

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—We report on the analysis of `gen_server`, a popular Erlang library to build client-server applications. Our analysis uses a tool based on choreographic models. We discuss how, once the library has been modelled in terms of communicating finite state machines, 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 WhatsApp¹, Facebook², and Basho Riak³ – a distributed database that was recently chosen to underpin the UK National Health Service IT infrastructure.

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.

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 finite state machines (CFSMs) [5].
- A choreography-based analysis to detect communication faults or to guarantee their absence.

1. <http://goo.gl/6ldR5z>

2. <https://goo.gl/Z1tpgA>

3. <http://goo.gl/GK9II3>

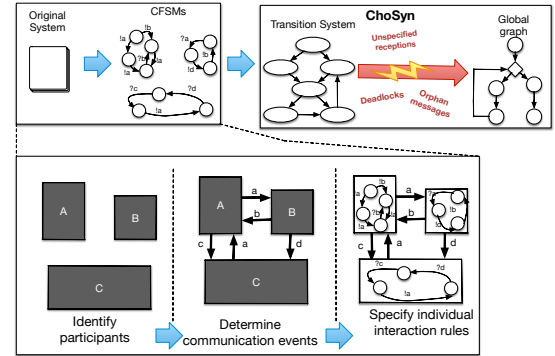


Figure 1. The process of extracting and analysing choreographies. The top-half shows the essential, high-level process. The bottom half provides an outline for our process of extracting CFSMs.

- 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.

Our methodology consists of the two phases shown in Figure 1. The first phase is a manual process⁴ and may requires some ingenuity; in the second phase a completely automatic analysis is performed using the ChoSyn tool.

We consider Erlang in this paper, however the methodology is suitable to any language featuring the basic characteristics of the actor-model [6] such as Go and Scala.

2. Background

2.1. Erlang by Examples

Figure 2 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.

4. We are currently working on its (partial) automatization.

```

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 2. Erlang ping-pong message-passing example.

Erlang has an 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 2 spawn processes that execute the `pong` 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 are bound to values by ‘pattern-matching’. For example, `ping` is defined with two clauses and takes two parameters. If the first parameter matches the 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 `Pong_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*. We use the characterisation of such errors given in [7], building on a similar characterisation in [5], 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 2.

Deadlock If the programmer types

```
Ping_PID ! poong,
```

instead of line 22 in Figure 2, 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 pings 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 consumed preventing the program from ever terminating.

Crucially such faults are impossible to detect by looking at processes in isolation. They *emerge* from interactions, which demands a macroscopic overview of the system.

Attempts to detect specific errors in Erlang programs have been made, albeit with limited success. Modelling asynchronous Erlang programs as CCS or CSP, which presume synchronous communication, is impractical due to the explicit modelling of associated buffer and carrier processes [8]. Several static source code analysis approaches have been devised to detect deadlocks [9], [10], however these can be highly inaccurate. More recently, testing approaches have emerged [11], [12], however these can only reveal the presence of error not guarantee their absence.

2.3. Choreographies

Recently advocated as a promising basis for modelling distributed behaviour (see e.g., [1], [2], [3], [4]), *choreographies* model distributed interactions of several *participants* from a *global point of view* (cf. [2]). We formalise the participants as *communicating finite state machines* [5]:

Definition 1 (CFSM) Fix sets \mathcal{P} of participants (ranged over by p, q) and Σ of messages, and let Act be a set of actions, where an action $\ell \in Act$ is a sending (resp. receiving) action $\ell = pq!a$ (resp. $\ell = qp?a$), namely p writes (resp. consumes) message $a \in \Sigma$ in (resp. from) the buffer to (resp. from) q . A communicating finite state

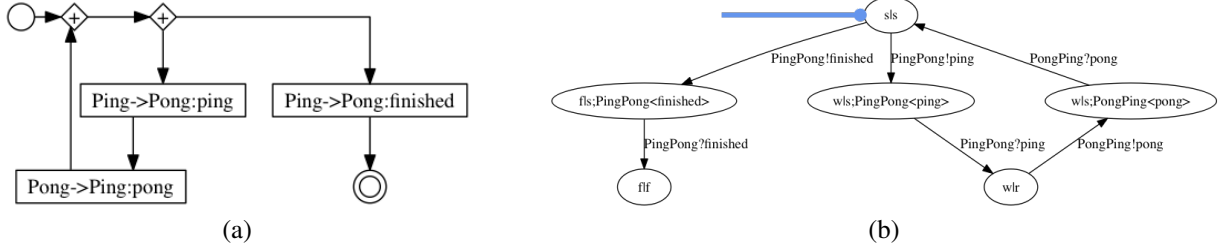


Figure 3. Global graph (a) and global transition system (b) for the Ping Pong example.

machine (CFSM) 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
- $\rightarrow \subseteq Q \times \text{Act} \times Q$ is a set of transitions; we write $q \xrightarrow{\ell} q'$ for $(q, \ell, q') \in \rightarrow$.

Given a CFSM $M_p = (Q_p, q_{0p}, \Sigma, \rightarrow_p)$ for each $p \in \mathcal{P}$, the tuple $\mathbf{S} := (M_p)_{p \in \mathcal{P}}$ 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 $\mathbf{S} = (M_p)_{p \in \mathcal{P}}$ be a communicating system. A configuration of \mathbf{S} is a pair $s = \langle \mathbf{q}; \mathbf{w} \rangle$ where $\mathbf{q} = (q_p)_{p \in \mathcal{P}}$ with $q_p \in Q_p$ and where $\mathbf{w} = (w_{pq})_{pq \in C}$ with $w_{pq} \in \Sigma^*$; \mathbf{q} is initial when q_p is the initial state of the corresponding CFSM and all buffers are empty.

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

- | | |
|---|---|
| 1. $\ell = \text{sr}!a$ and $(q_s, \ell, q'_s) \in \delta_s$
and | 2. $\ell = \text{sr}?a$ and $(q_r, \ell, q'_r) \in \delta_r$
and |
| a. $q'_p = q_p$ for all $p \neq s$ | a. $q'_p = q_p$ for all $p \neq r$ |
| b. and $w'_{sr} = w_{sr}.a$ and
$w'_{pq} = w_{pq}$ for all
$pq \neq sr$ | b. and $w_{sr} = a.w'_{sr}$ and
$w'_{pq} = w_{pq}$ for all
$pq \neq sr$. |

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 \mathbf{S} is $\text{RS}(\mathbf{S}) = \{s \mid s_0 \rightarrow^* s\}$. The k -reachability set of \mathbf{S} is the largest subset $\text{RS}_k(\mathbf{S})$ of $\text{RS}(\mathbf{S})$ 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 . Note that $\text{RS}_k(\mathbf{S})$ is finite.

Definition 3 (Synchronous transition system) Let $N := \{\mathbf{q} \mid \langle \mathbf{q}; \varepsilon \rangle \in \text{RS}_1(\mathbf{S})\}$ be the set of stable configurations of a system $\mathbf{S} = (M_p)_{p \in \mathcal{P}}$; N is ranged over by n . Define

$$\hat{\delta} := \{(n, e, n') \mid (n; \varepsilon) \xrightarrow{\text{sr}!a} \text{sr}?a (n'; \varepsilon) \wedge e = (n[s], n[r], s \rightarrow r:a)\}$$

where $E := \{e \mid \exists n, n' \in N : (n, e, n') \in \hat{\delta}\} \subseteq \mathcal{E}$. The synchronous transition system of \mathbf{S} is $\text{TS}(\mathbf{S}) = (N, \mathbf{q}_0, \hat{\delta})$.

The global graphs of a communicating system \mathbf{S} is a “work-flow” presentation of $\text{TS}(\mathbf{S})$ pinpointing distributed choices and causal dependencies among the interactions of \mathbf{S} . In [7] a condition on $\text{TS}(\mathbf{S})$ is given to guarantee the absence of communication errors. In fact, the analysis of \mathbf{S} relies on mismatches between $\text{TS}(\mathbf{S})$ and its asynchronous behaviour: When no errors arise, the global graph of \mathbf{S} is a faithful yet compact representation of its behaviour; otherwise, $\text{RS}_1(\mathbf{S})$ singles out error traces. Both models are automatically computed by ChoSyn [13], the tool we will use here for the analysis of `gen_server`. For instance, Figure 3(a) gives the global graph of the ping-pong protocol whose asynchronous behaviour is in Figure 3(b) (more details are given in Section 3): Following the path from the initial node, the ping process chooses whether to send `ping` or `finished`; in the former case the protocol loops after the participants exchange the `pong` message.

3. Modelling and Analysis

The process of modelling Erlang programs as choreographies is illustrated in the top half of Figure 1. There are two basic phases: (1) build CFSMs from the program under analysis and (2) apply the global graph construction and analysis (cf. [7]) to identify communication errors.

Ideally, automatic model extraction approaches should be applied in phase (1); however, we note that it is not always possible 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 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), and there may be some iteration if a particular choice of participants results in an analysis that is too cluttered or vague.

Producing models of an Erlang system goes as follows:

- 1) The participants of the choreography are identified according to the analysis objectives. These may be entire processes, or code (e.g. library modules) that can influence the behaviour of the process.
- 2) Communication events are identified that constitute the communication among participants. These may be explicit 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 the global graph constructed by the tool provides a succinct specification of the system that can be guaranteed to be free of deadlocks, orphan messages, or unspecified receptions. In presence of errors, the global graph reflects only the “good” executions while disregarding the error traces.

We now illustrate steps 1-4 on the ping-pong example.

1. IDENTIFYING PARTICIPANTS of a choreography could be a non-trivial task, as it often depends 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 analysis in Section 4 involves library code that might contain many individual processes internally, but the developer is only able to control their interaction through the library’s API.

Participants are therefore identified 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.

2. IDENTIFYING THE COMMUNICATION EVENTS amounts to determine the set of actions *Act* of the CFSMs (see Definition 1) once participants have been determined (i.e. across the previously defined boundaries). 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.

Events can be ascertained from the source code; in Erlang this would be achieved by inspecting any `send` or `receive` statements. If source code is unavailable, events 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 ping process, which it responds to with a `pong`. Accordingly, its alphabet is: $\{\text{pingpong?ping}, \text{pingpong?finished}, \text{pongping!pong}\}$. The communications of the ping process are the complement of the pong process, since it initiates either the ping-pong exchange, or sends `finished` to terminate both processes. Accordingly, its alphabet is: $\{\text{pingpong!ping}, \text{pingpong!finished}, \text{pongping?pong}\}$. Whereas this example is (deliberately) simple⁵, 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 actual executions, or it may simply require intractably large state machines to model the multiple variations of essentially the same message.

3. MODELLING COMMUNICATIONS AS CFSMS is done following 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:

5. The ping-pong example only contains two communicating components: The ping and the pong 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 2.

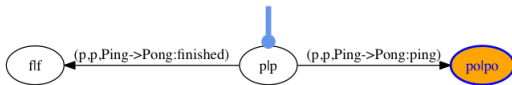
- 1) The alphabet Σ is defined as the set of communication events Act_p
- 2) Identify a set of states Q (and the initial state $q_0 \in Q$). Each state represents the set of possible sequences of communications that can occur at a given point during the execution of a participant.
- 3) For each state $q \in Q$, identify the set of communications $C \subseteq \Sigma$ that can occur from that state.
- 4) 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 2). 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).

4. DIAGNOSING ERRORS happens through the generation and analysis of the transition system (and the corresponding global graph [14]) derived from the CFSMs. In the transition system of our running example (cf. Figure 3), 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 synchronous 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`. The analyst can use global graphs to identify where changes have to be made when communication errors occur; for instance, the above transition system yields a global graph (different from the

expected one of Figure 3) where the `pong` is not exchanged; this suggests that either the send of the receive operation fail. The actual error on the send operation is highlighted by ChoSyn on the asynchronous transition system of ping-pong (which is not reported here for the lack of space).

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. The tool identifies the two deadlock states (one for each of the ping processes) 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 selector No choice awareness,
Rp [p,p,p] (p,p,0,1,finished) (p,p,2,1,ping)
  Not unique selector No choice awareness,
Rp [p,p,p] (p,p,0,1,ping) (p,p,2,1,finished)
  Not unique selector No choice awareness,
Rp [p,p,p] (p,p,0,1,ping) (p,p,2,1,ping) Not unique selector ]
```

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 message of ChoSyn:

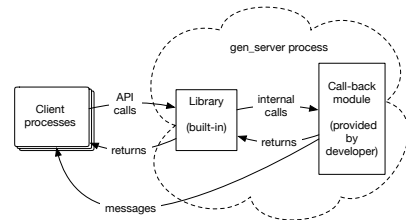
```
gmc: Branching representability: [Bp 0 "f",Bp 0 "p"]
...
chosyn: Is machine Ping representable? false
```

which flags the violation of the *branching representability* requirement of [7] imposing that in each distributed choice is determined by a unique machine (unique selector) while any other participant is either made aware of the chosen branch or does not participate in the choice (choices awareness).

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 methodology to `gen_server`, a component of the Erlang OTP standard library. We refer to the API documentation⁶ as a guide. Notably, `gen_server` is used in most substantive Erlang projects⁷ and provides a library of basic functionality for the development of client/server applications. The basic structure of `gen_server` can be depicted as



The library contains an API that enables client processes to start new `gen_server` instances (for which it returns

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

7. At the time of writing there were 60,155 uses of `gen_server` in Erlang GitHub projects.

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.

We focus on the core functionalities of `gen_server`: Starting a new server, stopping the server, handling synchronous / asynchronous calls⁸. 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 participants: As discussed in Section 3, 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 communication events: As detailed above, 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 as described at http://www.erlang.org/doc/man/gen_server.html, where it is also given a small specification of what expected from the call-back module, thus detailing all of the expected interactions between *S* and both *L* and *C*. The interactions expected between client *C* and the library *L* are: 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 5. 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 5; the second set in the union yields the events for the subsequent interactions between *L* and *S*. Finally, set Act_S in Figure 5 corresponds to the responses of *S* to *L* and those directly sent to *C*.

Building CFSMs: Figure 6 shows the machines produced for the participants using the actions defined previously. 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.

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

4.2. Identifying Communication Errors

Once the CFSMs are identified, ChoSyn computes the transition system reported in Figure 7 (according to Definition 2). 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; 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: *S* stopping upon receipt of a cast or a call message

The problem occurs when a cast or call message from *C* is handled by *L* invoking `handle_call` or `handle_cast` in *S* (states *running|calling|calling* or *running|casting|casting* respectively). If now *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.

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

Figure 5. Events for C, L, and S

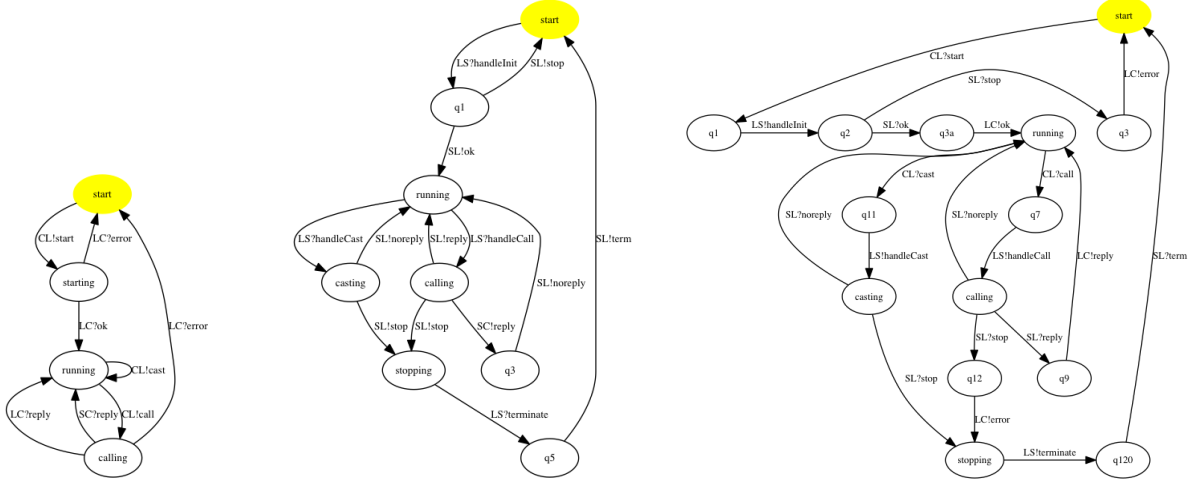
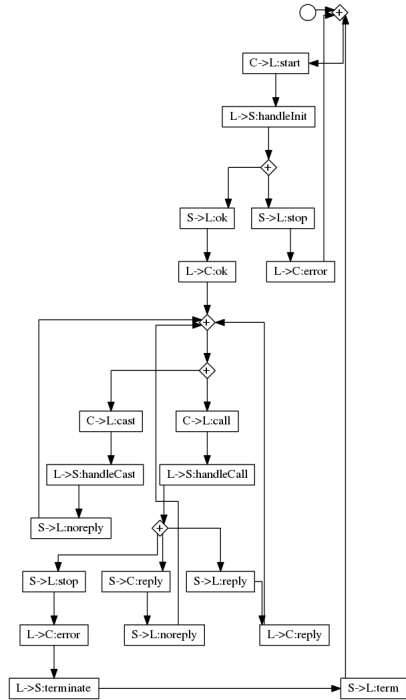


Figure 6. The CFSMs of `gen_server`: C (leftmost), S (middle), and L (rightmost)

The tool also produces the following global graph



capturing the possible sequences of interactions between the components that are *guaranteed to be free of deadlocks, orphan messages, and unspecified receptions*.

5. Conclusions

We applied a tool-supported methodology based on choreographies for the analysis of Erlang's `gen_server`.

We first modelled participants as CFSMs, and then applied the analysis technique formalised in [7] using the ChoSyn [13] tool. The analysis highlighted possible communication errors. We have also shown how ChoSyn helps in refining design guidelines to avoid such errors.

The coordination of distributed interactions has been studied in many contexts. In OO-programming *synchronisers* are proposed in [15], [16] as a mechanism to monitor execution while controlling the access to objects. Actor-based systems have been verified using model-based techniques [17], [18], [19], static analysis [20], and test-driven analysis [21], [22], [11], [12]. An added value of our methodology with respect to such approaches is the use of global graphs to support the analysis. In [23] an extension of LTL is proposed to specify views of *social states* in agent systems reducing verification to validity or satisfiability problems in such logic. The properties of interest in [23] (e.g., compliance with *social contracts*) are not concerned with communication errors (albeit reminiscent of choreographies, social states do not explicitly model interaction).

The automatic extraction of models may be problematic (cf. Sections 3 and 4). For instance, the extraction of models for `gen_server` can hardly be fully 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 [24], and on inferring *extended finite state machines* that are able to take account of the underlying data state [25]. Automatic inference of models will mitigate a disadvantage of our approach yielding two main repercussions (as typical in most model-based analysis techniques). On the one hand, manual extraction of models could make it difficult to pro-

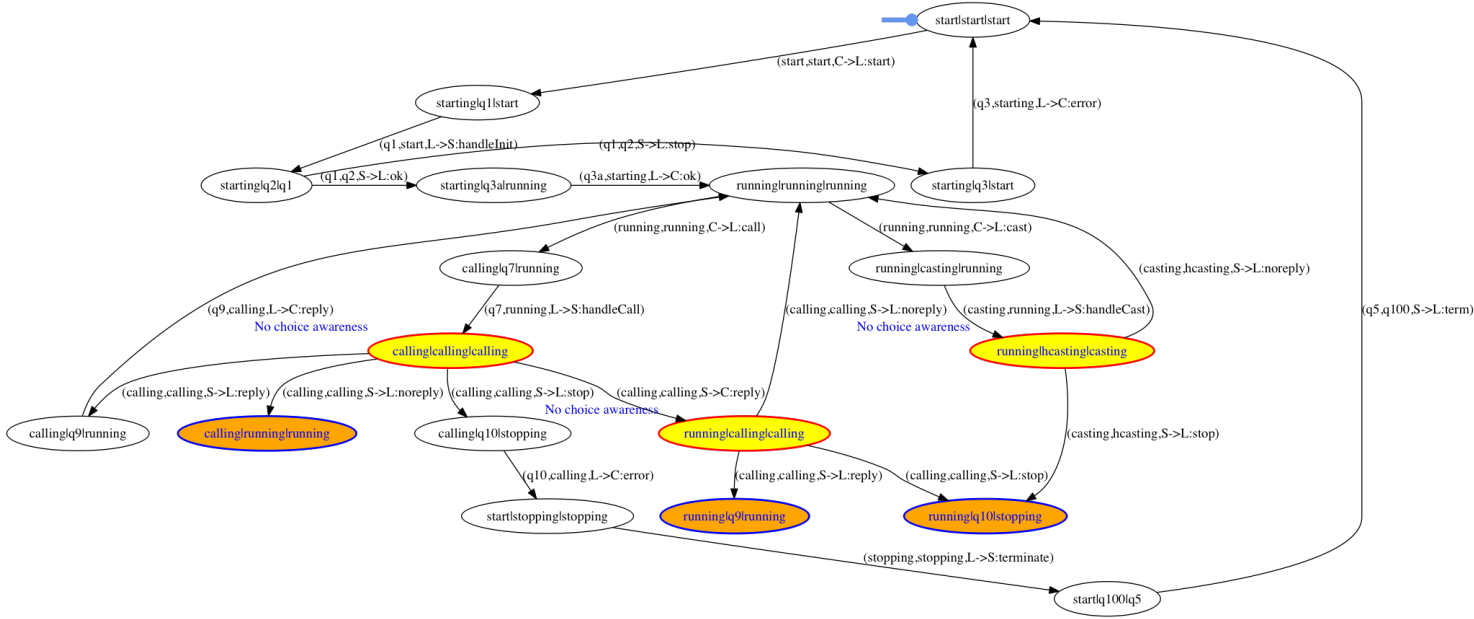


Figure 7. The transition system of `gen_server` computed by ChoSyn

vide 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.

Acknowledgements

The authors thanks the reviewers of 4PAD for their valuable comments.

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] D. Brand and P. Zafiropulo, “On communicating finite-state machines,” *JACM*, vol. 30, no. 2, 1983.
- [6] G. Agha, “Actors: a model of concurrent computation in distributed systems,” Ph.D. dissertation, MIT, 1986.
- [7] J. Lange, E. Tuosto, and N. Yoshida, “From communicating machines to graphical choreographies,” in *POPL15*, 2015.
- [8] R. Carlsson, “Towards a deadlock analysis for erlang programs,” *Master’s Thesis in Computing Science, Uppsala University*, 1997.
- [9] M. Christakis and K. F. Sagonas, “Static detection of race conditions in erlang,” in *PADL 2010*, ser. LNCS, vol. 5937, 2010.
- [10] —, “Detection of asynchronous message passing errors using static analysis,” in *PADL 2011*, ser. LNCS, vol. 6539, 2011.
- [11] 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.
- [12] 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.
- [13] “ChoSyn,” https://bitbucket.org/emlio_tuosto/gmc-synthesis-v0.2/.
- [14] P. Deniélou and N. Yoshida, “Multiparty session types meet communicating automata,” in *ESOP*, 2012.
- [15] S. Frølund and G. Agha, “A language framework for multi-object coordination,” in *In Proceedings of ECOOP*. Springer Verlag, 1993.
- [16] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa, “Abstracting object interactions using composition filters,” in *ECOOP’93*, ser. LNCS, vol. 791. Springer, 1993.
- [17] M. Sirjani, F. S. D. Boer, and A. Movaghgar, “Modular verification of a component-based actor language,” *JOURNAL OF UNIVERSAL COMPUTER SCIENCE*, vol. 11, no. 10, pp. 1695–1717, 2005.
- [18] F. Huch, “Verification of erlang programs using abstract interpretation and model checking,” in *ICFP ’99*. ACM, 1999.
- [19] M. Neuhäuser and T. Noll, “Abstraction and model checking of core erlang programs in maude,” *ENTCS*, vol. 176, no. 4, 2007.
- [20] E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martín-Martín, G. Puebla, and G. Román-Díez, “SACO: Static Analyzer for Concurrent Objects,” in *TACAS*, ser. LNCS. Springer, 2014, vol. 8413.
- [21] S. Tasharofi, R. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha, “TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs,” in *FORTE*, ser. LNCS. Springer, 2012, vol. 7273.
- [22] K. Sen and G. Agha, “Automated systematic testing of open distributed programs,” in *FASE*, ser. LNCS. Springer, 2006, vol. 3922.
- [23] L. Giordano, A. Martelli, and C. Schwind, “Specifying and verifying interaction protocols in a temporal action logic,” *J. of Applied Logic*, vol. 5, no. 2, 2007.
- [24] R. Taylor, K. Bogdanov, and J. Derrick, “Automatic inference of erlang module behaviour,” in *IFM*, 2013, pp. 253–267.
- [25] N. Walkinshaw, R. Taylor, and J. Derrick, “Inferring extended finite state machine models from software executions,” *Empirical Software Engineering*, 2015.