.
3) A call to a library / third-party participant that could affect the interactions with or among participants.

4) A return of control from synchronous calls to library / third-party participants.

Where available, this can be ascertained from the source code. In Erlang this would be achieved by inspecting any `send` or `receive` statements. If the source code is not available, the rules can be derived from any available API documentation, or by inspecting the interaction with other processes in the system.

In the ping-pong example, the `pong` function is defined with a single receive statement that either receives the atom `finished` and terminates, or receives a pair containing the atom `ping` and a process ID of the pinger, which it responds to with a `pong`. Accordingly, its alphabet is: $\{pingpong?ping, pingpong?finished, pongping!pong\}$
The pinger's communications are the complement of the pong process, since it initiates either the ping-pong transaction, or sends `finished` to terminate both processes. Accordingly, its alphabet is: $\{pingpong!ping, pingpong!finished, pongping?pong\}$.
Whereas this example is (deliberately) simple[4], using only the first two types of communication events, our case study will involve examples of other communication events that include function calls and returns.

As with the choice of participants, the process of identifying communication events can be iterative. If the communications are not sufficiently detailed the CFSM system can be non-deterministic or allow patterns of communication that are excluded by the real system. Conversely, excessive detail may exclude some paths that can be followed, or it may simply require intractably large state machines to model the multiple variations of essentially the same message.

3.3. Modelling Communications as CFSMs

The choice of communication events (Act) in the previous step determines the alphabet of the CFSMs models produced for the individual participants. To describe how the state machines are constructed, we adopt the notation defined in Definition 1.

Conceptually, the process of constructing a state machine for a participant M_p, a state machine $(Q, q_0, \Sigma, \rightarrow)$ can proceed as follows:

1) The alphabet Σ is defined as the set of communication events Act_p
2) Identify a set of states Q (and the initial state $q_0 \in Q$). Each state represents the set of possible sequences of communications that can occur at a given point during the execution of a participant.
3) For each state $q \in Q$, identify the set of communications $C \subseteq \Sigma$ that can occur from that state.

4. The ping-pong example only contains two communicating components: The pinger and the ponger processes, with the boundary between them being the sending and receiving of messages into their respective mailboxes. This captures the essential communication events that are perturbed by the faulty examples in Section 2.2, but it abstracts away the implementation details of their spawning, and their debugging outputs.

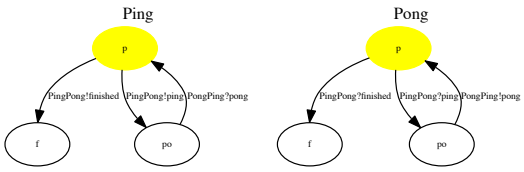


Figure 4. CFSMs corresponding to the Erlang code in Figure 1.

4) For each communication $c \in C$, a destination state $q' \in Q_p$ is identified, and a transition $q \xrightarrow{c} q'$ is added to the set of transitions $\rightarrow_p$.

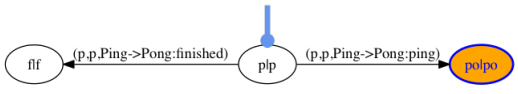
Depending on the complexity of the process, and the information available about its behaviour, identifying the states can rely on a degree of intuition and prior knowledge. If available, the source code can be inspected to follow the control-flow between source code statements that correspond to communication events. It is also possible to resort to prior knowledge – API documentation, or general sequencing rules (e.g. that a synchronous method call must be followed by a return of control from that call).

The ping-pong example is straightforward enough for both participants to be derived entirely from the source code (in Figure 1). Process `pong` is spawned on line 27 with a function call to the pong function (line 16). The CFSMs of are in Figure 4; for `pong`, state p and f are the initial and the terminal state respectively, while state po represents the state where message `ping` has been received and message `pong` is about to be sent (for `ping` – spawned on line 28 with a call to the `ping` function – the CFSM is similar).

3.4. Diagnosing errors

The CFSMs are now used to first generate and analyse the transition system and the corresponding global graph [12]. In the transition system of our running example (cf. Figure 2), each state represents a unique combination of states of `ping` and `pong`. As shown below, this can be used to highlight communication errors.

A deadlock state shows up clearly in the transition system as a state with no outgoing edges, but where some participants could still interact. These are highlighted in orange by the tool. For example, the transition system for the modified version of ping-pong from Section 2.2 computed by ChoSyn is



where in the state $po|po$ there is a deadlock due to the wrong message `poong` in the buffer of `ping`.

Orphan messages are illustrated by the example with two instances of the ping client. One ping process can send `finished`, which then terminates the pong process. Any messages from the other ping process cannot be received.

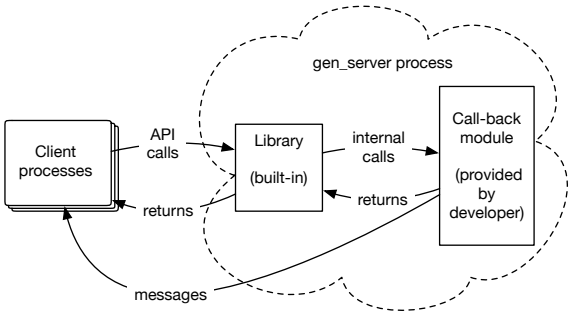


Figure 5. Key participants in a `gen_server` implementation

The tool identifies the two deadlock states (one for each of the pingers) but it also identifies the intrinsic conflict between the processes with this output:

```
gmc: Branching Property:
[Rp [p,p,p] (p,p,0,1,finished) (p,p,2,1,finished)
     Not unique sender  No choice awareness,
 Rp [p,p,p] (p,p,0,1,finished) (p,p,2,1,ping)
     Not unique sender  No choice awareness,
 Rp [p,p,p] (p,p,0,1,ping) (p,p,2,1,finished)
     Not unique sender  No choice awareness,
 Rp [p,p,p] (p,p,0,1,ping) (p,p,2,1,ping) Not unique sender ]
```

Finally, the issue of unspecified reception is illustrated by removing the clause in the pong process that would receive a `finished` message. This is identified by the tool through its *branching representibility* requirement:

```
gmc: Branching representability: [Bp 0 "f",Bp 0 "p"]
...
chosyn: Is machine Ping representable? false
```

This error declares that there is no transition in the transition system that represents the successful transfer of the `finished` message.

4. Case Study: Erlang/OTP `gen_server`

We now apply this process to `gen_server`, a component of the Erlang OTP standard library. We refer to the API documentation[5] as a guide. Notably, `gen_server` is used in most substantive Erlang projects[6] and provides a library of basic functionality to support the development of client/server applications. The basic structure is shown in Figure 5. The library contains an API that enables client processes to start new `gen_server` instances (for which it returns a process ID). Importantly for us, client/server interactions are mediated by appropriate functions in the `gen_server` API, passing the process ID of the server in question as an argument. This is an example of an application governed by combining the use of a library with some specific code. In fact, developers can customise the behaviour of a `gen_server` by providing their own *call-back* module, which contains functions with specific signatures that are invoked by the `gen_server` library.

5. See http://www.erlang.org/doc/man/gen_server.html and http://www.erlang.org/doc/design_principles/gen_server_concepts.html

6. At the time of writing there were 31,593 uses of `gen_server` in Erlang GitHub projects (there are 11,613 Erlang repositories).

```
=============STRUCTURE=============
gen_server module   Callback module
-----------------   ---------------
gen_server:start -> handle_init/1
gen_server:call  -> handle_call/2
gen_server:cast  -> handle_cast/2
-                -> terminate/2
============LIBRARY API=============
start(Module, Args, Opts) -> Result
   Result = {ok,Pid} | {error,Error}
Creates a stand-alone server process.
------------------------------------
call(ServerRef, Request) -> Reply
   Reply = term()
Makes synchronous call to server.
------------------------------------
cast(ServerRef, Request) -> ok
Makes asynchronous call to server.
======CALLBACK MODULE FUNCTIONS=====
[Called from gen_server:start]
Module:init(Args) -> Result
   Result = {ok,State}
   | {stop,Reason}
------------------------------------
[Called from gen_server:call]
Module:handle_call(Request, From,
   State) -> Result
   Result = {reply,Reply,NewState}
      | {noreply,NewState}
      | {stop,Reason,NewState}
If the functions returns {noreply,NewState},
the gen_server will continue executing with NewState.
Any reply to the client must be given explicitly (by
calling the gen_server:reply/2 function).
------------------------------------
[Called from gen_server:cast]
Module:handle_cast(Request, State) -> Result
   Result = {noreply,NewState}
     | {stop,Reason,NewState}
```

Figure 6. Curtailed API docs for `gen_server`.

We focus on the core functionalities of `gen_server`: Starting a new server, stopping the server, handling synchronous / asynchronous calls[7]. The goal of our analysis is to establish whether, given the API description, any of the communication errors described in Section 2.2 can arise.

4.1. Building the CFSMs

Identifying the participants: As discussed in Section 3.1, we have to identify (1) the main participants and (2) the interactions with the library.

We start by designating two (obvious) main participants: The server `S` and the client `C`. The specific `gen_server` behaviour is defined in a call-back module. This requires to reconsider the participants since `C` does not send messages directly to `S`, rather it invokes the `gen_server` library. We therefore create a participant `L` to represent the library.
Identifying the communication events: As detailed in Section 3.2, we identify (1) the messages that each participant possibly sends to other participants, (2) those received from other participants, (3) function calls to other participants, and (4) returns of these calls.

We start with the `gen_server` API, of which a reduced excerpt is shown in Figure 6; it also contains a small

7. In Erlang's jargon, a call is synchronous when a return message is expected and asynchronous otherwise.

$$
\begin{array}{lcl}
Act_C & = & \{CL!start, CL!call, CL!cast, LC?ok, LC?error, LC?reply, SC?reply\} \\
Act_L & = & \{CL?start, CL?call, CL?cast, LC!ok, LC!error, LC!reply\} \\
 & \cup & \{LS!handleInit, LS!handleCall, LS!handleCast, LS!terminate, SL?ok, SL?error, SL?reply, SL?noreply, SL?stop\} \\
Act_S & = & \{LS?handleInit, LS?handleCall, LS?handleCast, LS?terminate, SL!ok, SL!error, SL!reply, SL!noreply, SL!stop, SC!reply\}
\end{array}
$$

Figure 7. Events for C, L, and S

specification for what is expected from the call-back module, thus detailing all of the expected interactions between S and both L and C. By inspection we can see what interactions are expected between client C and the library L: Client C can call the start, call, or cast functions, and L can respond with ok, error, or reply. Furthermore, from the call-back module API (which captures the behaviour of S), we note that S can also send direct reply messages to C without going through L. From this we derive the communication events Act_C for C as shown in Figure 4.1. Participant L takes messages from C and forwards them to the appropriate handler functions in S; this yields the first set of the union constituting Act_L in Figure 4.1; the second set in the union yields the events for the subsequent interactions between L and S. Finally, set Act_S in Figure 4.1 corresponds to the responses of S to L and those directly sent to C.

Building the CFSMs: The state machines produced for the participants, using the actions defined previously, are shown in Figure 8. Individual machines are reasonably straightforward to interpret; sequences of ingoing and outgoing interactions correspond to paths through the machine. Such machines are the ones given in input to ChoSyn for our analysis.

4.2. Identifying Communication Errors

Once the CFSMs are identified, ChoSyn computes the transition system reported in Figure 9. The shaded states are those highlighted by ChoSyn and are either deadlock states or states that instigate the deadlocks (in lighter shade).

Deadlock 1 (S does not reply) – The problem occurs when, in state $calling|calling|calling$, S has returned a noreply. This is allowed by the API (see Figure 6); indeed, the documentation of handleCall does state that any reply to C "must be given explicitly" using a separate API function. However, this language is ambiguous because it does not imply that a reply must be sent. However, in the event that no direct reply is sent by the server, and the handleCall function returns noreply, C will remain blocked waiting for a response indefinitely.

Deadlock 2 (direct reply from S to C, followed by a stop) – In state $running|calling|calling$ S has returned a stop after sending a reply to C. This causes C to continue as though S was still running. However, the stop reply to L will cause it to terminate S. Client C will subsequently attempt a call operation, thinking that the server is still running, but it will never be consumed or receive a response.

Deadlock 3 (server stopping upon receipt of a cast or a call message) – The problem occurs at the point where C has sent a cast or call message to L and L has handled it by invoking handle_call or handle_cast in S (states $running|calling|calling$ or $running|casting|casting$ respectively). If, at this point, S decides to stop, it notifies the library accordingly but neither L nor S notify C. This leads to the erroneous state ($running|start|start$). Client C is still running as normal and is assuming that L and S are still able to receive messages. However, this will not be the case since S has terminated. As a result, cast messages would be orphaned, and the client will deadlock on a call message.

4.3. Fixing the deadlocks

These deadlocks can all be avoided by introducing three design requirements. Deadlock 1 can be avoided by ensuring that S either returns reply to L or sends a direct reply to C and returns noreply to L. Deadlock 2 can be avoided by ensuring that S is required to never return stop to L after sending a direct reply to C. Finally, deadlock 3 can be avoided by introducing the requirement that S never responds to a cast by stopping, and that any decision to stop in response to a call is communicated to C.

With these changes in place, ChoSyn confirms that the resulting system can be guaranteed to be deadlock free. It is also able to produce a global graph, capturing the possible sequences of interactions between the components that are *guaranteed to be free of deadlocks, orphan messages, and unspecified receptions*. This is shown in the Appendix.

5. Conclusions

We have applied a tool-supported methodology based on choreographies for the analysis of Erlang's gen_server. Our approach consisted in encoding the participants as CFSMs, and thus applying the CFSM-based choreography-analysis technique proposed in [5] using the ChoSyn [13] tool. The analysis highlighted possible errors that compromise the execution. We have also shown how ChoSyn helps in identifying design guidelines that, when adopted, guarantee the absence of these errors.

As discussed in Sections 3 and 4, the automatic extraction of models may be problematic. For instance, the extraction of models for gen_server can hardly be completely automated since the library specifications are informal. Our future work will focus on applying model-inference techniques building upon our work on inferring finite state machines from Erlang code [14], and on inferring Extended Finite State Machines that are able to take account of the underlying data state [15]. Automatic inference of models will mitigate a disadvantage of our approach, which as is the case with most model-based analysis techniques, has two main repercussions. On the one hand, manual extraction of

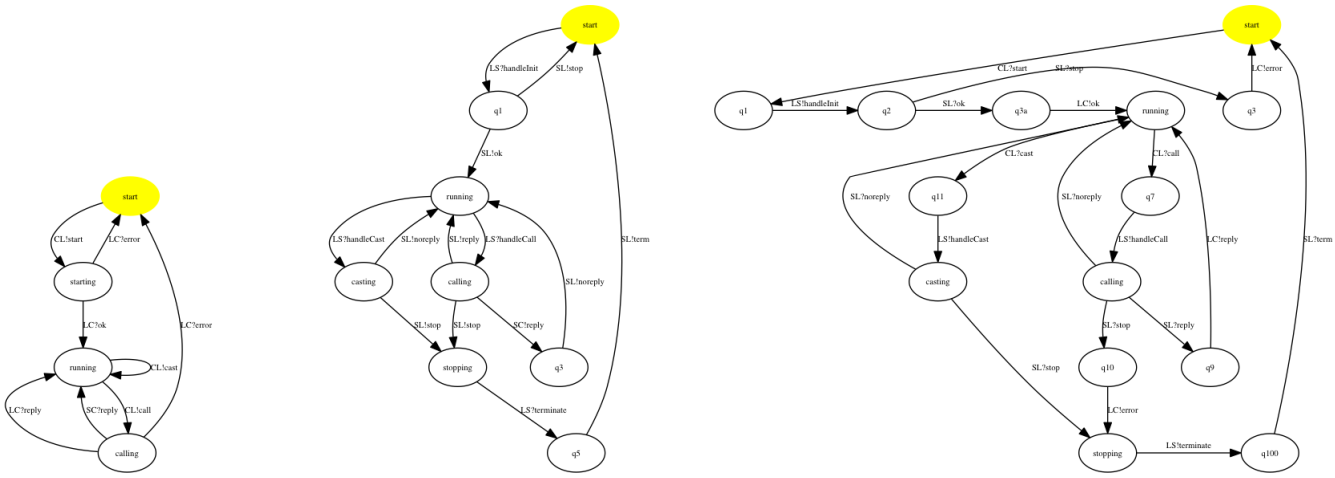


Figure 8. The CFSMs of `gen_server`: `C` (leftmost), `S` (middle), and `L` (rightmost)

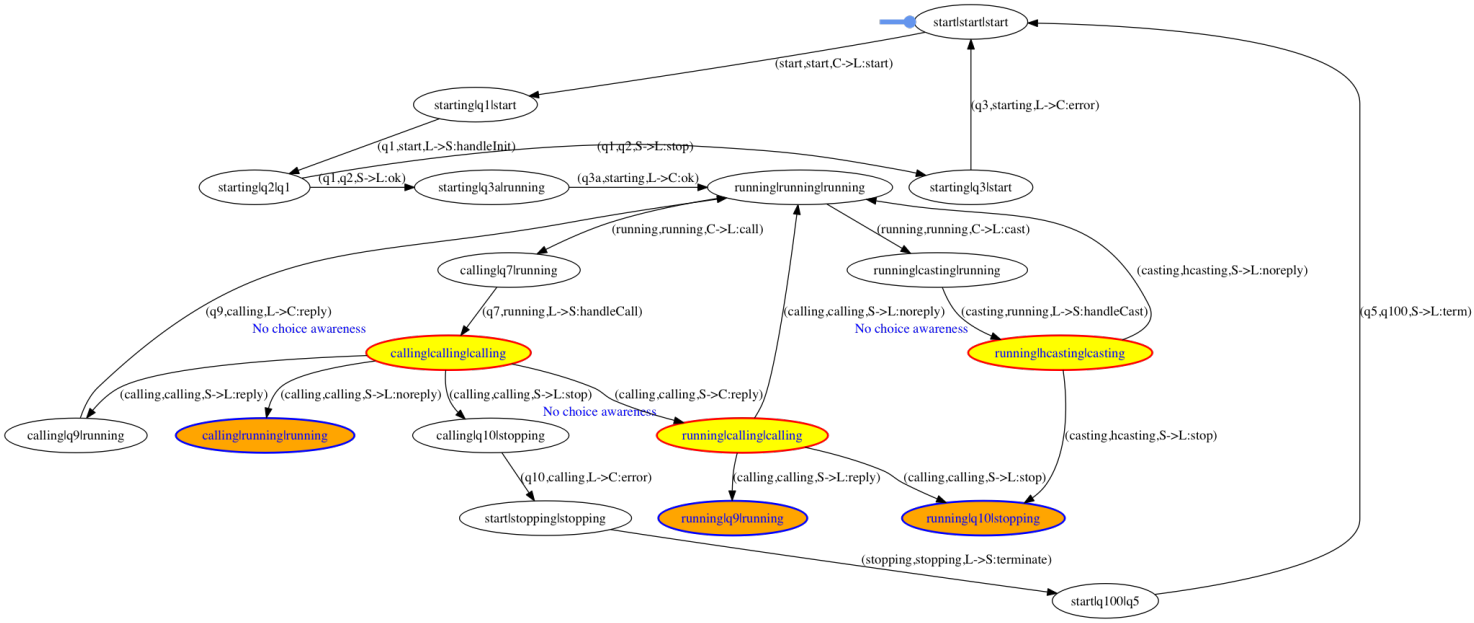


Figure 9. The transition system of `gen_server` computed by ChoSyn

models makes it sometimes difficult to provide convincing arguments that the models faithfully reflect the actual behaviour of the system under study. On the other hand, the manual generation of models is a potential obstacle for the widespread use of a verification methodology.

References

[1] "SAVARA Testable Architecture," http://www.jboss.org/savara.

[2] W. W. W. Consortium, "Web services choreography description language version 1.0," http://www.w3.org/TR/ws-cdl-10/, 2005.

[3] S. Basu, T. Bultan, and M. Ouederni, "Deciding choreography realizability," in *POPL12*, 2012.

[4] S. McIlvenna, M. Dumas, and M. T. Wynn, "Synthesis of orchestrators from service choreographies," in *APCCM*, 2009.

[5] J. Lange, E. Tuosto, and N. Yoshida, "From communicating machines to graphical choreographies," in *POPL15*, 2015.

[6] D. Brand and P. Zafiropulo, "On communicating finite-state machines," *JACM*, vol. 30, no. 2, 1983.

[7] R. Carlsson, "Towards a deadlock analysis for erlang programs," *Master's Thesis in Computing Science, Uppsala University*, 1997.

[8] M. Christakis and K. F. Sagonas, "Static detection of race conditions in erlang," in *PADL 2010*, ser. LNCS, vol. 5937, 2010.

[9] ——, "Detection of asynchronous message passing errors using static analysis," in *PADL 2011*, ser. LNCS, vol. 6539, 2011.

[10] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger, "Finding race conditions in erlang with quickcheck and pulse," *ACM Sigplan Notices*, vol. 44, no. 9, 2009.

[11] A. Gotovos, M. Christakis, and K. Sagonas, "Test-driven development of concurrent programs using concuerror," in *Proc. 10th ACM SIGPLAN workshop on Erlang*, 2011, pp. 51–61.

[12] P. Deniélou and N. Yoshida, "Multiparty session types meet communicating automata," in *ESOP*, 2012.

[13] "ChoSyn," https://bitbucket.org/emlio_tuosto/gmc-synthesis-v0.2/.

[14] R. Taylor, K. Bogdanov, and J. Derrick, "Automatic inference of erlang module behaviour," in *IFM*, 2013, pp. 253–267.

[15] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," *Empirical Software Engineering*, 2015.

Appendix

The Global Graph of the amended `gen_server`, guaranteed to be free of communication errors