Erlang on TOAST: Generating Erlang Stubs with Inline TOAST Monitors*

Jonah Pears
University of Kent
Canterbury, United Kingdom
jjp38@kent.ac.uk

Laura Bocchi
University of Kent
Canterbury, United Kingdom
l.bocchi@kent.ac.uk

Raymond Hu
Queen Mary University of London
London, United Kingdom
r.hu@qmul.ac.uk

Abstract

Implementing concurrent systems based on asynchronous communications is intrinsically complex. In this work, we consider the formal framework TOAST for *timed* asynchronous interactions featuring *mixed-choice* states. TOAST extends the theory of timed asynchronous session types to support modelling of communication protocols featuring timeouts, which despite being commonplace in practice were previously out of reach for session type theory. We present ongoing work towards a practical toolchain that (a) automates the generation of correct-by-construction program stubs with timeouts in Erlang from TOAST processes that implement a TOAST protocol, and (b) provides an inline monitoring framework for TOAST protocols integrated with Erlang supervisors. Our toolchain generates Erlang code with a close correspondence to the source TOAST model by building on a formal correspondence between session types and Communicating Finite State Machines. The monitoring framework can be configured to perform either runtime verification or enforcement with respect to the source protocol, ensuring communication safety.

CCS Concepts: • **Theory of computation** → *Formal languages and automata theory*; *Concurrency*; • **Software and its engineering** → **Source code generation**; *Monitors*; • **Networks** → *Network protocols*.

Keywords: Code generation, Erlang, Timed Protocols, Runtime monitors, Asynchronous Communication

ACM Reference Format:
Jonah Pears, Laura Bocchi, and Raymond Hu. 2024. Erlang on TOAST: Generating Erlang Stubs with Inline TOAST Monitors. In *Proceedings of the 23rd ACM SIGPLAN International Workshop on Erlang (Erlang '24), September 2, 2024, Milan, Italy.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3677995.3678192

*This work has been partially supported by EPSRC project EP/T014512/1 (STARDUST) and the BehAPI project funded by the EU H2020 RISE under the Marie Sklodowska-Curie action (No: 778233).

Erlang '24, September 2, 2024, Milan, Italy

ACM ISBN 979-8-4007-1098-8/24/09
https://doi.org/10.1145/3677995.3678192

1 Introduction

Implementing concurrent systems with asynchronous communication is intrinsically complex. Such systems are often distributed and prone to failure [18]. *Session types* [4, 16, 17] is a type theory for concurrent communicating processes that offers formal methods for specifying and verifying the communication protocol between the processes.

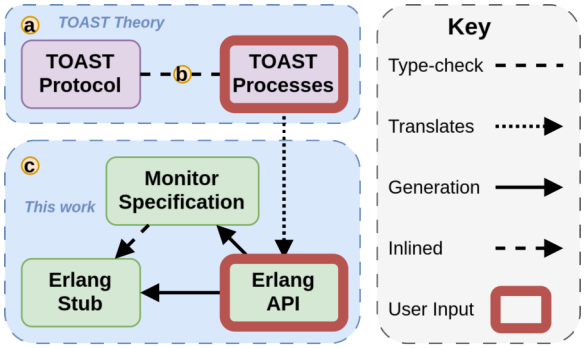
Recently proposed in [26] is a formal framework called TOAST, standing for **T**ime**O**ut **A**synchronous **S**ession **T**ypes. It extends the theory of *timed* asynchronous session types [6, 8] to support safe instances of *mixed states*. A protocol state is 'mixed' if the next action can be chosen from a set including both send and receive actions. In contrast, session types normally feature states that are either just sending or just receiving. TOAST enables modelling of an important class of protocols that feature interactions with *timeouts*, and supports reasoning about *safety* properties of such time-sensitive protocols, i.e., that the protocol is free of errors such as reception-errors and deadlocks. Previously, the restrictions imposed by asynchronous session types to ensure safety prohibited mixed-choice states and timeouts.

In this paper, we build on TOAST theory [26] to present our ongoing work towards a practical toolchain [1] for specifying and verifying asynchronous communication protocols featuring timeouts in Erlang. Our toolchain aims to (a) automate the generation of correct-by-construction program stubs with timeouts (and other mixed-choice) in Erlang from TOAST processes that implement TOAST protocols, and (b) provide a transparent inline monitoring framework for TOAST protocols that can be integrated with Erlang supervision trees, to ensure our stubs continue to adhere to their source TOAST model once users expand upon them with functionality for their specific use case. The code generation builds upon the formal correspondence between session types and Communicating Finite State Machines (CFSM) [12] to build the stubs incrementally.

Altogether, our toolchain ensures that the Erlang implementation and execution of a TOAST process that correctly implements a well-designed TOAST protocol is *safe* by a combination of correct-by-construction code generation and inline monitoring: the protocol is free of *stuck*-errors such as deadlocks, and all communication behaviours are either

verified or enforced to comply with the protocol (dependent on configuration). We show that our toolchain can (a) facilitate the design of application-level protocols, (b) offer formal guarantees of safety, and (c) aid in the development of implementations whose behaviour is faithful to the model.

Toolchain Overview. Shown in (1) is an overview of the toolchain we present in this work, for generating Erlang stub programs with runtime monitors from TOAST models.



(1)

Our work builds upon the theoretical work ⓐ of [26, 27] which presented TOAST types and TOAST processes.

Hereafter, we refer to TOAST types as *protocols*. An extension of [26] is currently in progress [27], which presents ⓑ a typing system checking a TOAST process correctly implements a given TOAST protocol. Our toolchain ⓒ can take as input either a TOAST process formulated using our Erlang API, or the TOAST process itself which our tool can automatically translate into our Erlang API. We discuss the differences between the two in Section 3 and Section 8.

Contribution Outline. Section 2 provides an introduction to the theoretical background of TOAST. An outline of our toolchain is presented in Section 3, and Section 4 provides details on how we generate stubs with some examples, and Section 5 describes our runtime monitors. In Section 6 we study a use case protocol, 'Two Factor Authentication' and discuss the difficulties that can arise when bridging the gap between designing and implementing time-sensitive protocols. Section 7 discusses related work. In Section 8 we conclude with a closing discussion and outline future work.

2 Background on TOAST

Session types [4, 16, 17] formally model the behaviour of processes in concurrent systems that interact via message-passing as part of an ongoing communication *session*. A session type specifies how interactions should occur in a session: send or receive, the message types, and their causality with other interactions. *Timed* session types [6, 8] further express timing constraints for the interactions. TOAST [26, 27] is an extension with safe mixed-choice that enables modelling behaviours previously out of reach, such as timeouts.

In TOAST protocols, each interaction consists of: (i) a direction of communication, either sending (!) or receiving (?), (ii) a message label, (iii) a condition tuple of time constraints and a set of virtual clocks to reset, and (iv) are prefixed by '.' to the continuation protocol. Eventually, a protocol either reaches with an 'end' or loops recursively.

As an example, in (2) S is a TOAST protocol modelling a server that waits for a message a with a *timeout* of 5 time units (seconds, say). If a is not received within 5 seconds, then S instead sends a notification b and retries to execute S.

$$S = \{?\mathtt{a}(x \quad 5, \varnothing).\mathtt{end}, \quad !\mathtt{b}(x = 5, \{x\}).S\} \tag{2}$$

Concretely, the behaviour of S as defined in (2), is a choice (curly brackets) between: (i) to receive a message a within 5s and then terminate, or (ii) to send a message b at any time after 5 seconds and continue as S (recursively).[1] Such choices are called 'mixed-choice' since they offer a 'mix' of both send and receive actions. Time constraints are expressed on a clock x that is local to S and is initially set to 0. A clock can be reset to express relative time constraints. For instance, in S, clock x is reset after the sending of b. Since x is reset before each iteration of the loop, this defines the behaviour of S as repeatedly send b at a rate of 5s until a is received and then S terminates.[2]

Safety. One advantage of *timed session types* is the notion of *well-formedness* [26], which acts as a sanity check that protocols are not intrinsically flawed, namely that they are *safe*. In the scope of this paper, we say a protocol is *safe* if it satisfies two properties: *feasibility* and *stuck-freedom*.

Feasibility. A protocol is *feasible* if there exists an implementation that satisfies its constraints. For example, in (3) S_F models a server that waits for a request that will arrive by time 5, and then replies by time 3. In a contract between a client and server, the server is guaranteed to receive `request` by time 5, yet is required to send a `response` by time 3.

$$S_F = ?\mathtt{request}(x \leq 5).!\mathtt{response}(x \leq 3).\mathtt{end} \tag{3}$$

While there are instances where S_F is able to progress and terminate successfully, the protocol is *infeasible* since, if `request` arrives at time 4, then the server is unable to fulfil its obligation of sending `response` by time 3.

In order to amend the protocol in (3), one could, for example, reset x after receiving `request`, or postpone the deadline for sending `response`.[3] Unlike the protocol in (2), S_F does not have a 'timeout branch' specifying how the server should behave if no `request` arrives by time 5.

Stuck-freedom. A protocol is *stuck-free* if neither client nor server ever get stuck in a state due to waiting to receive a message that will never arrive. Consider the protocol S_S together with its *dual* client C_S in (4). We say C_S and S_S are

[1] Note: for accessibility we use a simplified notation to the standard in [26].
[2] In line with the semantics of receive-after in Erlang, the timeout action (!b) can only happen if no message matching a is received while $(x < 5)$.
[3] Latency can be encoded into the model but it not shown here for simplicity.

dual since they share the same interaction structure and timing constraints but have opposing communication directions (and they have independent local clocks x and y).

$$S_S = \begin{Bmatrix} ?\mathsf{a}(x \leq 5).\mathsf{end}, \\ !\mathsf{b}(x \not< 0, \{x\}).?\mathsf{a}(x \not< 0).?\mathsf{c}(x \not< 0).\mathsf{end} \end{Bmatrix}$$
$$C_S = \begin{Bmatrix} !\mathsf{a}(y \leq 5).\mathsf{end}, \\ ?\mathsf{b}(y \not< 0, \{y\}).!\mathsf{a}(y \not< 0).!\mathsf{c}(y \not< 0).\mathsf{end} \end{Bmatrix} \quad (4)$$

S_S is *not* stuck-free: if after 2 seconds both parties decide to send a and b, respectively, then the client will terminate while the server will consume a and wait to receive c forever, getting *stuck*. Stuck-freedom also rules out *deadlocks*, where a party gets stuck waiting to receive from the other party.

Safe Mixed-choice and Timeouts. The expressive power of TOAST lies within its ability to model *safe* mixed-choice, which encapsulate the behaviour of timeouts and dually, *co-timeouts* along with other behaviours. (We discuss co-timeouts in Section 4.4. Put briefly, while a timeout waits to receive before sending, a *co-timeout* must send within a given time range before then waiting to receive.) A mixed-choice is an interaction pattern where a process is able to perform either a send or receive action from the same state.

For example, both (2) and (4) feature mixed-choice between exchanging a or b from one party to another. Mixed-choice, especially when combined with asynchronous communication, have been ruled out by previous work on session types, as they may be unsafe. In fact, we observed in (4) that mixed-choice with unruly time-constraints can even lead dual participants to get stuck by becoming incompatible.

The main contribution of TOAST [26] is how to structure mixed-choice in such a way that it is *safe* and hence, capable of modelling timeouts, by using a notion of *well-formedness*. Concretely, well-formedness is a decidable algorithm that, for each state, ensures: there is always at least one viable future (feasibility) and, there only actions of the same direction may occur at the same time, and actions with differing directions must occur in disjoint periods of time (stuck-freedom).

For example, recall protocol (4) is *not* stuck-free since there is a mixed-choice between the client sending a and receiving b at the same time, since $(x \leq 5)$ and $(y \not< 0)$ are both true when initially $x = y = 0$ and $(y = x) \leq 5$. By contrast, protocol (2) is stuck-free, since the timing constraints on sending and receiving actions have no intersection.

TOAST allows us to express mixed-choice, which underpin a very common programming pattern: *timeouts*, both those that would yield a termination of a session, and those with continuations such as ones that could be expressed with the Erlang 'receive-after' expression. More complex behaviour can also be expressed, such as more than one interleaved sending and receiving intervals (nested timeouts) and, interactions with multiple clocks (useful to model absolute and relative constraints in the same protocol). The main contribution of the (under review) extension [27] of TOAST [26] is a typing system for checking TOAST processes adhere to TOAST protocols. Both the well-formedness and the type-checking algorithm rely on a Simple Theorem Prover (STP) Solver to reason on time constraints. Put briefly: (i) a *well-formed* TOAST protocol is guaranteed to be free from deadlocks and enjoy progress [26]; (ii) a *well-typed* TOAST process has been statically type-checked to only behave as prescribed by a TOAST protocol.

In this paper we address the problem of generating safe implementations and monitors from safe TOAST processes. We assume that a given TOAST process is *well-typed* against a *well-formed* TOAST protocol.

3 A TOAST-to-Erlang Toolchain

3.1 TOAST Processes in Erlang by Example

Our toolchain relies on an Erlang API for defining TOAST processes which implement TOAST protocols. Following [26, 27], our Erlang API supports the construction of protocol implementations composed of send and receive actions with time constraints, safe mixed-choice and recursion.

We require the following adaptations to the time-sensitive components of TOAST in order to integrate with Erlang. Firstly, due to the behaviour of the 'receive-after' pattern in Erlang, we can model the behaviour of *inclusive* constraints (i.e., $\geq$, $\leq$ and $=$) but not *exclusive* constraints (i.e., $\not<$, $>$ or $\neq$).

Secondly, the timers native to Erlang count *downwards* from a set value, and upon reaching 0 send a message to a predefined process. However, the timers in TOAST processes count *upwards* from 0 when set, and their values can be used for conditional statements for time-sensitive selection. Therefore, when defining a TOAST process using our Erlang API, it is typical to define a new timer in the Erlang API for each condition imposed on a timer, as this provides the upper-bound necessary. (See Example 3.1.) In practice, either can be used as input since our toolchain can automatically translate TOAST processes to our Erlang API.

Example 3.1 (TOAST Process Timers & Erlang API)**.** For example, consider the following TOAST process:

$$P = \mathtt{set}(x).\mathtt{delay}(t \leq 7).\mathtt{if}\ (x = 0)\mathtt{then}\ Q_1 \quad \mathtt{else\ if}\ (x \quad 5)\mathtt{then}\ Q_2 \quad \mathtt{else}\ Q_3 \quad (5)$$

In (5), some TOAST process P sets a timer x and experiences a non-deterministic delay of up to 7 time units in duration. If there is no delay (i.e., $x = 0$) then P proceeds as process Q_1. Else, P proceeds as Q_2 if $x \quad 5$ and otherwise as Q_3. Such a TOAST process translates to the following Erlang API:

```
1 _P() ->{timer,"x0",1,{timer,"x5",5000,_P1()}}.
2 _P1()->{delay,7000,{if_timer,"x0",_P2(),_Q1()}}.
3 _P2()->{if_timer,"x5",_Q2(),_Q3()}.
```

where `_Q1()`, `_Q2()` and `_Q3()` are further Erlang API which correspond to Q_1, Q_2 and Q_3 in (5) respectively. Notice on line 1 of the above: (a) timer `x0` is set with duration 1, and (b)

for each occurrence of x in the time constraints of the TOAST process in (5), we defined distinct timers in the Erlang API.

For both (a) and (b), consider the Erlang snippet below:

```
1 P() -> erlang:start_timer(1,self(),timer_x0),
2   erlang:start_timer(5000,self(),timer_x5),P1().
3 P1() -> timer:sleep(rand:uniform_real()*7000),
4   receive {timeout,_TID,timer_x0} -> P2()
5   after 0 -> Q1() end.
6 P2() -> receive {timeout,_TID,timer_x5} -> Q2()
7   after 0 -> Q3() end.
```

The above corresponds to (5) and the previously shown Erlang API. (Note: the above is a simplified Erlang code snippet implementing (5), and only serves to expose the approach used by our tool when generating Erlang stubs.)

As before, `P()` (lines 1–2) initialises the timers and proceeds to `P1()`. While it is possible to read the value of an Erlang timer, since an upper-bound is always required, this approach has shown to be simple and effective for our purposes.

Observe `P1()` on lines 3–5 which reads, delay for a random duration between 0 and 7 seconds, if '`timer_x0`' has completed, then proceed to `P2()`, since following (5) we should only proceed to `Q1()` if no delay has been experienced. Otherwise, in the case of a delay we proceed to `P2()`. Similarly, observe `P2()` on lines 6–7, which more intuitively follows (5) and reads, if '`timer_x5`' has completed then proceed to `Q2()`, else `Q3()`. The use of `after 0` enables the mailbox to be checked for the message from the timer in a *non-blocking* manner. If the timer has not completed yet (i.e., less than 5 time units have passed) then we are able to proceed to perform the corresponding action in `Q3()`. △

Example 3.2 (Simple Timeout)**.** Recall the TOAST protocol in (2). We define an implementation using the Erlang API:

```
1 S() -> {timer,"x5",5000, {rec,"A", {
2   act,r_a,endP, aft,"x5", {
3     act,s_b,{timer,"x5",5000, {rvar,"A"}}}}}}.
```

While in the theory all *clocks* are initially 0, in our Erlang API, all *timers* must be set to an initial value. Above, we set timer `x5` to 5000 milliseconds, define a recursive point `A` and then, wait to receive `a` until timer `x5` reaches 0, at which point if `a` has not been received, we then send `b`. After sending `b`, we then reset timer `x5` before then looping back to `A`.

Alternatively, (2) can be expressed without using timers:

```
1 S() -> {rec, "A", {act, r_a, endP,
2   aft, 5000, {act, s_b, {rvar, "A"}}}}.
```

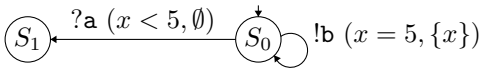
We discuss when timers must be used in Section 4. △

Remark 3.3 (Equality Constraints)**.** Recall Example 3.1, where for the constraint $(x = 0)$ we utilise a timer with minimal duration (1ms) to determine if there had been any delay at all. When implementing the behaviour of other such constraints, where for some timer y, we say $(y = n)$, then we follow similarly to Example 3.1, by defining two timers, one set to n in milliseconds and the other of $n + 1$ms. △

3.2 From TOAST Processes to Finite-State Machines

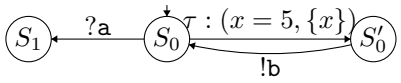
Given a TOAST process defined using our Erlang API, our tool begins by extracting a finite-state machine (FSM) which is used internally for generating code. This is common in other works that generate code from session types [5, 9, 10, 13, 20–22, 25] following the close correspondence between session types and communicating finite state machines (CFSM) [12]. Due to the encoding of TOAST clocks and constraints using Erlang timers discussed in Section 3.1, we use an intermediate representation of FSMs that differs from the formal Communicating Timed Automata (CTA) [19] that most directly correspond to TOAST theory. This intermediate representation is *only* used internally by the tool, for the joint purposes of generating: (a) Erlang stubs and, (b) protocol specifications used by our monitors.

The internal FSM representation is *untimed* since we do not model the behaviour of clocks. Instead, we extend the kinds of *edges* the FSM supports to encapsulate the additional information. For example, (2) is equivalent to:



(6)

where $S = S_1$. However, upon using as input to our tool (2) written using the Erlang API (shown in Example 3.2), our tool instead produces the following internal representation:



(7)

As shown in (7), mixed-choice are transformed from one 'mixed' state, to a series of non-mixed states, connected via silent actions (τ) that correspond to an interval between sending or receiving actions. The FSM representation simplifies the process of generating our stubs, by allowing each edge to be built in isolation from Erlang snippets.

3.3 From FSMs to Erlang Stubs

Using the internal FSM extracted from a given protocol, our tool then proceeds to traverse the FSM depth-first in order to generate the stubs. Put briefly, we handle each state depending on a classification that reflects the kinds of outgoing edges it has. We build our stubs by combining snippets that correspond to both the state as a whole, and each outgoing edge. We handle recursion by moving the body of the loop into a new tail-recursive function, which is entered via a function call in the outer-scope. We discuss this in Section 4.

3.4 From FSMs to Runtime Monitoring

Once we have obtained the internal FSM, our tool then converts it to a more accessible format that our monitoring template can use to either perform runtime verification of a process, or runtime enforcement. (For more details on runtime monitoring see [2, 14].) Our monitors are intended to be used within Erlang supervision trees, so that any violation

of their prescribed protocol yields a measured response from the system, as specified. Our monitors are a single Erlang file template, and are capable of monitoring the interactions of an assigned process against a protocol specification automatically generated from the Erlang API when generating the stub programs. As part of our toolchain, this protocol specification is derived from our TOAST protocols, via the internal FSM. We discuss our monitors further in Section 5.

4 Erlang Stub Generation

We explain how our toolchain generates program stubs from TOAST processes that are well-typed against a well-formed TOAST protocol. We first provide further details of our Erlang API which our tool uses as input and can be automatically translated from TOAST processes. We rely upon the fact that a TOAST process that is well-typed against a well-formed TOAST protocol, since it provides us with certain guarantees of the behaviour of the TOAST process, without having to worry about reasoning on the potentially far more complex timing constraints of TOAST protocols in Erlang. This streamlines the process of going from the theory of TOAST processes to our Erlang API, and ultimately, Erlang stubs. The mapping between TOAST processes and our Erlang API is included in the tool [1]. This section provides a conceptual overview of the mapping between these two.

Erlang API for TOAST Processes. Below we provide an overview of the constructors provided by our Erlang API: (a) `{timer, t, n, S}` creates an Erlang timer named `t` with a duration `n` and continues to `S`. (b) `{act, a, S}` allows action `a` to be performed with continuation `S`. (c) A simple timeout: `{act, a, S1, aft, d, S2}` behaves similarly to (b) except action `a` must be performed before duration `d` elapses, where `d` is either: an integer duration in milliseconds or, a *timer* `t` from (a). Output selection and input branching are specified by `{select, [{a1, S1},...]}` and `{branch, [{a1, S1},...]}` respectfully. Both are capable of specifying a *timeout*, as in (c). Beyond timeouts, `{if_timer, t, S1, S2}` behaves as `S1` if timer `t` *has* completed, or as `S2` otherwise. The opposite behaviour is corresponds to `{if_not_timer, t, S1, S2}`. Recursive definitions and calls are specified by `{rec, r, S}` and `{rvar, r}` respectfully, where `r` is a recursive variable. Delays are specified by `{delay, d, S}` which behave as `S` after delay `d`. Successful terminations are denoted by `endP` while protocol violations are denoted by `error`.

In the following subsections, we explain how behaviour specified by TOAST protocols and TOAST processes corresponds to Erlang code through examples of core communication protocol patterns and interaction structures, and aim to provide some insight into how our tool builds Erlang stubs.

4.1 Delays, Constraints & Errors

Consider the protocol below, which features two interactions, each constrained by an upper and lower bound:

$$E = \{?\mathsf{a}(1 \quad x \leq 5, \varnothing).\mathsf{end}, \quad !\mathsf{b}(7 \quad x \leq 9, \varnothing).\mathsf{end}\} \quad (8)$$

The time constraints of E describe two intervals, $(1, 5)$ and $(7, 9)$, separated by a gap of 2s. The protocol in (8) can be written using the Erlang API as follows:

```
E() -> {delay, 1000, {
  act, r_a, endP, aft, 4000, {delay, 2000, {
    act, s_b, endP, aft, 2000, error}}}}.
```

Above, the `delay` requires that the process waits 1 second before starting to receive `a`, matching the lower bound of constraint $(1 \quad x \leq 5)$ in (8). The upper bound is rendered using `aft`. Next, we model the lower-bound of sending `b` with a 2s delay. Finally, in order to capture the upper-bound of 9s for sending `b`, we introduce a timeout leading to `error` . Here, reaching error corresponds to violating the protocol, and can be handled by adding appropriate action at the level of the implementation. An alternative way to write E using the Erlang API is by using explicit timers as shown in (8):

```
E() -> {timer, "x5", 5000, {timer, "x7", 7000,
  {timer, "x9", 9000, {delay, 1000, {
    act, r_a, endP, aft, "x5", {delay, "x7", {
      act, s_b, endP, aft, "x9", error}}}}}}}.
```

While either of the above could be valid mappings of (8), in our automation of the mapping we are opting for the latter for generality, to rely on the flexibility of timers with respect to hard-coded constraints. A hard-coded value for an 'after' branch is prone to allowing a process to receive for longer than intended, in the case where the process experiences prior delays. However, we can use timers to address this issue, since they are a separate process, they are unaffected by delays the main process experiences (intended or otherwise).

Interleaved Mixed-choice. TOAST allows for protocols where a mixed state has multiple intervals in which send and receive action interleave. For example:

$$S_1 = \{?\mathsf{a}(x \leq 5, \varnothing).S_2, !\mathsf{b}(5 \quad x \leq 7, \varnothing).S_3, ?\mathsf{c}(x \not\leq 7, \varnothing).S_4\}$$

Implementing the behaviour of such a protocol (or its dual) as a TOAST process is straightforward, by using timers combined with time-sensitive conditional statements. Such a TOAST process maps to the following Erlang API:

```
S1() -> {timer,"x5",5000, {timer,"x7",7000,
  {act,r_a,S2(), aft,"x5", {if_timer,"x7",
    {act,s_b,S3()}, {act,r_c,S4()}}}}}.
```

Basically, `S1()` behaves as a timeout in interval $(0, 5)$ and as a *co-timeout* in interval $(5, 7)$ (namely as the party that has to provide a message `b` within a deadline). The input notation of `S1()` uses a conditional statement on a timer `x7` to express this situation: if you are in time then send `b` otherwise receive `c`.

Multiple clocks. The following example illustrates the mapping of a recursive feeds-server that uses multiple clocks.

$$S_F = \{!\texttt{stop}(20 \leq y).\texttt{end}, \quad !\texttt{feed}(x \leq 1 \wedge y \quad 20, \{x\}).S_F\}$$

S_F models a server that repeatedly produces feeds at the pace of 1s and stops after 20s. The *feeds-server* corresponds to a TOAST process which maps to the following Erlang API:

```
1 Sf() -> {timer,"x1",1000, {timer,"y20",20000,
2   {rec,"Sf", {if_timer,"y20",
3     {act,s_stop,endP},
4     {delay,"x1", {act, s_feed,
5       {timer,"x1",1000, {rvar,"Sf"}}}}}}}}.
```

Above, the timer `y20` is used to track the total amount of time the feeds-server has been looping. Only once 20s have elapsed does the feeds-server send `stop`. Before `y20` reaches 0, the feeds-server sends `feed` at a rate no faster than every 1s.

4.2 Stub Behaviour

By default, generated stubs include a file named `stub.hrl` which contains a palette of useful functions utilised by our tool during generations. We choose to move this behaviour to an Erlang header file to improve the readability and mitigate duplicated code. Each function in our stubs take two parameters: (1) the process ID of the other party in the sessions (`CoParty`) and, (2) a map to store necessary data accumulated as the program runs (`Data`). When initially generated, the map `Data` is intended to store the timers started and any messages received by the program; although, the user may store additional information in `Data` when they extend the stub.

4.3 Simple Timeout Stub

Erlang is capable of implementing the behaviour of a timeout directly, thanks to the 'receive-after' expression. A programmer may choose to implement the behaviour of (2) using:

```
1 S(CoParty, Data) ->
2   receive {CoParty,a,Payload} -> exit(normal)
3   after 5000 -> CoParty ! {self(),b,some_data},
4     S(CoParty, Data) end.
```

Note the format of the messages exchanged: a triple comprised of the process ID, message label and message payload.

Our tool utilises the 'receive-after' expression of Erlang, and also systematically implements functionality to aid the user when expanding the stubs, including: (1) State-by-state date isolation, providing the user with a distinct `Data` variable for each state, before any actions are performed. (2) Automatic saving of received messages to `Data`. (3) Placeholder functions for acquiring payloads to send as part of a message.

Below is an example of the kind of code generated by our tool from the protocol (with timers) in Example 3.2 for (2):

```
1 main(CoParty,Data) ->
2     {Data1,_TID_x5} = set_timer(x5,5000,Data),
3     S(CoParty,Data1).
```

The function call (`set_timer(x5,5000,Data)` on line 2 will create a new timer (via `erlang:start_timer(x5,self(),5000)` and returns the process ID of the timer process and the updated `Data` containing the timer under the name `x5`. If a timer under that name already exists, then it is cancelled and replaced with the new one (effectively acting as a reset). On line 3 we enter a function `S` since, recall in (2) and (6), the initial state is immediately re-entered after sending `b`. Therefore, we must begin a new scope:

```
4 S(CoParty, Data) ->
5   {ok, _TID_x5} = get_timer(x5,Data),
6   receive {CoParty,a,Payload_A} ->
7     Data2 = save_msg(a,Payload_A,Data1),
8     stopping(CoParty,Data2);
9   {timeout, _TID_x5, timer_x5} ->
10    {Data2,Payload_B} = get_payload_b(Data1),
11    CoParty ! {self(), b, Payload_B},
12    {Data3,_TID_x5} = set_timer(x5,5000,Data2),
13    S(CoParty, Data3) end.
```

Following (2) or (6), the process must first wait to receive `a` for the duration of 5s. Shown above, the process first retrieves the process ID for the corresponding timer (`_TID_x5`) and proceeds to wait for the duration. If `a` is received from `CoParty`, then the message is saved to `Data2` and the process terminates successfully (via `stopping(...)`). Otherwise, if timer `_TID_x5` completes, then the process: (1) sends `b` (along with some payload `Payload_B`) to `CoParty` (2) resets the timer `x5`, and, (3) loops via tail-recursion.

Functions such as `get_payload_b` can introduce vulnerabilities into processes, if `get_payload_b` never completes. These vulnerabilities are somehow orthogonal to the problem space of TOAST that only guarantees stuck-freedom of the interaction structure. However, in the design of a TOAST protocol, if one foresees that some actions are likely to be diverging, a timeout should be introduced.

Remark 4.1 (Timer Timeouts)**.** Shown on line 9 in the code snippet above is a method for using Erlang timers as timeouts, rather than the value-based timeout of the 'receive-after' expression. To reiterate Section 4.1, while both approaches are functionally equivalent, using timers as in this approach can be more robust in the presence of delays. △

4.4 Co-Timeouts

We will now discuss stub generation for timeouts from the perspective of the other role involved: the one that needs to provide a timely message. We will refer to these co-parties as *co-timeouts*. First, consider the *dual* D of the protocol in (2):

$$D = \{!\texttt{a}(y \quad 5).\texttt{end}, \quad ?\texttt{b}(y = 5, \{y\}).D\} \tag{9}$$

Naturally, (9) describes a client that is either able to send `a` within 5 time units, or afterwards wait to receive `b`. Receiving `b` causes clock y to be reset, and loops back to the start. The implementation of a co-timeout is less intuitive than the one of its co-party. There are three possible ways of

implementing D: (a) The process that is always too late to send a. (b) The process that is always early and sends a. (c) The process that can send a by the deadline *sometimes*. The case of (b) effectively makes receiving b redundant. We now discuss how to implement cases (a) and (c).

Arriving Too Late. In the case of (a) some prior delay must have occurred, causing the upper-bound of sending a to be missed. Following Remark 4.1, we implement using timers:

```
D(CoParty,Data) -> TID_x5 = get_timer(x5, Data),
  receive {timeout, TID_x5, timer_x5} ->
    receive {CoParty, b, Payload_B} ->
      Data1 = save_msg(b, Payload_B, Data),
      D(CoParty, Data1) end;
  after 0 ->
    {Data1, Payload_A} = get_payload_a(Data),
    CoParty ! {self(), a, Payload_A},
    {Data2, _TID_x5} = set_timer(x5,5000,Data1),
    D(CoParty, Data2) end.
```

Above, we implement a non-blocking receive-after by having a timeout of duration of 0s, ensuring that `D` must be able to receive the signal that the timer `x5` has completed *immediately*. In the case that timer `x5` has completed and `D` is able to receive `{timeout, TID_x5, timer_x5}`, then `D` begins to wait to receive `b`. Otherwise, less than 5s have passed, and `D` is able to go ahead and prepare to send `a`. What if `get_payload_a` is *unreliable* and may get stuck? Such a scenario falls under (c).

Unreliable Dependencies. In the case of (c) then there is some factor that sending a is dependant upon. Such as, a function for obtaining the payload to send along with a which is *unreliable* or in the least, not guaranteed to complete within the time frame. In our case, this is function `get_payload_a`. We need a way of calling `get_payload_a` while ensuring the main process remains non-blocked and responsive.

We present an Erlang snippet below:

```
nonblocking_payload(Fun, Args, PID, Timeout)
when is_integer(Timeout) -> spawn( fun() ->
 Waiter = self(),
 Timer= erlang:start_timer(Timeout,self(),nb_p),
 TimeConsumer = spawn( fun() ->
    Waiter ! {self(), ok, Fun(Args)} end ),
 receive {TimeConsumer, ok, Result} ->
   PID ! {self(), ok, Result};
 {timeout,Timeout,nb_p} ->
   PID ! {self(), ko},
   exit(TimeConsumer, normal) end end );
```

Above, `nonblocking_payload` creates a new timer for the duration of `Timeout` and spawns a process `TimeConsumer` to complete the potentially *unreliable* function `Fun(Args)`, and send the results back to itself. If `Fun(Args)` completes before `Timer`, then the `Result` is sent back to the main process via `PID` with label `ok`. Otherwise, the message `ko` is returned, signalling it took too long and in the case of (9), that `D` should begin waiting to receive b. Below is another clause of `nonblocking_payload` which takes a timer, reads the value and re-enters through the function previously shown.

```
nonblocking_payload(Fun, Args, PID, Timer)
when is_pid(Timer) ->
  Value = erlang:read_timer(Timer),
  nonblocking_payload(Fun, Args, PID, Value).
```

In practice, our tool utilises `nonblocking_payload` for the case of co-timeouts in the following way:

```
send_before(CoParty,{Label,{Fun,Args}},Timeout)
when is_integer(Timeout) ->
 NonBlocking =
   nonblocking_payload(Fun,Args,self(),Timeout),
 receive {NonBlocking, ok, Result} ->
   send(CoParty, {Label, Result});
 {NonBlocking, ko} -> ko end.
```

This snippet above outsources the potentially blocking behaviour to another process spawned within the same node, and waits to either receive `ok` and a payload `Result`, which is then sent to `CoParty`, or receive `ko` which indicates the function took too long to complete. The snippet above is utilised by our tool to generate code corresponding to co-timeouts. Similarly to before, we also support the use of timers:

```
send_before(CoParty,{Label,{Fun,Args}},Timer)
when is_pid(Timer) ->
 Value = erlang:read_timer(Timer),
 IsKo =
  send_before(CoParty,{Label,{Fun,Args}},Value),
 case IsKo of ko ->
  receive {timeout, Timer, _Name} -> ko end;
  _Else -> _Else end.
```

Since the value of the `Timer` is passed through and used by the spawned process to determine if the function finishes on time, in the case that `ko` is returned, the original timer will have also reached 0 and so we remove this message from the mailbox of our main process `D`.

4.5 Producer-Consumer Pattern

Next, we illustrate recursion using the Producer-Consumer pattern. We only show the *producer* behaviour in (10):

$$P_S = ?\texttt{start}(\texttt{true}, \{x\}).P'_S \qquad P'_S = \left\{?\texttt{stop}(x \leq 1).\texttt{end}, \quad !\texttt{data}(x \text{ ¡ } 1, \{x\}).P'_S\right\} \tag{10}$$

Since the *producer* must both send and receive from the same state, this behaviour is clearly mixed-choice. The mixed-choice are inherently *safe*, as illustrated in (10). Note, the constraint 'true' indicates no upper or lower bound.

After receiving `start`, the *producer* first waits to receive a `stop` signal for 1s. If `stop` is received, the *producer* terminates. Otherwise, the *producer* is to send `data`, and then reset x and recursively loop to P'_S. For the first second of each iteration, the *producer* must wait in case a `stop` signal can be received. The *producer* in (10) can be written using the Erlang API as follows:

```
Ps() -> {timer,"x1",1000,{act,r_start,{rec,"Ps",
  {act,r_stop,endP, aft,"x1", {act,s_data,
      {timer,"x1",1000, {rvar,"Ps"}}}}}}}.
```

Above, when `x1` completes the *producer* can send `data`. Since in (10) there is no upper-bound constraint for the *producer* sending `data`, the protocol allows this to take potentially forever. It is plausible that the use case for implementing a protocol with a Producer-Consumer pattern, the *producer* is dependant on some other service for obtaining the data to send. Since this is the latest possible action, if this service encountered an error and the *producer* was unable to send `data`, then the *producer* would become stuck, as would the *consumer*. Ideally, the latest action should be a dependency-free timeout signal used to provide a means to stop a program from being stuck when another process has seized [28].

For example, we can amend P'_S in (10):

$$P'_S = \begin{Bmatrix} ?\texttt{stop}(x \leq 1).\texttt{end}, \\ !\texttt{data}(1 \quad x \quad 99, \{x\}).P'_S, \\ !\texttt{pass}(x \ ¡ \ 100, \{x\}).P'_S \end{Bmatrix}$$

Above, instead of a timeout, we have added the option to send `pass` if `data` on time. In practice, our tool would utilise the `nonblocking_payload` function discussed in Section 4.4.

5 Runtime Monitor Generation

A Monitoring Template. The runtime monitors in our tool are not themselves generated. Instead, our tool provides a single monitoring template program, and generates the protocol specification in the form of an Erlang map, which is derived from the internal FSM representation. We choose to use this map representation since it allows us to more easily make use of pattern matching in function guard sequences.

Protocol Specification. Currently, the protocol specification provided for our monitors is derived from TOAST processes rather than TOAST protocols. In some cases it is possible to have a TOAST process that is a *maximal implementation* of a TOAST protocol, and in such cases the process encapsulates the full range of behaviour described by the protocol. Therefore, it follows that a protocol specification derived from a maximal TOAST process is also capable of being used to monitor Erlang stubs derived from *non-maximal* TOAST processes. (Discussed further in Section 8.)

Transparency. Our monitors act as transparent mediators for communication, and can be inlined at either party within a session. Being transparent, our monitors do not modify the contents of the messages exchanged and therefore, neither party in a session has to be aware of whether the other party, or even if they themselves are being monitored.

Configurability. Our monitors are highly configurable, capable of performing both runtime verification and runtime enforcement. Runtime verification is the default, where they remain transparent and any violation will result in the supervisor being notified. For runtime enforcement, we provide several presets in the project documentation [1], such as:

Enforce (strong) is a are more flexible preset than *verification* that allows the timing of interactions to be minimally adjusted to be more lenient, whilst still ensuring that the protocol is not violated. (I.e., allowing sending and receiving actions to be re-tried at the next available state, in the case they were received a state too early.)

Enforce (weak) preset allows the monitor to fully handle the timing of interactions for their monitored process within a session. In effect, this preset causes a monitor to only strictly enforce that interactions occur as prescribed by the protocol, and does not verify when the process actually attempts to perform certain actions.

The configurability of our monitors enable them to be useful outside the application of our toolchain. The enforce (weak) preset allows developers to write completely *time unaware* programs that can adhere to *time-sensitive* protocols, since the timing of interactions is handled by the monitor.

Handling Events. Our template uses Erlang `gen_statem` behaviour to traverse the FSM map. By using callback mode `handle_event_function` we are able to describe how each kind of event should be handled, rather than describing each and every event that happens for each state (as it would be using `state_functions` callback mode). Therefore, the programs of the monitors themselves are of a fixed size with minimal code duplication, compared to generated monitors using `state_functions`, whose size grows with the number of states and edges of the FSM.

For example we only define once how the monitors handles message receptions, which is by checking if the current state has a receiving action matching the label of the message received. In the case where the reception is *not* prescribed, then the monitor will signal to the supervisor that Otherwise, if the reception *is* prescribed then, the monitor will forward the message to the monitored process and proceed to the next state (just as an FSM would behave). Continuing the example of (2), the internal FSM in (7) becomes:

```
1  #{init => state1a_recv_a,
2    map => #{state1a_recv_a =>
3               #{recv => #{a => stop_state}},
4             state1b_send_b =>
5               #{send => #{b => state1a_recv_a}}},
6    timeouts => #{
7      state1a_recv_a => {x5, state1b_send_b},
8      state1b_send_b => {?EQ_LIMIT_MS, error} },
9   resets => #{state1a_recv_a => {x5 => 5000}},
10  errors => #{state1b_send_b => took_too_long} }.
```

In the above FSM map: (i) line 1 specifies the initial state. (ii) lines 2–5 is a map between states, where each state has a set of `send` or `recv` actions, each with their own labels and resulting states. (iii) lines 6–8 is a map between states and a

tuple that defines a timeout, with a duration and destination, where the duration can be an integer value for milliseconds, or reference an Erlang timer. (iv) line 9 shows a map for specifying upon entering a state, which Erlang timers to start (if any) and what value to start them with. (v) line 10 shows a map between states and error messages, which are to be returned if state `error` is reached from any of the states.

The macro `?EQ_LIMIT_MS` is the degree of leniency given to actions that are meant to be performed at an exact time. Since constraints use integers as constants, we allow the user to specify how precise these constraints should be. However, a user should note that these values should be the same in practical scenarios where both participants are monitored.

For example, in the case of (2), sending `b` must happen when $(x = 5)$. Once state `state_1b_send_b` is reached, a timeout is set for the duration of `?EQ_LIMIT_MS` which when triggered will automatically state-transition to `error`, signalling to the supervisor that the protocol has been violated. However, if the monitored process performs the action of sending `b`, then the subsequent state-transition to `state1a_recv_a` cancels the current timeout that leads to `error`.

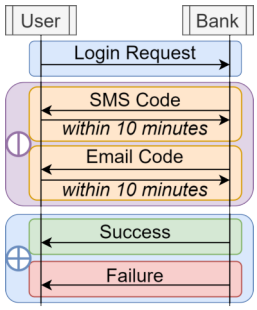
Extensibility. Our template specifies how events should be handled by the monitor in general, providing a foundation for runtime monitoring of an Erlang program against the specification derived from the Erlang API.

The event handlers found in the template are specified as generically as possible in order to facilitate as many states as possible. Each kind of event handler function is separated and clearly annotated within the template. In the case where the user requires more specific or unique behaviour from an event handler, it is simply a matter of: (1) re-defining the event handler on the lines above the generic behaviour, using the original handler as a basis, and then (2) amending the pattern matching guard sequence of the function to specify the distinct and identifiable circumstances this definition should handle (e.g., specifying the specific state where this should be called). Finally, (3) add the desired functionality to the body of the newly define event handler function. If done correctly, all other instances of the even will be unaffected by the new addition, and be routed through the to the same function clauses as before. By default, the monitoring template file features an example of this unique behaviour with a set of event handler functions that serve to 'reserve' a transition with the label '`emergency_signal`'. In the case that a monitor performs an action with the label '`emergency_signal`' (initiated by the monitored process) then, regardless of the protocol specification, the monitor will create a file containing all of it's logs, and signal to the supervisor that an error has occurred. This is just one example of how the generic event handling behaviour of our template can be extended to handle specific use cases, even beyond those prescribed by the protocols. Naturally, such behaviour is *also* a violation of protocols, and is merely offered as a debugging tool.

6 Use Case: Two Factor Authentication

In this section we discuss the design and implementation of a Two Factor Authentication (2FA) protocol. For services that need to be secure, such as online banking, it is crucial to ensure that a given user is who they claim to be. Such services typically employ 2FA to verify the integrity of a users claimed identity, and it is common for such protocols to enforce timing constraints for the user's response.

In (11) we present a mock-up 2FA protocol which describes, upon a login request from a User, a Bank sending two different codes to the User, one via SMS and the other to their email. (11) specifies that the User must send these codes back to the bank within 10 minutes, after which the bank will determine if the login was a success of failure. The symbol (⊕) indicates that the SMS and Email interactions can occur in parallel, and the symbol (⊕) indicates a choice between the bank sending either `Success` or `Failure`. It can be interpreted that these initial messages sent by the bank are assumed to occur instantly.



(11)

In (12), we model (11) as a TOAST protocol, where we 'collapse' the parallelism into a single sequence of actions since otherwise, we would need to specify all possible interleavings:

$$U_0 = \texttt{!login(true, \{}x, y\texttt{\})} \\ \texttt{.?email\_code(true).!email\_auth(true)} \\ \texttt{.?SMS\_code(true).!SMS\_auth(true)} \\ \cdot \begin{Bmatrix} \texttt{?success(}x \quad 10 \wedge y \quad 10\texttt{).}U_1, \\ \texttt{?failure(}x \geq 10 \vee y \geq 10\texttt{).end} \end{Bmatrix} \tag{12}$$

Above, (12) specifies that `email_code` will be received before the `SMS_code`, and `SMS_code` will not be received before the `email_auth` is sent back. While capturing the parallel behaviour in (11) using TOAST is possible, it would be cumbersome. Thankfully, the mailboxes of Erlang processes enable (12) to be a plausible model of (11) since messages may be received slightly out of order, and the end result of sending either `success` or `failure` depends on the parallel interactions both completing. (See fork and join in [12].)

However, to deploy runtime monitoring for *verification* the monitor requires an extended description of (12), detailing *all* possible traces, since *verification* does not allow message receptions to be postponed (discussed in Section 8).

Misleading Timing Constraints. Notably, in (12) there are no timing constraints for how promptly the user is required to respond to the bank. Instead, the timing constraints determine whether a `success` or `failure` are returned to the user, which is misleading since the contents of the codes are also crucial in determining whether the user is authenticated or

not. Therefore, (12) could be improved by adding a timeout branch at each point of the protocol to determine if the bank should continue to wait for the users codes if its is already guaranteed that they will result in `failure`.

Below we provide an input protocol for our tool corresponding to behaviour of (11) *and*: (1) the additional timeouts on the user responding within 10 minutes, (2) the full parallelised behaviour that was missing (or 'collapsed') in (12):

```
u0() -> {act,s_login, {rec, "retry", {
  timer,"x600",600000, {
    branch, [ u0(branch_r_email),
              u0(branch_r_sms) ]}}}}.
```

where timer `x600` is used to enforce an overarching timeout of 10 minutes for the user to send back `sms` and `email`.

```
u0(branch_r_email) -> {act,r_email, {
 act,s_email, {act,r_sms, {
   act,s_sms, {
     branch, [ {act,r_success,u1()},
               {act,r_failure,endP} ]},
   aft,"x600", {act,r_timeout,{rvar,"retry"}}}},
 aft,"x600", {act,r_timeout,{rvar,"retry"}}}};
```

Note that in the case above, the protocol specifies that the user must respond to `email` before receiving `sms`. In practice, the bank and the user are *not* dual and therefore, the bank will likely send both `email` and `sms` in sequence, before waiting to receive either `success` or `failure`. Below is the other branch:

```
u0(branch_r_sms) -> {act,r_sms, {
 act,s_sms, {act,r_email, {
   act,s_email, {
     branch, [ {act,r_success,u1()},
               {act,r_failure,endP} ]},
   aft,"x600", {act,r_timeout,{rvar,"retry"}}}},
 aft,"x600", {act,r_timeout,{rvar,"retry"}}}}.
```

Note that each user's sending action to the bank, is now a *co-timeout*. If timer `x600` completes at any point before the user has sent both codes, then they must stop and wait to receive a timeout from the bank, and will be able to `retry`.

7 Related Work

Protocol re-engineering. We build on the tool that accompanies [7], which takes an *untimed* process as input and generates Erlang `gen_statem` stubs for API implementation. In the original tool [7] an internal FSM is extracted from the input, which is used to generate the stubs. Our tool [1] extends [7] from here, using the extracted FSMs to generate (non-`gen_statem`) Erlang stub implementations. While `gen_statem` offers a close correlation with the theory, ordinary stubs are more intuitive, easier to interpret and extend for programmers unfamiliar with `gen_statem`. We use `gen_statem` for our monitor template. We have dropped certain features present in [7] that do not correspond to TOAST, i.e., 'assert', 'require' and 'consume'.

Runtime Monitoring. Runtime monitoring [2, 3, 14, 15] is a form of dynamic verification that observes the behaviour of a program as it executes. An in-depth discussion can be found in [11]. Put briefly: (1) Runtime verification monitors analyse the trace of the program as it executes, and report any violations to the system. In Erlang/OTP, this could be a supervisor, which would then deploy a predefined recovery strategy. (2) Runtime adaptation monitors also detect violations, but instead will reactively attempt to change the behaviour of the monitored process to stop the error from occurring again. (3) Runtime enforcement monitors fully mediate the communication of the monitored process, acting on their behalf and ensuring no violation occurs. Our monitors support runtime verification and enforcement (1 & 3). Presented in [13] is a tool (based on [23]) for generating runtime monitors for Erlang from Scribble protocols [29], which is based on Multiparty Session Types [17] (MPST). Both our tool and [13] derive monitors from the theory of asynchronous session types, and utilise the supervision tree structure of Erlang for session coordination, instantiation and reporting violations. Our work only focuses on single binary sessions, [13] allows for multiple multiparty sessions.

Outside of Erlang, Scribble [20, 21, 25] has been used to generate runtime monitors for Python from MPST. As in our work, [20, 21, 25] encode session types into a CFSM, following the correspondence established in [12]. Notably, only [21] is in the timed setting, and presents runtime *enforcement* to our monitors similar to our own.

Outside of session types, [2] generates runtime monitors which are then inlined into a program via code injection at compile-time. Our work achieves inlining differently; by spawning the monitor within the same node the monitored process interacts with their monitor synchronously. Unlike [2, 5, 9, 10, 13, 20, 21, 25], we do not *generate* our monitors. Instead, our monitors are spawned from a single template, which either verifies or enforces the behaviour specified by an automata notation derived from session types.

While session types present systems in their theory for static type-checking, in practice this is often infeasible. For example, large, highly concurrent and often distributed systems, such as those in Erlang/OTP are typically unable to be statically checked due to the sheer scale of asynchronous interactions between processes. Since we are in the timed setting, dynamic verification remains the only feasible option, as we encounter a similar issue of checking interactions over a continuous period of time. For this reason, our work joins many others [5, 9, 10, 13, 20, 21, 25] which use session types for dynamic verification.

Code Generation & Session Types. Presented in [22] is a toolchain for generating F# code from Scribble protocols [29]. [22] offers a more immersive, intuitive and accessible 'interface' for their code generation, which takes the form of code snippet hints/suggestions of the next action to be performed.

This provides the programmer with 'live' feedback as they build their own implementation and insert the generated code snippets as they progress. By contrast, our tool is more straightforward by only producing an Erlang file for a programmer to extend. Naturally, our tool shares similarities with the one in [7]. Our tool uses an extended definition of the original protocol notation used to generate code, which in [7] was based off a calculus for designing protocols outside of session types. Beyond the generation of an FSM describing a given protocol, the process of code generation is fundamentally different, since our stubs do not follow Erlang/OTP `gen_statem` behaviour. Additionally, our tool produces stubs programs capable of being executed from the command-line almost immediately, allowing programmers to begin developing and testing the program. (See the project page [1].)

8 Conclusion

The Toolchain. Our tool [1] generates Erlang stub programs from an Erlang API that can be obtained via a mapping from TOAST processes [26, 27]. When obtaining an Erlang API from a TOAST process, we rely on the TOAST processes being well-typed against a well-formed TOAST protocol in order to ensure a good interaction structure, and certain guarantees of its behavioural properties. The generated programs can be configured to automatically start their own inline monitor, and when used within the '`sample_app`' directory in the project folder, are able to be run from the command-line immediately. (See the project `README.md`.)

Session Initiation. Sessions are started in an Erlang/OTP supervision tree, with a supervisor for each party. Each generated stub features the macro `?MONITORED` which if set `true` will cause the stub to automatically start their inline monitor, which by default acts as a transparent mediator.

Session Configuration. Since our monitors act as transparent mediators within a session, the data exchanged is unaffected, and either party does not have to accommodate if the other is or isn't monitored. Therefore, our monitors can be used in: (1) Fully symmetric sessions, where both parties are monitored and verified. (2) Asymmetric sessions, where only one party is monitored and verified. In the latter case, it would be the sole responsibility of the *un-monitored* party to adhere to all of the timing constraints of the protocol.

Limitations. Our toolchain follows the theory in [26, 27] and therefore, is limited to binary sessions. Additionally, in Erlang we cannot distinguish between inclusive ($\geq, \leq$) and exclusive (¡ ,) bounds featured in the time constraints of both TOAST protocols and processes.

Supported features. Table 1 illustrates the descriptive capabilities of TOAST protocols, TOAST processes and our Erlang API; where (✓) denotes supported, (×) unsupported and (◇) indicates the feature or pattern is *indirectly* supported.

Table 1. Interaction features & patterns in TOAST models.

Features & Patterns	TOAST [26] Protocols	TOAST [26] Processes	Erlang API
Timeouts	✓	✓	✓
Co-timeouts	✓	✓	✓
Selection	✓	×	✓
Safe Mixed-choice	✓	✓	✓
Unsafe Mixed-choice	×	×	×
Producer-Consumer	✓	✓	✓
Diagonal Constraints	✓	◇	◇
Complex Constraints	✓	◇	◇

In [26, 27], TOAST processes do *not* support output *selection* (i.e., the ability to *select* one sending action from several options). However, our Erlang API has not inherited this limitation and is fully capable of describing selection. Additionally, our toolchain is fully capable of generating Erlang stubs with selections and, in a similar fashion to non-blocking payloads, will automatically provide a means of making the selection in a non-blocking manner. For example, the protocol '!a(x 3, ∅).end' can be directly expressed with a *maximal implementation* TOAST process 'delay(t 3).p ◁ a.0', and defined using the Erlang API '{act, s_a, aft, 3000, error}'. While, for the protocol '{!b(true, ∅).S, !c(true, ∅).S'}' there are no TOAST processes that are maximal implementations, since TOAST processes require sending actions to be explicitly defined.

Supporting Complex Constraints. TOAST processes and our Erlang API can *indirectly* (◇) implement diagonal constraints, constraints with multiple upper-bounds and constraints with negation. Both TOAST processes and our Erlang API do not support such constraints to be defined directly. However, since we can rely on our TOAST process being well-typed against a well-formed TOAST protocol that *does* support these constraints, and may feature them, then we know that such a TOAST process (and the corresponding Erlang API) are guaranteed to adhere to these constraints, even if they themselves do not feature them. This can either be achieved: (a) by using a *non-maximal implementation* (i.e., where not all of the interactions actions are implemented into the process); or, (b) by making use of the time-sensitive structures in the processes, as shown in Section 6; or (c) by simplifying the time constraints to use fewer process timers.

Future Work. Our primary objectives in future work will be to: (a) aid users in designing TOAST protocols (b) semi-automate the process of extracting TOAST processes from TOAST protocols and (c) derive our monitor specification directly from TOAST protocols. We would also like to extend this tool with multiparty sessions, which would naturally follow the theory. Other work with session types [24] presents an alternative recovery strategy where the process, and all those causally related within the session, are reverted to an earlier point in the protocol, before the error occurred.

Exploring other recovery strategies for time-sensitive protocols would be interesting future work. Another consideration would take inspiration from the tool in [22], which instead of generating code to a file, provides the user with responsive feedback and suggestions. It would be crucial to consider how any functionality added by the programmer outside the scope of the protocol could affect the timings of interactions.

Benchmarks. In future work we hope to conduct a more thorough evaluation of our toolchain by comparing the performance of different generated implementations, the effectiveness, accuracy and performance overhead of our monitors and provide a wider range of examples and benchmarks.

Acknowledgments

We thank Simon Thompson for his insightful feedback on early versions of this work.

References

[1] 2024. Tool. https://github.com/jonahpears/Erlang-on-TOAST

[2] Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Léo Exibard, Adrian Francalanza, and Anna Ingólfsdóttir. 2024. A monitoring tool for linear-time μHML. *SCP* (2024), 103031. https://doi.org/10.1016/j.scico.2023.103031

[3] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. *Introduction to Runtime Verification*. Springer, 1–33. https://doi.org/10.1007/978-3-319-75632-5_1

[4] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2008. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR (LNCS)*. Springer, 418–433. https://doi.org/10.1007/978-3-540-85361-9_33

[5] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. 2013. Monitoring Networks through Multiparty Session Types. In *FTDS*. Springer, 50–65. https://doi.org/10.1007/978-3-642-38592-6_5

[6] Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2019. Asynchronous timed session types: from duality to time-sensitive processes. In *ESOP (LNCS)*. Springer, 583–610. https://doi.org/10.1007/978-3-030-17184-1_21

[7] Laura Bocchi, Dominic Orchard, and A. Laura Voinea. 2023. A Theory of Composing Protocols. *ACM* (2023). https://doi.org/10.22152/programming-journal.org/2023/7/6

[8] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. 2014. Timed Multiparty Session Types. In *CONCUR (LNCS)*. Springer, 419–434. https://doi.org/10.1007/978-3-662-44584-6_29

[9] Christian Bartolo Burlò, Adrian Francalanza, and Alceste Scalas. 2021. On the Monitorability of Session Types, in Theory and Practice. In *ECOOP (LIPIcs)*. Schloss Dagstuhl, 20:1–20:30. https://doi.org/10.4230/LIPIcs.ECOOP.2021.20

[10] Christian Bartolo Burlò, Adrian Francalanza, Alceste Scalas, Catia Trubiani, and Emilio Tuosto. 2022. PSTMonitor: Monitor synthesis from probabilistic session types. *SCP* (2022), 102847. https://doi.org/10.1016/j.scico.2022.102847

[11] Ian Cassar, Adrian Francalanza, Duncan Attard, Luca Aceto, and Anna Ingólfsdóttir. 2017. A Suite of Monitoring Tools for Erlang. In *RV-CuBES*. EasyChair, 41–47. https://doi.org/10.29007/7lrd

[12] Pierre-Malo Deniélou and Nobuko Yoshida. 2012. Multiparty Session Types Meet Communicating Automata. In *ESOP (LNCS)*. Springer, 194–213. https://doi.org/10.1007/978-3-642-28869-2_10

[13] Simon Fowler. 2016. An Erlang Implementation of Multiparty Session Actors. *EPTCS* (2016), 36–50. https://doi.org/10.4204/eptcs.223.3

[14] Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. 2017. A Foundation for Runtime Monitoring. In *RV*. Springer, 8–29. https://doi.org/10.1007/978-3-319-67531-2_2

[15] Adrian Francalanza, Jorge A. Pérez, and César Sánchez. 2018. *Runtime Verification for Decentralised and Distributed Systems*. Springer, 176–210. https://doi.org/10.1007/978-3-319-75632-5_6

[16] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP (LNCS)*. Springer, 122–138. https://doi.org/10.1007/BFB0053567

[17] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL*. ACM, 273–284. https://doi.org/10.1145/1328438.1328472

[18] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *HTOS*. ACM, 150–155. https://doi.org/10.1145/3102980.3103005

[19] Pavel Krcál and Wang Yi. 2006. Communicating Timed Automata: The More Synchronous, the More Difficult to Verify. In *CAV (LNCS)*. Springer, 249–262. https://doi.org/10.1007/11817963_24

[20] Rumyana Neykova. 2013. Session Types Go Dynamic or How to Verify Your Python Conversations. *EPTCS* (2013), 95–102. https://doi.org/10.4204/EPTCS.137.8

[21] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. 2017. Timed runtime monitoring for multiparty conversations. *FAC* (2017), 877–910. https://doi.org/10.1007/s00165-017-0420-8

[22] Rumyana Neykova, Raymond Hu, Noboku Yoshida, and Fahd Abdeljallal. 2018. A session type provider: compile-time API generation of distributed protocols with refinements in F#. In *CC*. ACM, 128–138. https://doi.org/10.1145/3178372.3179495

[23] Rumyana Neykova and Nobuko Yoshida. 2014. Multiparty Session Actors. In *COORDINATION (LNCS)*. Springer, 131–146. https://doi.org/10.1007/978-3-662-43376-8_9

[24] Rumyana Neykova and Nobuko Yoshida. 2017. Let it recover: multiparty protocol-induced recovery. In *CC*. ACM, 98–108. https://doi.org/10.1145/3033019.3033031

[25] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. 2013. SPY: Local Verification of Global Protocols. In *RV (LNCS)*. Springer, 358–363. https://doi.org/10.1007/978-3-642-40787-1_25

[26] Jonah Pears, Laura Bocchi, and Andy King. 2023. Safe asynchronous mixed-choice for timed interactions. In *COORDINATION (LNCS)*. Springer, 214–231. https://doi.org/10.1007/978-3-031-35361-1_12

[27] Jonah Pears, Laura Bocchi, Maurizio Murgia, and Andy King. 2024. Introducing TOAST: Safe Asynchronous Mixed-Choice For Timed Interactions. arXiv:2401.11197 https://arxiv.org/abs/2401.11197

[28] J. Peralta, P. Anussornnitisarn, and S. Y. Nof. 2003. Analysis of a time-out protocol and its applications in a single server environment. *IJCIM* (2003), 1–13. https://doi.org/10.1080/713804980

[29] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2014. The Scribble Protocol Language. In *TGC*. Springer, 22–41. https://doi.org/10.1007/978-3-319-05119-2_3

Received 2024-05-30; accepted 2024-06-27