

Dynamic Effective Timed Communication Systems

Tobias Heindel^a, Jonathan Prieto-Cubides^a, and Anthony Hart^a

^aHelix AG

* E-Mail: tobias@helix.dev, jonathan@helix.dev, anthony@helix.dev

Abstract

Message passing concurrency is a widely used paradigm in distributed systems research. Communicating finite state machines (CFSM) are probably the simplest model of message passing concurrency—introduced in the eighties, but still the default model in the context of communication protocols, session types, and choreographies. Another well-known family of models are Agha’s actors, which populate the other end of the expressivity and complexity spectrum: actors may behave in ways that even go beyond the computable. We want to avoid the complexities of the actor model and separate out matters that go beyond the computable. Ideally, we want something as simple as CFSMs to give semantics to engines of the Anoma specification but with adequate expressive power.

This paper thus introduces a generalization of CFSMs, called *dynamic effective timed communication systems* (DETCs)—somewhat baroque but descriptive: they have arbitrary computable state transitions, can dynamically create new state machines, and come equipped with a clock for each machine. We retain the isolated turn principle of the actor model. *Each machine performs “turns” one after the other: a turn is taking a waiting message, interpreting it, and deciding on both a state update for the machine and a collection of actions to take in response, perhaps sending messages or creating new machines.*¹ The technical core of the paper is the definition of DETCSs as their labelled transition systems, which can be used to give operational semantics to engine systems of the Anoma specification.

Keywords: Actor Model ; Distributed systems ; Time-stamped events ; Denotational semantics ; Temporal dependencies ; Enriched Event Diagrams ;

(Received: February 24, 2025; Version: March 6, 2025)

1. Introduction

Message passing concurrency is an established paradigm for modelling concurrent systems in order to reason about them with mathematical rigour. The basic idea of this paradigm is to put all agency and state into PROCESSES, which then communicate by exchanging messages with each other. The advantage is homogeneity, which is a bonus, in particular for *reasoning* about models. However, on the flip side, as all processes are created equal, it is potentially hard to distinguish between the purely computational aspects of a process and effects caused by agents operating under the guise of processes.

In this paper, we present a generalization of communicating finite state machines (CFSM) that enables controlled interaction with human operators

¹The wording is taken from [GJ16], replacing “actor” with “machine”.

and external agents while preserving the simplicity of message passing concurrency, as detailed in [Section 4.2](#). To the best of our knowledge, existing CFSM-based frameworks exhibit two key limitations for modelling real-world distributed protocols. First, current communication state-based systems models lack support for dynamic process creation —a limitation we address in [Section 4.2](#). Second, they fail to properly incorporate clock-based timing mechanisms, as discussed in [Section 4.3](#). The development of this work complements the ongoing development of the Anoma protocol [[Co24](#)].

In more detail, the paper generalizes CFSMs in three ways: the state transition functions are arbitrary computable functions between countable sets; the number of communicating state machines is finite but may change dynamically, see [Section 4.2](#); finally, transitions are timestamped by local clocks of the machines, see [Section 4.3](#). We dub these systems *dynamic effective timed communication systems* since they are a generalization of the *communication systems* of [[BLT20](#)], state transitions are *effective* in the sense of computability theory, the number of process is *dynamic*, and we have local clocks to assign *timestamps* to incoming messages. In [Section 5](#), we describe the main ideas of how we can obtain an interactive version, such that users can make decisions that need not be deterministic or computable from previous messages and the local state.

Concerning related work, the connection to communicating finite state machines is already covered by the above description. The main conceptual difference to the actor model [[Agh86b](#)] is the separation of the purely computational aspects of actors and other sources of agency. Machines take care of the computational parts, but everything else is pushed outside the system; the system interacts with the environment via channels, for which it may be natural to assume stronger synchrony assumptions than the default assumptions of distributed systems (namely, asynchrony or partial synchrony).

The remainder of this paper is structured as follows. We first revise the background material on message passing concurrency and review the Token Ring Protocol (standardized as [IEEE 802.5](#)), which will serve as our running example in [Section 2](#). Then we describe conventions of notation and the relevant elements of computability theory in [Section 3](#). After that, we come to the technical core of the paper, culminating in the definition of DETCSS and their labelled transition semantics (see [Definition 12](#)), in [Section 4](#). The mechanism for channels through which the system can be affected by external agents is described in [Section 5](#). Finally, we discuss related work and conclude.

2. Background: communicating processes and protocols

The main protagonist of the paper is a model of computation² for processes with *local clocks*, which may communicate according to some protocol—or not.³ The purpose of the computational model is to enable reasoning about such sets of timed processes. Our approach is based on state machines, a common basic abstraction in computer science. The slogan is: *the simpler the model, the better for protocol design!*

We thus review the most important context before defining the model itself: we start with the definition of the term “protocol”, followed by some simple examples for protocol implementation that involve communicating processes—directly or implicitly.

2.1. Protocol generalities: communication protocol definition

We take the following text-book definition of “protocol” [Her20] as a reference point:

a set of rules which determine how two or more entities should communicate is called a *communication protocol*,

or PROTOCOL for short. The point we want to emphasize is the normative aspects of protocols: protocols are about the admissible sets of messages that each participant *may* send at a given point in time and they describe which messages a participant *should* accept under which circumstances and act accordingly. Let us look at the Anoma Protocol [Co24] for an example that illustrates the normative character of protocols: one desired property is that each user request for matching an intent or settlement of a transaction *should* be considered by the collective of operators of the Anoma instance (leading to matching or settlement with adequate speed, if possible). However, in the present paper, we are mainly concerned with the *actual* communication that entities engage in; the reason is that we first want to agree on a framework for how to reason about implementation candidates for a protocol before we start addressing the question of how we can check that a set of observed message exchanges adheres to the protocol.

2.2. Protocol generalities: protocol instances

In the present paper, we want to distinguish between protocols, specifying *desired* communication patterns, and something else that concerns the *actual*

²Please be assured, the authors would rather take a model of computation off the shelf instead of going through the ordeal of comparing with the literature (see also Section 6). There simply seems no model out there that strikes a useful compromise between the simplicity of communicating (finite) state machines and the expressive power of the actor model, while providing a local wall-clock time for each protocol participant. In particular, there seems to be no good match with any the four categories proposed in Ref. [DKVCDM16], especially when it comes to user interactions as described in Section 5.

³We discuss protocols informally in Section 2.1. Let us point out here, once more, that the question of what a protocol is “exactly” is out of scope.

communications performed. Let us formulate a candidate definition:

A **PROTOCOL INSTANCE** is the actual message passing of processes that conforms to a protocol.

(1)

The main variations of such a notion of protocol instance concern the nature of message passing and the assumptions about the participating processes. Concerning message passing, we shall follow the usual paradigm of eventual delivery of messages; concerning processes, we aim for a variation on the theme of actors, but restricted in suitable ways so that they can be implemented on a general purpose computer. The model of computation that we describe takes communicating finite state machines [BZ83] as a starting point for a generalization. There are also similarities and indirect influences from the co-algebraic description of object oriented systems [Jac95]. However, the present paper aims to keep matters as basic as possible. Matters of fault tolerance will be covered in future versions of the paper.

2.3. Communicating processes: abstract and concrete descriptions

A communicating process, not necessarily in the context of a protocol, can be described abstractly as a function that takes a *stream of inputs* and produces a *stream of outputs*. In the case of communicating finite state machines, this conversion is described concretely, using state updates, transforming inputs to outputs, one letter at a time⁴ However, abstractly, the communication between the processes *described by* a set of communicating finite state machines arises by the suitable connection of input streams to output streams. In the case of CFSMs, this “wiring” between processes is fixed in advance and stays forever. In distributed systems research, it is common to abstract away the details of how output streams relate to input streams through a generic communication network.

2.4. Last but not least: the communication network

How does a message “travel” between the communicating processes? The idea is to refrain from answering this question! We assume a generic asynchronous communication network, as is common in distributed systems research. The only assumption is that each message that is sent will be receivable at the destination, eventually. In other words, each message that can be read from an output stream will eventually be processed as part of the input stream of the process that is the destination of the message.

In the original work on communicating finite state machines [BZ83], the network was required to preserve the order of messages, *i.e.*, the network

⁴ A related notion is that of transducer in automata theory. Also note that this very much fits the idea of the *Isolated Turn Principle* [DKVCDM16].

could be described as a clique of pair-wise FIFO channels. We drop the assumption that message order is preserved, which is the usual assumption in distributed systems. We only postulate that the network routes one message at a time, taking a message out of one of the output streams and delivering it to the corresponding input stream, *e.g.*, by pushing it to the recipient's “inbox”-queue.⁵

2.5. Example protocols: token-ring variations

A common family of examples for illustrating communication protocols is based on *The Token Ring Protocol*, standardized as [IEEE 802.5](#). This protocol assumes a fixed set of processes that are organized in a ring topology. The

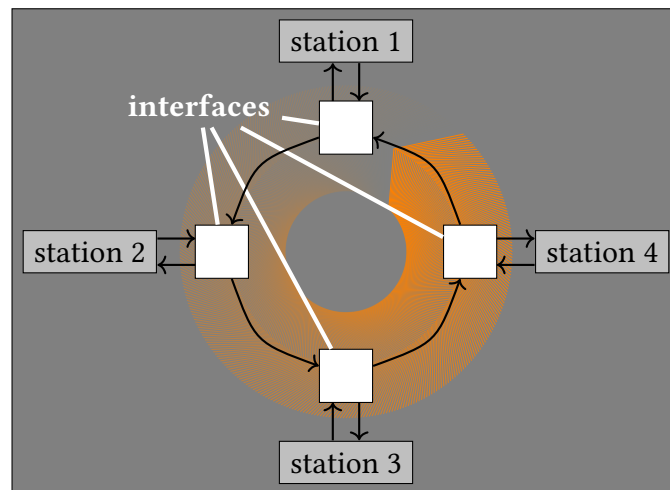


Figure 1. Token ring illustration (based on [\[MV93\]](#)).

main idea of the Token Ring Protocol is to allow every process to send, from time to time, a data payload to every other process in a fixed ring topology, disseminating the data one hop at a time around the ring until it arrives again at the sender (which then can check that the message was transmitted correctly); in the terminology of this protocol, this is a *data frame* (see also [\[MV93\]](#)).

When a bit arrives at a ring-interface it is copied into a 1-bit buffer and then put into the ring again. While in the buffer, the bit can be inspected and possibly modified before being written out into the ring again.

To this end, an “idle”-token is circulated until some participant in the ring stops circulating the “idle”-token and starts transmitting a message.

⁵The case of dynamic creation of new processes is more challenging in that we essentially have to extend processes with streams of spawning requests and an “operator” that actually creates the system.

2.6. Miscellaneous notes to the reader

In this paper, we only consider *flat* protocols. Aspects of non-trivial protocol structure are left for future work. We are agnostic to how the rules of communications are given in a protocol, simply because the question of correctness of protocol instantiation are out of scope.

One guiding motive of the paper is the question of how we can avoid the *yet another programming language*-trap and keep a direct relation to communicating finite state machines [BZ83] (see also [Appendix A](#)). This will in particular concern the setting where processes are supposed to spawn new processes. We do not propose a programming language for this.

3. Notation and conventions

In definitions, we use the symbol $:=$ to introduce new terminology and/or notation: the symbols to the left are the ones that are to be defined in terms of symbols to the right. Generally speaking, we follow established conventions of the computer science community, *e.g.*, \mathbb{N} is the set of natural numbers including zero. For the sake of clarity, we recall notation and conventions that we use in the paper.

Capital letters denote sets unless stated otherwise. Thus, the letters A and B stand for sets (making no claim about whether A and B are equal). Set inclusion is denoted by \subseteq , *i.e.*, we write $A \subseteq B$ when every element of A belongs to B ; the empty set is denoted by \emptyset . The Cartesian product of a pair of sets A, B is denoted by $A \times B$, and ordered pairs of elements a, b are written $\langle a, b \rangle$, *i.e.*, $\langle a, b \rangle \in A \times B$ if, and only if, $a \in A$ and $b \in B$. The zeroary Cartesian product is denoted by $1 = \{\langle \rangle\}$. Given two sets A, B , their union is $A \cup B$ and their disjoint union is $A + B := A \times \{0\} \cup A \times \{1\}$. Two disjoint sets A and B are said to be *disjoint* if $A \cap B = \emptyset$. The cardinality of a set A is denoted by $\#(A)$. The power set of a set A is denoted by $\text{Powerset}(A)$; the set of all finite subsets of a set A is $\text{Powerset}_{fn}(A)$, *i.e.*,

$$\text{Powerset}_{fn}(A) := \{L \subseteq A \mid \#(L) \in \mathbb{N}\}.$$

A **RELATION** between sets A and B is a subset of the Cartesian product $A \times B$ in which A is called the **DOMAIN** of the relation and B its **CODOMAIN**. A relation $R \subseteq A \times B$ is a **PARTIAL MAP** when for each element $a \in A$, there is at most one element $b \in B$ such that $\langle a, b \rangle \in R$; we write $\varphi: A \rightharpoonup B$, when φ is a partial map from A to B . A partial map $\varphi: A \rightharpoonup B$ is defined for an element $a \in A$ when $\langle a, b \rangle \in \varphi$ for some element $b \in B$; whenever φ is defined for $a \in A$, then $\varphi(a)$ denotes the unique element of B such that $\langle a, \varphi(a) \rangle \in \varphi$. The **DOMAIN OF DEFINITION** of a partial map is denoted by $\text{df}(\varphi)$, *i.e.*, $\text{df}(\varphi) := \{a \in A \mid \exists b \in B. \langle a, b \rangle \in \varphi\}$. We write $\varphi: A \dot{\rightharpoonup} B$ when φ is a partial map from A to B with finite domain of definition. A **FUNCTION**

is a partial map whose domain of definition coincides with its entire domain; we write $f: A \rightarrow B$ when f is a function with domain A and codomain B . The set of all functions from A to B is B^A , *i.e.*,

$$B^A := \{f \in \text{Powerset}(A \times B) \mid f: A \rightarrow B\}.$$

We use postfix notation for the projection function of each product of a Cartesian product, *i.e.*, $-_1: A \times B \rightarrow A$ $-_2: A \times B \rightarrow B$, and thus, we have $\langle a, b \rangle_1 = a$ and $\langle a, b \rangle_2 = b$, for every $\langle a, b \rangle \in A \times B$.

Implementability via computability theory. We want to make sure that the machines that we shall introduce (see [Definition 1](#)) are actually implementable. For this, we appeal to the usual apparatus of computability theory [[Tur37b](#), [RJ67](#)].⁶ A PARTIAL RECURSIVE FUNCTION between sets A and B is a partial map $\varphi: A \rightarrow B$ that can be implemented by a Turing machine [[Tur37b](#)], a lambda calculus term [[Tur37a](#)], or a program that is written in the reader's favourite programming language as long as it takes elements of A as input and either outputs elements of B if the input was part of the domain of definition or runs forever if it was not part of the domain of definition.

For every countable set A , we assume a “reasonable”⁷ injective function $[_]: A \rightarrow \mathbb{N}$, called the **ENCODING** of the set A . We denote the image of the set A as subset of the natural numbers as $[A] \subseteq \mathbb{N}$. Sometimes, it is important that the set $[A]$ is *decidable*, *i.e.*, it must be decidable whether an element $x \in \mathbb{N}$ belongs to $[A]$ (and not to $\mathbb{N} \setminus [A]$), which, in turn, means that there exists a partial computable function $\chi: \mathbb{N} \rightarrow \{0, 1\}$ that computes 1 on input x iff $x \in [A]$. Finally, we are using the fact that we can enumerate computable functions between sets in a suitable way, known as admissible indexing [[RJ67](#), Example 2-10], *i.e.*, a numbering of computable functions that allows to represent every computable function that moreover satisfies Kleene's S_n^m -theorem [[Kle52](#)].

4. Communication systems of effective Moore machines

In this section, we generalize communicating finite state machines [[BZ83](#)] in three steps:

1. We define a variation of Moore's sequential machines [[Moo56](#)];
2. We add a mechanism for spawning new machines, dynamically;

⁶ These matters are technically important, but may be skipped safely on a first reading.

⁷ We want to elide the details of how elements of sets are represented as natural numbers. Unfortunately, for definitions to be well-defined, we must encode inputs and outputs to make them machine readable. Things become easier if we simply assume that every relevant set actually is a subset of the natural numbers; then, we do not need to worry about details of encoding sets. If that is too drastic for the reader's taste, we can equip each set A with an encoding that is computable and total, using a definition of computability in terms of Turing machines [[HMu01](#)] and the usual binary encoding of the natural numbers.

3. We equip each machine with its own clock.

We choose Moore machines as our state machine model instead of Mealy machines because they simplify the semantics. With Moore machines, one can clearly separate a machine’s internal state transitions from its overall effects that it has on the system, making the definition of the basic labelled transition semantics much more straightforward (see [Definition 8](#)).⁸

Sequential machines à la Moore, but with computable behaviour. Moore’s description of sequential machines [[Moo56](#)] contains an explanation of what he means by *deterministic behaviour*, namely

that the present state of a machine depends only on its previous input and previous state, and the present output depends only on the present state.

We want to keep this description, but remove the restriction to finite sets that Moore imposes when writing [[Moo56](#)] that the

sequential machines considered have a finite number of states, a finite number of possible input symbols, and a finite number of possible output symbols.

This paper aims for the most straightforward generalization of Moore’s sequential machines whose behaviour is *computable*⁹.

Dynamic creation of new machines. We also want to model the dynamic creation of new machines as a new “action primitive”, in addition to the usual sending and receiving of messages in protocol instances (*cf. transmissions and receptions*, resp. [[BZ83](#)]). Hence, we augment the output set of our generalized Moore machines so that machines can make requests to spawn new machines, much like how a process can spawn child processes. This feature is directly analogous to actor creation [[Agh86b](#)]. Spawning a new machine requires a piece of “source code”¹⁰ that describes the behaviour of the new machine, along with an “address” at which the machine will be available (see [Definition 6](#) for more details). These “addresses” will be called *participant IDs* (PID), mainly to make the relation to the terminology of choreography automata [[BLT20](#)] evident.¹¹

⁸We describe in [Section 5](#) how user interactions can be incorporated into the Moore-esque approach.

⁹We shall use the usual mathematical definitions of partial recursive functions [[Com](#)] that has become established to capture the intuition of what is effectively calculable.

¹⁰In ABCL/1, these source code snippets are called *scripts*.

¹¹Incidentally, it is “not entirely wrong” to read PID as *process ID*, which is used to reference an Erlang/Elixir process.

Wall-clock timestamps for every message. The third and final step consists of adding a clock for every participating machine. In more detail, each incoming message is stamped at the moment of reception with the receiving machine’s wall-clock time. Using local timestamps, we can implement busy waiting¹² (by sending messages to self). More interestingly, wall-clock time-stamps can be used to measure latency, round-trip times, and the like.

4.1. Moore-esque machines with computable behaviour

We first define a generalization of Moore’s sequential machine that matches his description of deterministic behaviour [Moo56]; then, we follow up with the desired restriction to computable behaviour, which we dub *effective Moore machines* (see Definition 3).

Definition 1 (Moore-esque machine). Relative to two fixed sets Σ and Γ , a MOORE-ESQUE MACHINE with input alphabet Σ and output alphabet Γ is a quadruple $\langle Q, q_0, \delta, \mu \rangle$, consisting of

- a set of STATES Q ,
- a CURRENT STATE $q_0 \in Q$,
- a STATE TRANSITION MAP $\delta: Q \times \Sigma \rightarrow Q$, and
- an OUTPUT MAP $\mu: Q \rightarrow \Gamma$.

For any Moore-esque machine m , we denote its state set, current state, transition map, and output map by States_m , s_m , in_m , and out_m , respectively. Finally, given a Moore-esque machine $m = \langle Q, q_0, \delta, \mu \rangle$ and a state $q \in Q$, we define the STATE SUBSTITUTION

$$\langle Q, q_0, \delta, \mu \rangle @ q := \langle Q, q, \delta, \mu \rangle,$$

which reads “ m at state q .”

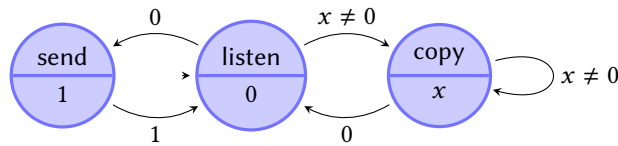


Figure 2. A Moore-esque machine

¹²A “native” timer mechanism is described in Appendix B, which comes at the price of somewhat complex notation and lengthy definitions.

Example 2. We depict an example of a Moore-esque machine using the usual graphical representation in [Figure 2](#). The upper half of each node represents a state, *i.e.*, the set of states is $\{\text{send}, \text{listen}, \text{copy}\}$. The lower half of each node is the result of applying the output map to the upper half, *e.g.*, when transitioning into the send state, the output 1 is generated. Finally, as is customary for all finite state machines, the result of applying the state transition map to a node u with outgoing edge labelled x is the target v of the edge, *i.e.*, the pair $\langle u, x \rangle$ is mapped to v . Thus, the transition map is the following.

$$\begin{aligned} \langle \text{send}, 1 \rangle &\mapsto \text{listen} \\ \langle \text{listen}, 0 \rangle &\mapsto \text{send} \\ \langle \text{listen}, x \rangle &\mapsto \text{copy if } x \neq 0 \\ \langle \text{copy}, 0 \rangle &\mapsto \text{listen} \\ \langle \text{copy}, x \rangle &\mapsto \text{listen if } x \neq 0 \end{aligned}$$

The notion of Moore-esque machine goes far beyond the finite state case. In order to be just general enough to reason about actual implementations in existing programming languages, we restrict to machines whose behaviour is computable.

Definition 3 (Effective Moore machine).¹³ Let Σ and Γ , be sets. An EFFECTIVE MOORE MACHINE (EMM) with inputs from Σ and outputs in Γ is a Moore-esque machine with inputs from Σ and outputs in Γ such that the transition and the output map are both *computable*. The set of all effective Moore machines with inputs in (a subset of) Σ and outputs in (a subset of) Γ is denoted by $\text{EMM}(\Sigma, \Gamma)$; its elements are called Σ - Γ -EMMs.¹⁴

Note that the transition and output maps of EMMs are computable, but not necessarily total. Also note that input and output alphabets may have non-trivial structure (while the notation is chosen to make the connection to Moore machines crystal clear).

We now describe the idea of *communicating systems* [[BLT20](#), Definition 2.4] in terms of EMMs. We fix two at most countable, decidable sets \mathcal{M} and \mathcal{P} , containing *messages* and *participant IDs* (PIDs), respectively.¹⁵ We range over these two sets by $m, n, \dots \in \mathcal{M}$, and $A, B, \dots \in \mathcal{P}$, respectively; we assume messages and PIDs to be disjoint. With these two sets at hand, we define sets of RECEPTION and TRANSMISSION LABELS for each PID, denoted by \mathcal{R}^A and \mathcal{T}^A , respectively. Elements of \mathcal{R}^A and \mathcal{T}^A are called A -receptions and A -transmissions, respectively.

¹³ Here, we are following the *mathematical Red Herring principle*: an effective Moore machine need not be a Moore machine; moreover, every Moore machine is actually effective.

¹⁴ The notation $\text{EMM}(\Sigma, \Gamma)$ hints at the possibility to regard Moore machines as transducers with inputs in Σ and outputs in Γ .

¹⁵ As in [[BLT20](#)], we do not impose any additional restrictions on the nature of \mathcal{M} and \mathcal{P} (and we do not even need the restriction that $\mathcal{M} \cap \mathcal{P} = \emptyset$); we do however add a cardinality restriction.

$$\mathcal{R}^B := \left\{ AB?_m \mid \begin{array}{l} A \in \mathcal{P} \\ m \in \mathcal{M} \end{array} \right\} \quad \text{where } AB?_m = \langle A, B, m \rangle. \quad (\text{B-reception})$$

$$\mathcal{T}^A := \left\{ AB!_m \mid \begin{array}{l} B \in \mathcal{P} \\ m \in \mathcal{M} \end{array} \right\} \quad \text{where } AB!_m = \langle m, A, B \rangle. \quad (\text{A-transmission})$$

Note that the sets \mathcal{R}^A and \mathcal{T}^B are always disjoint, even if $A = B$ holds. Then, in direct analogy to communicating systems [BLT20, Definition 2.4], we define effective communicating systems.

Definition 4 (Effective communicating system). An EFFECTIVE COMMUNICATING SYSTEM (ECS) is a partial map $S: \mathcal{P} \rightharpoonup \text{EMM}$ from PIDs to EMMs with finite domain of definition such that, for each PID $A \in \mathcal{P}$ for which the partial map S is defined, the corresponding EMM $S(A)$ consumes A -receptions and produces finite sets of A -transmissions, i.e., for all $A \in S(A)$,

$$S(A) \in \text{EMM} \left(\mathcal{R}^A, \text{Powerset}_{fin}(\mathcal{T}^A) \right).$$

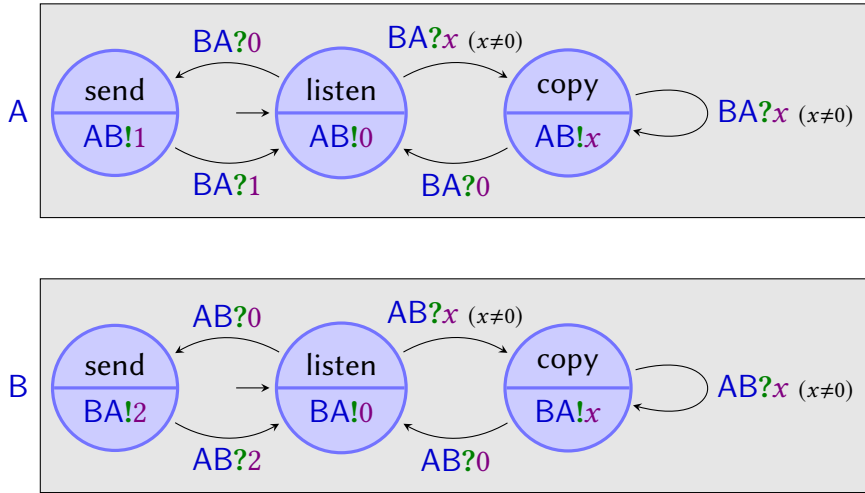


Figure 3. An effective communicating system with two Moore-esque machines A and B

Example 5 (Effective communicating system). A simple example of an effective communicating system is given in Figure 3. Intuitively, messages are sent back and forth between the machines A and B , consuming each others outputs and responding accordingly. This can be taken to be a simplified version of a Token Ring with two participants that alternate between sending their payloads, namely the numbers 1 and 2, and circulating the token;

the token is simplified to be the number 0. Note that the token is performing “idle” runs around the ring. One could move the target of the transition edge from the copy state to go directly to the send state, without waiting a round in the listen state.¹⁶

The formal details of the LTS-semantics of ECSS are essentially the same as the ones for communicating systems [BLT20, Definition 2.4], except for differences in the network model. We defer the details to after we have introduced the dynamic version.

4.2. Creating new machines dynamically

We now complement the reception and transmission labels with spawning labels, which then will be used to extend the set of “active” PIDs dynamically. Each dynamic spawning request must provide a piece of information that *encodes* the behaviour of an EMM (alongside a new “address”). Concerning encodings of behaviours, we resort to the established theory for enumerating computable functions (either Gödel numbering [Göd31] or any other admissible indexing [RJ67] as described in Section 2).

$$\mathcal{N}^A := \{An^*B \mid B \in \mathcal{P}, n \in \mathbb{N}\} \quad \text{where } An^*B = \langle A, n, B \rangle \quad (2)$$

As a result, we obtain *interpretations* of the spawning labels in Equation (2) as EMMs. For this, we fix an admissible numbering¹⁷ [RJ67] of EMMs with inputs and outputs in \mathcal{R}^A and $\text{Powerset}_{fin}(\mathcal{T}^A \cup \mathcal{N}^A)$, respectively; this numbering is parameterized over A , *i.e.*, different participants may have different a different enumeration of machines. We write $\llbracket n \rrbracket_A$ for the n -th \mathcal{R}^A - $\text{Powerset}_{fin}(\mathcal{T}^A \cup \mathcal{N}^A)$ -EMM, *i.e.*, we have

$$\text{EMM} \left(\mathcal{R}^A, \text{Powerset}_{fin}(\mathcal{T}^A \cup \mathcal{N}^A) \right) = \left\{ \llbracket n \rrbracket_A \mid n \in \mathbb{N} \right\}.$$

With this encoding function to hand, we now generalize communicating systems to dynamic ones whose PIDs are mapped to EMMs.

Definition 6 (Dynamic effective communicating system). A DYNAMIC EFFECTIVE COMMUNICATING SYSTEM (DECS) is a partial map $S: \mathcal{P} \dashrightarrow \text{EMM}$ from PIDs to EMMs such that for each PID for which the partial map S is defined, the EMM $S(A)$ consumes A -receptions and produces finite sets of A -transmission and A -spawning requests, *i.e.*,

$$S(A) \in \text{EMM} \left(\mathcal{R}^A, \text{Powerset}_{fin}(\mathcal{T}^A \cup \mathcal{N}^A) \right)$$

holds for each PID $A \in \text{df}(S)$. A PID $A \in \mathcal{P}$ is ACTIVATED in a DECS S when it belongs to the domain of definition of the partial map S , *i.e.*, when $A \in \text{df}(S)$.

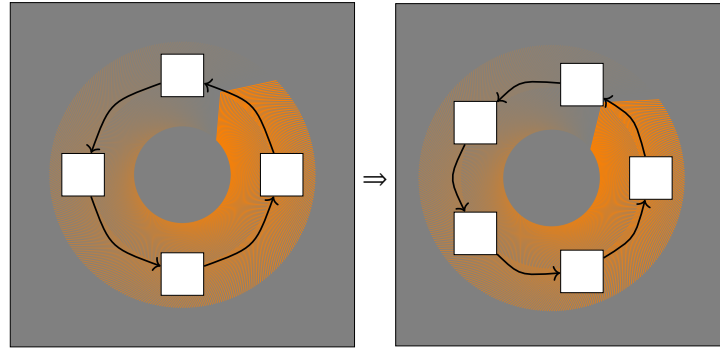
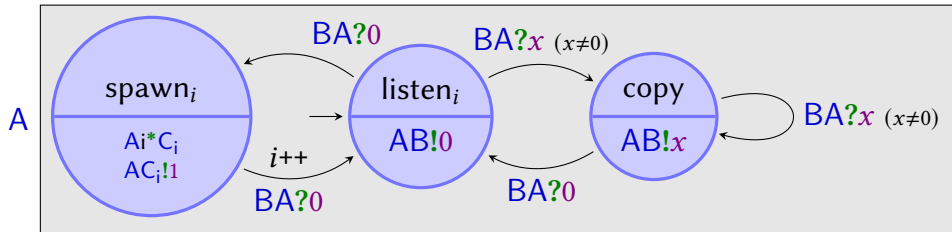


Figure 4. Dynamic Token Ring expansion.

The feature of dynamic spawning allows for growing rings. For example, we could grow a ring of dining philosophers (see [BCK01]). However, let us stay with our running example, the Token Ring. The idea of a growing the ring is illustrated in Figure 4. The following example spells out some of the detail in terms of DECSS.

Example 7 (Making the Token Ring grow). The feature of spawning new machines allows one to dynamically extend the number of participants in Figure 3. Instead of grabbing the token and sending the payload, a new participant could be spawned. So, suppose we want to adapt machine *A* in Figure 3 along these lines. The idea is to replace the machine at *A* by a machine that spawns an unbounded number of *copies* of the *A*, each encoded as $\llbracket i \rrbracket_A$ to be residing at address C_i . The first copy at C_0 would send to *B*, and all later copies C_{i+1} will send to C_i .



The machine *A* would look as above. In more detail, the machine at *A* is not sending any payload any more, but spawns a new machine instead. To keep track of how many machines it has already spawned, its state has a counter *i*—recall that we are allowed to have a countable number of states. The machine that is spawned at C_i will behave very much like the machine *A* in Figure 3, except for that, depending on the value of the counter *i*, it will send messages to C_{i-1} (if *i* > 0) or to *B* (if *i* = 0). Note that after spawning, machine *A* waits for the token from *B*. Also note that *A* is sending an “init”

¹⁶The presented version yields a nicer graphical presentation.

¹⁷See 2 for more details.

message $AC_i!1$, even before any new machine is spawned. The behaviour of the new machine C_i is such that it will send the token 0 along the ring to keep the Token Ring operational (as response to $AC_i!1$).

We now formalize the intended dynamics of DECSs. In the present paper, we want to keep track of communications “inside” the system (all activity is visible, even though we are focusing on “closed” systems). We keep track only of message *receptions* in the operational semantics.

Definition 8 (Labelled transition system of a DECS). A DECS-STATE is defined as a pair $\langle S, M \rangle$, consisting of a DECS S and a finite set of transmissions $M \in \text{Powerset}_{fin}(\bigcup_{A \in \text{df}(S)} \mathcal{T}^A)$. Given a DECS-state $\langle S, M \rangle$ and a transmission $AB!m \in M$, the $AB!m$ -SUCCESSOR of the DECS-state $\langle S, M \rangle$ is the DECS-state $\langle \hat{S}, \hat{M} \rangle$ where

$$\hat{S}(C) = \begin{cases} S(B) @ \text{in}_{S(B)}(\langle s_{S(B)}, AB?m \rangle) & \text{if } C = B, \\ S(C) & \text{if } C \in \text{df}(S) \setminus \{B\}, \\ \llbracket n \rrbracket_C & \text{if } Bn^*C \in \text{out}_{\hat{S}(B)}(s_{\hat{S}(B)}) \\ & \text{and } C \notin \text{df}(S), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$\hat{M} = (M \setminus \{AB!m\}) \cup (\text{out}_{\hat{S}(B)}(s_{\hat{S}(B)}) \cap \mathcal{T}^B),$$

i.e., the recipient B of the message $AB!m$ will transition to the next state according to its state transition map, all other existing participants remain unchanged, new machines will be spawned as specified by the output of the next state of participant B , and all (other) undefined participants remain undefined; note that this definition includes the case in which the reception of the message leads to termination/crash of B , modelled by the state transition map being undefined.

A LABELLED TRANSITION between DECS-states $\langle S, M \rangle$ and $\langle \hat{S}, \hat{M} \rangle$ is a triple

$$\langle \langle S, M \rangle, AB!m, \langle \hat{S}, \hat{M} \rangle \rangle$$

such that $\langle \hat{S}, \hat{M} \rangle$ is the $AB!m$ -successor of $\langle S, M \rangle$. The LTS of EMMs is the pair $\langle X, \rightarrow \rangle$, whose first component X is the set of DECS-states, and whose second component \rightarrow is the set of all labelled transitions between DECS-states; we write

$$\langle S, M \rangle \xrightarrow{AB!m} \langle \hat{S}, \hat{M} \rangle$$

when $\langle \langle S, M \rangle, AB!m, \langle \hat{S}, \hat{M} \rangle \rangle \in \rightarrow$.

Note that each DECS induces a DECS-state by pairing it with the union of the set of messages that any of the Moore machines (in the range of the DECS) output in their current state.

Remark 9 (On machine termination detection). The partiality in partial recursive functions often is directly related to non-termination of the underlying computation. Thus, in actual implementations, one may want to employ techniques that avoid machines that are indefinitely delayed. One work around is to restrict attention to machines whose behaviour is guaranteed to terminate, by use of type systems that guarantee termination at the expense of expressive power.¹⁸ There are even programming languages such that the type system makes sure that functions are terminating within certain resource bounds [Gir94]. There is an obvious relation to crash failure because non-terminating machines could just be crashed. Thus matters of termination checking are out of scope of the present paper.

4.3. Equipping communicating systems with clocks

Equipping EMMs in a DECS with local clocks amounts to adding a partial map of clocks to DECS-state. However, the access to the clock is indirect: the current local time is taken at each message reception.

Definition 10 (Dynamic effective timed communicating system). A DYNAMIC EFFECTIVE TIMED COMMUNICATING SYSTEM (DETCES) is a pair $\langle S, W \rangle$, consisting of

- a partial map $S: \mathcal{P} \rightharpoonup \text{EMM}$ from PIDS to EMMs such that

$$S(A) \in \text{EMM} \left(\mathbb{N} \times \mathcal{R}^A, \text{Powerset}_{fn}(\mathcal{T}^A \cup \mathcal{N}^A) \right)$$

holds for each PID A in the domain of definition $\text{df}(S)$, which, in turn, we call the set of ACTIVATED PARTICIPANTS,

- a partial map $W: \mathcal{P} \rightharpoonup \mathbb{N}$, which assigns the WALL-CLOCK TIME to every activated participant.

Using local clocks, we can calculate the observed round-trip time, relative to each clock. The idea is that we send a short test payload; we can use the so far observed average time for this. The idea, in more detail, is as follows.

Example 11 (Average round-trip time in a Token Ring). We can adapt the two machine example from Figure 3 to calculate an average round trip time. For this, we add variables for the average delay Δ (initialized to 0), the number of previous measurements k (initialized to 0), and memory for time t at which the token was received most recently.

¹⁸Unfortunately, there is no computable enumeration of all total computable functions.

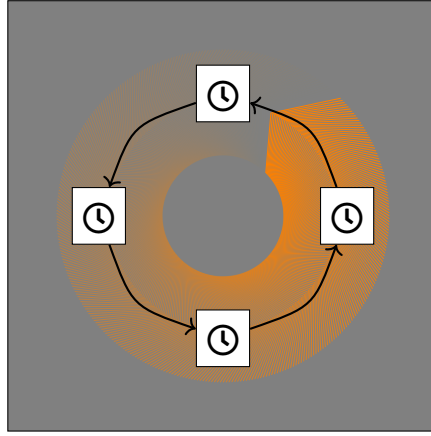
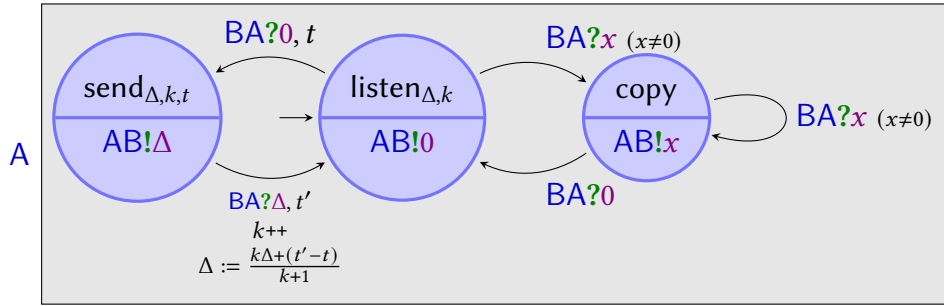


Figure 5. Wall-clocks at each station.



However, there is no guarantee that the average round-trip time will converge, since we are considering a fully asynchronous setting. Therefore, without additional assumptions about the “delivery speed” of the network, we cannot make any guesses about the relative speeds of clocks, since the network may be “infinitely slow” or slowing down due to a growing number of participants.

We now extend the labelled transition semantics of DECSS under modest assumptions about the network and clocks, namely, that every message will be available for reception and that messages take at least one clock cycle to be received. The only difference in message reception is the additional timestamp, which the receiving machine may take into consideration (or ignore). Note that we must use an adapted machine encoding $\llbracket n \rrbracket_A''$ that handles the changes in the input and output alphabets of EMMS .

Definition 12 (Labelled transition system of a DETCS). A DETCS -STATE is defined as a triple $\langle S, W, M \rangle$, such that the first two components together form a DETCS and the last component is a finite set of transmissions $M \in \text{Powerset}_{fin}(\bigcup_{A \in \text{df}(S)} \mathcal{T}^A)$. Given a DETCS -state $\langle S, W, M \rangle$, a transmission $AB!m \in M$, and a strictly positive delay $0 < t \in \mathbb{N}$, the $AB!m$ - t -SUCCESSOR

of $\langle S, W, M \rangle$ is the DETCS-state $\langle \hat{S}, \hat{W}, \hat{M} \rangle$ where

$$\hat{S}(C) = \begin{cases} S(B) @ \text{in}_{S(B)} \left(\langle s_{S(B)}, \langle t', AB!_m \rangle \rangle \right) & \text{if } C = B, \\ \quad \text{where } t' = W(B) + t & \\ S(C) & \text{if } C \in \text{df}(S) \setminus \{B\}, \\ \llbracket n \rrbracket_C'' & \text{if } Bn^*C \in \text{out}_{\hat{S}(B)}(s_{\hat{S}(B)}) \\ & \text{and } C \notin \text{df}(S), \\ \text{undefined} & \text{otherwise} \end{cases}$$

—this is in direct analogy to the definition for the case without clocks, but augmented by the “time stamp” t' of the newly arrived message—and we have the following definition for updating clocks:

$$\hat{W}(C) = \begin{cases} W(B) + t & \text{if } B = C \\ W(C) & \text{if } C \in \text{df}(S) \setminus \{B\}, \\ 0 & \text{if } Bn^*C \in \text{out}_{\hat{S}(B)}(s_{\hat{S}(B)}) \\ & \text{and } C \notin \text{df}(S). \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$\hat{M} = (M \setminus \{AB!_m\}) \cup \left(\text{out}_{\hat{S}(B)}(s_{\hat{S}(B)}) \cap \mathcal{T}^B \right),$$

i.e., the clock of the receiving machine is incremented by an arbitrary positive delay, all other clocks are unaffected, except for newly spawned clocks that are initialized to 0.

A LABELLED TRANSITION between DETCS-states $\langle S, W, M \rangle$ and $\langle \hat{S}, \hat{W}, \hat{M} \rangle$ is a quadruple

$$\left\langle \langle S, W, M \rangle, AB!_m, t, \langle \hat{S}, \hat{W}, \hat{M} \rangle \right\rangle,$$

such that $\langle \hat{S}, \hat{W}, \hat{M} \rangle$ is the $AB!_m$ - t -successor of $\langle S, W, M \rangle$. Finally, the LTS OF DETCS-STATES is the LTS $\langle X, \rightarrow \rangle$ whose set of states X is the set of DETCS-states and the labelled transition relation \rightarrow is the set of all labelled transitions between DETCS-states. We write

$$\langle S, W, M \rangle \xrightarrow[t]{AB!_m} \langle \hat{S}, \hat{W}, \hat{M} \rangle$$

when $\langle \hat{S}, \hat{W}, \hat{M} \rangle$ is the $AB!_m$ - t -successor of $\langle S, W, M \rangle$.

Remark 13 (Non-deterministic message delivery). Messages are delivered completely asynchronously, possibly including the re-ordering of messages (although this cannot happen in the examples). Additional assumptions about the message delivery can be made later on. Roughly, the scheduling is *demonic* non-determinism, not controlled by the system, but by the network operator.

model	state machine	dynamic	time	user I/O
CFSM	DFA	×	×	×
ECS	EMM	×	×	×
DECS	EMM	✓	×	×
DETCIS	EMM	✓	✓	×
DETCIS	ECIMM	✓	✓	✓

Figure 6. Comparison matrix of computational models for protocol instantiation.

5. Make it talk to the world: synchronous I/O interfaces

Conceptually, we are still in a *closed world*, because the development is starting from communicating finite state machines. This is why, so far, we could not meaningfully model the actual token ring protocol, because we are missing the possibility to read the data to send from the stations (and in a synchronous manner). We now sketch how we add the missing link to the environment, which makes it possible to model the original token ring protocol using the interactive version of communicating systems. To open up our systems and make them talk to the world, we introduce an interface for communication with users, external hardware, and other input/output agents in the environment (from the perspective of the machine). One way to make the system open is as follows: first, we assume that interaction with the local environment is *synchronous* (e.g., the user is prompted whenever an important message arrives); second, we take the simplest possible interface, considering only one prompt in each isolated turn. This approach is sufficient to offer the user a complete interaction automaton with finite state space, which could be given as a traditional Moore machine with synchronous input and output.

This approach is inspired by *read-eval-print loops* (REPLs): the machine prompts the local environment for input whenever it needs to make choices or read data. It then evaluates the data, transitions to the next state, and waits for the next message to arrive (after sending messages and spawning machines based on the new state).¹⁹ In the example of the Token Ring, the interaction with the station is used to fill up memory with a payload for the next turn of a partaking process in the Token Ring Protocol.

Let us describe in more detail how we can add interactivity to Moore-esque machines. For every received message $AB^?m$, there may arise the need to request input from the environment. Whenever such input is required, the machine produces a signal $\$z$ —in analogy to the print part in a REPL—that depends on the received message $AB^?m$ and the current state. The environment then produces a data item x in response to the prompt; e.g., this could be an answer to a multiple choice question or input that follows a regular expression pattern. Finally, the machine processes the message-data pair

¹⁹ This approach is compatible with a seamless user experience, as the user may be notified asynchronously about what has happened, using a message to itself.

$\langle \text{AB?m}, x \rangle$ to advance to the next state, then sends messages and spawns new machines.

Thus, in our definition, for each participant A , we additionally need a set of signals \mathcal{S}_A , a set of data \mathbb{X} , and a map of type $Q \times \mathcal{R}^A \rightarrow \mathcal{S}_A$ that outputs a signal to which the environment is expected to respond, synchronously. To formalize the synchronous communication, we suppose that each machine is connected with the environment via an input stream; then a choice of the environment is given by a map from signals $z \in \mathcal{S}_A$ to data $x \in \mathbb{X}$.²⁰ We augment the interaction labels to have the shape $\text{AB!m}.x$, representing a message reception $\langle \text{AB?m}, x \rangle$ and input data $x \in \mathbb{X}$, which was read from the environment in response to receiving the message AB!m . Accordingly, we could define the following generalization of Moore-esque machines.

Definition 14 (Moore-esque machine with interfaces). Relative to four sets $\Sigma, \Gamma, \Psi, \Xi$, a MOORE-ESQUE MACHINE WITH INTERFACES with input alphabet Σ , output alphabet Γ , signal alphabet Ψ , and data alphabet Ξ , is a quintuple $\langle Q, q_0, \delta, \rho, \mu \rangle$, consisting of

- a set of STATES Q ,
- an CURRENT STATE $q_0 \in Q$,
- a SIGNAL MAP $\rho: Q \times \Sigma \rightarrow \Psi$,
- a STATE TRANSITION MAP $\delta: Q \times \Sigma \times \Xi \rightarrow Q$, and
- an OUTPUT MAP $\mu: Q \rightarrow \Gamma$.

In summary, this means that whenever the machine is to react according to a received message, the machine will start an interaction “through” the I/O stream with whatever entities are on the “other side” of the I/O, be it a user, a specific hardware device, or anything that is within the same location and thus allows for synchronous communication. With this variation of Moore-esque machines at hand, we plan to capture the case of synchronous interaction with the environment, *i.e.*, the last line in Figure 6. This brings into the picture synchronous variations of the actor model.

6. Related work

Communication systems [BLT20] bear obvious family resemblances with actor models [Agh86b, Agh86a]. The proposed DETCSS are a minor variation of communication systems—at least if we take a step back to look at the big picture; in particular, they inherit the family resemblances. The main motivation for writing up a machine-based model is that we want to avoid committing to any specific one of the multitude of actor models, but rather have a suitable abstraction for all of them.

²⁰This input stream can be optional, and in principle two machines could share such input streams.

Let us consider some specific points to make the relation between actor models and communicating systems clearer.²¹ First, a labelled transition semantics has also been given to actor systems [AMST97]. If we summarize several transitions of the latter into *macro-steps* [DKVCDM16], we obtain a close correspondence between these macro-steps of actor systems and labelled transitions of DETCSS. Moreover, we can also make a connection to Clinger’s interaction diagrams [Cli81]: each path in the LTS of a DETCS that leads to a state with an empty message set corresponds to an interaction diagram with one event per labelled transition. In summary, the main point is that if we seriously consider the *Isolated Turn Principle* (ITP) [DKVCDM16]—shared by all actor models—we can think of the LTS semantics as the counterpart of the ITP for DETCSS as there is a bijective relation between labelled transitions on the one hand and turns performed by machines on the other hand.²²

Turning to more application-oriented work, there is a specific line of research that aims to bring session types [HYC08] to actor *languages*. Roughly, session types refine *interfaces* (in the sense of actor systems [DKVCDM16]), which describe which messages an actor is ready to process at any given point in time. Let us point out ElixirST [FT23] as one specific example of this line of research. ElixirST comes with a labelled transition semantics that is slightly more fine-grained, as it also considers outputs and, more importantly, function calls. However, the input labels of both LTSS correspond to each other directly. More generally, it seems hard to find a model of distributed message-passing systems that allows for Turing-complete computational processes and user interactions while being described *independently* of a specific programming language. Naturally, all early actor languages, such as ABCL/1 [YBS86], come with a built-in scripting language. However, even more recent general frameworks, such as timed Rebeca [ACI⁺11], introduce specific syntax to describe process behaviour. A production-ready approach is safe asynchronous event-driven programming in P [DGJ⁺13]. However, here again, the focus is not on a general model but rather on a specific way to enable asynchronous computation. A short overview of the wider context is given in a recent position paper [FHK⁺24].

7. Conclusion

We proposed DETCSS as a model of distributed computation that lies between the presentation of actor models and communicating finite state machines, including a timed labelled transition semantics. Instead of inventing yet another process description language, we presented a direct generalization

²¹ A fully fledged comparison to the actor model is beyond the scope of the paper.

²² Here, we abstract away from any substrate on which machines are running.

of communicating finite state (Moore) machines, using concepts from computability theory. Precisely, we elaborated semantics for two new main features: the dynamic creation of processes, [Section 4.2](#), and the use of clocks, [Section 4.3](#). These are necessary to reason about general protocols. To complete the picture, in [Section 5](#), we sketched how we can incorporate synchronous interactions with the environment, allowing machines “to talk to the outside world” [[Arm02](#)].

The main open task for future work is to study the relationship between choreographies, session types, and other mechanisms and calculi for protocol specification and implementation. The direct next step consists of filling in the details of user interactions, especially for the timed setting, in a way that matches existing programming languages in common use. A related topic is reasoning about the communication in a system as observed by a fixed set of “monitors” and how it relates to message logic [[GZ24](#)]. Further directions include structured systems with components or modules (see e.g., [[GY23](#)]), fault tolerance, and distributed runtime verification of protocols.

Acknowledgements. We would like to thank Murdoch James Gabbay for his clearly stated advice for how to improve the paper.

References

- ACI⁺11. Luca Aceto, Matteo Cimini, Anna Ingolfsdottir, Arni Hermann Reynisson, Steinar Hugi Sigurdarson, and Marjan Sirjani. Modelling and simulation of asynchronous real-time systems using timed rebeca. *Electronic Proceedings in Theoretical Computer Science*, 58:1–19, July 2011. URL: <http://dx.doi.org/10.4204/EPTCS.58.1>, doi:10.4204/eptcs.58.1. (cit. on p. 20.)
- Agh86a. Gul Agha. An overview of actor languages. *SIGPLAN Not.*, 21(10):58–67, jun 1986. doi:10.1145/323648.323743. (cit. on p. 19.)
- Agh86b. Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986. (cit. on pp. 2, 8, and 19.)
- AMST97. Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, January 1997. doi:10.1017/S095679689700261X. (cit. on p. 20.)
- APS14. Matteo Avalue, Alfredo Pironti, and Riccardo Sisto. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing*, 26:99–123, 2014.
- Arm97. Joe Armstrong. The development of erlang. *SIGPLAN Not.*, 32(8):196–203, aug 1997. doi:10.1145/258949.258967.
- Arm02. Joe Armstrong. Getting erlang to talk to the outside world. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Erlang, ERLANG ’02*, page 64–72, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/592849.592858. (cit. on p. 21.)
- BCG⁺21. Alessandro Bruni, Marco Carbone, Rosario Giustolisi, Sebastian Mödersheim, and Carsten Schürmann. *Security Protocols as Choreographies*, pages 98–111. Springer International Publishing, Cham, 2021. doi:10.1007/978-3-030-91631-2_5.
- BCK01. Paolo Baldan, Andrea Corradini, and Barbara König. A static analysis tech-

- nique for graph transformation systems. In *International Conference on Concurrency Theory*, pages 381–395. Springer, 2001. (cit. on p. 13.)
- BLT20. Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Choreography automata. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages*, pages 86–106, Cham, 2020. Springer International Publishing. (cit. on pp. 2, 8, 10, 11, 12, 19, and 26.)
- BP16. Carlos Baquero and Nuno Preguiça. Why logical clocks are easy: Sometimes all you need is the right language. *Queue*, 14(1):53–69, feb 2016. doi:10.1145/2898442.2917756.
- BP19. Tomasz Brengos and Marco Peressotti. Behavioural equivalences for timed systems. *Logical Methods in Computer Science*, Volume 15, Issue 1, February 2019. URL: <https://lmcs.episciences.org/5220>, doi:10.23638/LMCS-15(1:17)2019.
- BZ83. Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983. (cit. on pp. 4, 6, 7, 8, and 26.)
- CL85. K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, feb 1985. doi:10.1145/214451.214456.
- Cli81. William Douglas Clinger. *Foundations of Actor Semantics*. PhD thesis, Massachusetts Institute of Technology (MIT), 1981. URL: <https://dspace.mit.edu/handle/1721.1/6935>. (cit. on p. 20.)
- Co24. Anoma Contributors and others. Anoma Specification v0.1.4, 2024. URL: <https://github.com/anoma/nspec>. (cit. on pp. 2 and 3.)
- Com. Computable function. http://encyclopediaofmath.org/index.php?title=Computable_function&oldid=41830. retrieved 11 Nov. 2024. (cit. on p. 8.)
- DGJ⁺13. Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, page 321–332, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491956.2462184. (cit. on p. 20.)
- DKVCDM16. Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 years of actors: a taxonomy of actor models and their key properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016*, page 31–40, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/3001886.3001890. (cit. on pp. 3, 4, and 20.)
- FAS⁺23. Simon Fowler, Duncan Paul Attard, Franciszek Sowul, Simon J. Gay, and Phil Trinder. Special delivery: Programming with mailbox types. *Proceedings of the ACM on Programming Languages*, 7(ICFP):78–107, August 2023. URL: <http://dx.doi.org/10.1145/3607832>, doi:10.1145/3607832.
- FHK⁺24. Simon Fowler, Philipp Haller, Roland Kuhn, Sam Lindley, Alceste Scalas, and Vasco T. Vasconcelos. Behavioural types for heterogeneous systems (position paper). *Electronic Proceedings in Theoretical Computer Science*, 401:37–48, April 2024. URL: <http://dx.doi.org/10.4204/EPTCS.401.4>, doi:10.4204/eptcs.401.4. (cit. on p. 20.)
- FT23. Adrian Francalanza and Gerard Tabone. Elixirst: A session-based type system for elixir modules. *Journal of Logical and Algebraic Methods in Programming*, 135:100891, 2023. URL: <https://www.sciencedirect.com/science/article/pii/S2352220823000457>, doi:10.1016/j.jlamp.2023.100891. (cit. on p. 20.)
- Gir94. Jean-Yves Girard. Light linear logic. In *International Workshop on Logic and Computational Complexity*, pages 145–176. Springer, 1994. (cit. on p. 15.)
- GJ16. Tony Garnock-Jones. History of actors. <https://eighty-twenty.org/2016/10/18/actors-hopl>, Oct 2016. Accessed 24th of February 2025. (cit. on p. 1.)
- Göd31. Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica

- und verwandter systeme i. *Monatshefte für mathematik und physik*, 38:173–198, 1931. (cit. on p. 12.)
- Gur00. Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic (TOCL)*, 1(1):77–111, 2000.
- GY23. Lorenzo Gheri and Nobuko Yoshida. Hybrid multiparty session types – full version, 2023. URL: <https://arxiv.org/abs/2302.01979>, [arXiv:2302.01979](https://arxiv.org/abs/2302.01979). (cit. on p. 21.)
- GZ24. Murdoch J. Gabbay and Naqib Zarin. Message Logic. *Anoma Research Topics*, Dec 2024. URL: <https://doi.org/10.5281/zenodo.14251397>, [doi:10.5281/zenodo.14251398](https://doi.org/10.5281/zenodo.14251398). (cit. on p. 21.)
- HBS73. Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI’73*, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
- Her20. Drago Hercog. *Communication protocols: principles, methods and specifications*. Springer Nature, 2020. (cit. on p. 3.)
- Hew07. Carl Hewitt. What is commitment? physical, organizational, and social (revised). In Pablo Noriega, Javier Vázquez-Salceda, Guido Boella, Olivier Boissier, Virginia Dignum, Nicoletta Fornara, and Eric Matson, editors, *Coordination, Organizations, Institutions, and Norms in Agent Systems II*, pages 293–307, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- HMU01. John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001. (cit. on p. 7.)
- HYC08. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284, 2008. (cit. on p. 20.)
- IS14. Shams M. Imam and Vivek Sarkar. Selectors: Actors with multiple guarded mailboxes. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control, AGERE! ’14*, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery. [doi:10.1145/2687357.2687360](https://doi.org/10.1145/2687357.2687360).
- Jac95. Bart Jacobs. Objects and classes, co-algebraically. In *Object orientation with parallelism and persistence*, pages 83–103. Springer, 1995. (cit. on p. 4.)
- JR97. Bart Jacobs and Jan Rutten. A tutorial on (co) algebras and (co) induction. *Bulletin-European Association for Theoretical Computer Science*, 62:222–259, 1997.
- Kle52. Stephen Cole Kleene. *Introduction to Metamathematics*. P. Noordhoff N.V., Groningen, 1952. (cit. on p. 7.)
- KP04. Marco Kick and John Power. Modularity of behaviours for mathematical operational semantics. *Electronic Notes in Theoretical Computer Science*, 106:185–200, 2004.
- Lam78. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978. [doi:10.1145/359545.359563](https://doi.org/10.1145/359545.359563).
- LY19. Julien Lange and Nobuko Yoshida. Verifying asynchronous interactions via communicating session automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 97–117, Cham, 2019. Springer International Publishing.
- Mal95. Grant Malcolm. Behavioural equivalence, bisimulation, and minimal realisation. In *Workshop on the Specification of Abstract Data Types*, pages 359–378. Springer, 1995.
- Mil89. R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., USA, 1989.
- Mon23. Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023.

- Moo56. Edward F. Moore. *Gedanken-Experiments on Sequential Machines*, pages 129–154. Princeton University Press, Princeton, 1956. URL: <https://doi.org/10.1515/9781400882618-006> [cited 2024-11-19], doi:doi:10.1515/9781400882618-006. (cit. on pp. 7, 8, and 9.)
- MV93. S. Mauw and G. J. Veltink, editors. *Algebraic specification of communication protocols*. Cambridge University Press, USA, 1993. (cit. on p. 5.)
- RJ67. Hartley Rogers Jr. *Theory of recursive functions and effective computability*. McGraw-Hill Book Company, 1967. (cit. on pp. 7 and 12.)
- Sch90. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. doi:10.1145/98163.98167.
- SHC⁺23. Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. Semantics for Noninterference with Interaction Trees. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:29, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.29>, doi:10.4230/LIPIcs.ECOOP.2023.29.
- She15. Justin Sheehy. There is no now: Problems with simultaneity in distributed systems. *Queue*, 13(3):20–27, mar 2015. doi:10.1145/2742694.2745385.
- SS76. Dana Scott and Christopher Strachey. *Toward a Mathematical Semantics for Computer Languages*. Prentice-Hall, 1976.
- Stu24. Felix Stutz. *Implementability of Asynchronous Communication Protocols - The Power of Choice*. PhD thesis, Kaiserslautern University of Technology, Germany, 2024. URL: <https://kluedo.ub.rptu.de/frontdoor/index/index/docId/8077>.
- Tal98. Carolyn L. Talcott. Composable semantic models for actor theories. *Higher Order Symbolic Computation*, 11(3):281–343, 1998. URL: <http://dx.doi.org/10.1023/A:1010042915896>, doi:10.1023/a:1010042915896.
- TF21. Gerard Tabone and Adrian Francalanza. Session types in elixir. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2021, page 12–23, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3486601.3486708.
- Tur37a. A. M. Turing. Computability and λ -definability. *Journal of Symbolic Logic*, 2(4):153–163, 1937. doi:10.2307/2268280. (cit. on p. 7.)
- Tur37b. A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. URL: <https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230>, arXiv:<https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s2-42.1.230>, doi:10.1112/plms/s2-42.1.230. (cit. on p. 7.)
- XZH⁺19. Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, December 2019. URL: <http://dx.doi.org/10.1145/3371119>, doi:10.1145/3371119.
- YBS86. Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in abcl/1. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA ’86, page 258–268, New York, NY, USA, 1986. Association for Computing Machinery. doi:10.1145/28697.28722. (cit. on p. 20.)
- YW15. Shohei Yasutake and Takuo Watanabe. Actario: A framework for reasoning about actor systems. In *Workshop on Programming based on Actors, Agents,*

and Decentralized Control (AGERE), 2015.

A. Additional background

For the reader's convenience, we quickly review the main ideas of a well-cited paper that introduces itself as an investigation of *a model of communications protocols based on finite-state machines* [BZ83]. It defines *protocol* as a finite set of communicating automata (and that is why we would rather call this concept a *protocol instance*); however, this is the original definition.

Definition 15 (Protocol [BZ83, Definition 2.1]). A PROTOCOL is a quadruple

$$\langle \langle S_i \rangle_{i=1}^N, \langle o_i \rangle_{i=1}^N, \langle M_{ij} \rangle_{i,j=1}^N, \text{succ} \rangle$$

where

- N is a positive integer (representing the number of processes),
- $\langle S_i \rangle_{i=1}^N$ are N disjoint finite sets (S_i represents the set of states of process i),
- each o_i is an element of S_i (representing the initial state of process i),
- $\langle M_{ij} \rangle_{i,j=1}^N$ are N^2 disjoint finite sets with M_{ii} empty for all i (M_{ij} represents the messages that can be sent from process i to process j).
- succ is a partial function mapping for each i and j ,

$$S_i \times M_{ij} \rightarrow S_i \quad \text{and} \quad S_i \times M_{ji} \rightarrow S_i.$$

($\text{succ}(s, x)$ is the state entered after a process transmits or receives message x in state s . It is a transmission if x is from M_{ij} , a reception if x is from M_{ji} .)

There are a couple of fairly trivial observations about this definition that do not even need to consider the dynamics of a protocol: each process has an identifier $i \in \{1, \dots, N\}$, and each message has (implicitly) a sender and a receiver because for each message x , there is a unique pair of indices $\langle i, j \rangle$ such that x in M_{ij} . These indices will be called participants in the current paper (following [BLT20]) where we set out to generalize the above classic definition.

B. Timers

Finally, we give EMMs the ability use a crude “timer” to avoid busy waiting; however timer notifications are allowed to come “late”, in a sense to be made precise in the definition of the LTS.²³

Timers only help in avoiding busy waiting, in case we have suitable clock processes in our implementations. Timers will be rather limited in that they only make sure that machines are waiting long enough, *e.g.*, to declare a time-out. A machine can set a timer to a future point in time and a local “clock process” will send a notification at that point in time or *thereafter*. In other words, timer notifications are not assumed to be on time.²⁴

In the formal definitions, we have yet a different variation of the encoding $\{\llbracket n \rrbracket_A''\}_{A \in \mathcal{P}}$, because the involved input and output alphabets are different.

Definition 16 (Dynamic effective timed communicating system with timers). A DYNAMIC EFFECTIVE TIMED COMMUNICATING SYSTEM WITH TIMERS (DETCST) is a triple $\langle S, W, T \rangle$ consisting of

- a partial map $S: \mathcal{P} \dashrightarrow \text{EMM}$ from PIDS to EMMs such that

$$S(A) \in \text{EMM} \left(\mathbb{N} \times (\mathcal{R}^A \cup 1), \text{Powerset}_{fn}(\mathcal{T}^A \cup \mathcal{N}^A) \times \mathbb{N}_\perp \right)$$

holds for each PID $A \in \text{df}(S)$ where $\mathbb{N}_\perp = \mathbb{N} \cup \{\text{undefined}\}$,

- a partial map from PIDS of to the natural numbers, which assigns to each activated PID its WALL-CLOCK TIME,
- a partial map from PIDS to the natural numbers that is only defined for a subset of the activated PIDS,

where a PID is ACTIVATED if S is defined for it.

We now extend the labelled transition semantics in the expected way.

Definition 17 (Labelled transition system of a DETCS). A DETCS-STATE is defined as a quadruple $\langle S, W, T, M \rangle$, such that the first three components together form a DETCS and the last component is a finite set of transmissions $M \in \text{Powerset}_{fn}(\bigcup_{A \in \mathcal{P}} \mathcal{T}^A)$. Given a DETCS-state $\langle S, W, T, M \rangle$, a transmission $AB!m \in M$, and a strictly positive *delay* $0 < t \in \mathbb{N}$, we define *the*

²³ In case the reader is wondering why we have not proceeded in analogy to what would be communicating finite *timed automata*: one reason is that we only want a very weak notion of timer, that only is useful to ensure that a machine has wait long enough; in particular, we do not want to add the global assumption that we can implement strict timers. Timed automata on the other hand have stronger requirements for clocks, which seem almost unimplementable.

²⁴ The “weakness” of this notion of timer is intended to facilitate faithful implementations on common consumer hardware using one of the popular operating systems.

$AB!m$ - t -SUCCESSOR of $\langle S, W, T, M \rangle$ as the DETCS-state $\langle \hat{S}, \hat{W}, \hat{T}, \hat{M} \rangle$ given by

$$\begin{aligned}\hat{S}(C) &= \begin{cases} S(B) @ \text{in}_{S(B)} \left(\langle s_{S(B)}, \langle t', AB!m \rangle \rangle \right) & \text{if } C = B \\ \text{where } t' = W(B) + t \\ S(C) & \text{if } C \in \text{df}(S) \setminus \{B\} \\ \llbracket n \rrbracket'_C & \text{if } Bn^*C \in (\text{out}_{\hat{S}(B)}(s_{\hat{S}(B)}))_1 \\ & \text{and } C \notin \text{df}(S) \\ \text{undefined} & \text{otherwise} \end{cases} \\ \hat{W}(C) &= \begin{cases} W(B) + t & \text{if } B = C \\ W(C) & \text{if } C \in \text{df}(S) \setminus \{B\} \\ 0 & \text{if } Bn^*C \in (\text{out}_{\hat{S}(B)}(s_{\hat{S}(B)}))_1 \\ & \text{and } C \notin \text{df}(S) \\ \text{undefined} & \text{otherwise} \end{cases} \\ \hat{T}(C) &= \begin{cases} (\text{out}_{\hat{S}(B)}(s_{\hat{S}(B)}))_2 & \text{if } B = C \\ T(C) & \text{if } C \in \text{df}(S) \setminus \{B\} \\ \text{undefined} & \text{otherwise} \end{cases} \\ \hat{M} &= (M \setminus \{AB!m\}) \cup \left(\text{out}_{\hat{S}(B)}(s_{\hat{S}(B)}) \cap \mathcal{T}^B \right)\end{aligned}$$

Given a DETCS-state $\langle S, W, T, M \rangle$, a non-negative *delay* $t \in \mathbb{N}$, and a PID B such that $T(B)$ is defined, *the* B - t -SUCCESSOR of the DETCS-state $\langle S, W, T, M \rangle$ is the DETCS-state $\langle \hat{S}, \hat{W}, \hat{T}, \hat{M} \rangle$ where

$$\begin{aligned}\hat{S}(C) &= \begin{cases} S(B) @ \text{in}_{S(B)} \left(\langle s_{S(B)}, \langle t', \langle \rangle \rangle \rangle \right) & \text{if } C = B \\ \text{where } t' = W(B) + t + T(B) \\ S(C) & \text{if } C \in \text{df}(S) \setminus \{B\} \\ \llbracket n \rrbracket'_C & \text{if } Bn^*C \in (\text{out}_{\hat{S}(B)}(s_{\hat{S}(B)}))_1 \\ & \text{and } C \notin \text{df}(S) \\ \text{undefined} & \text{otherwise} \end{cases} \\ \hat{W}(C) &= \begin{cases} W(B) + t + T(B) & \text{if } B = C \\ W(C) & \text{if } C \in \text{df}(S) \setminus \{B\} \\ 0 & \text{if } Bn^*C \in (\text{out}_{\hat{S}(B)}(s_{\hat{S}(B)}))_1 \\ & \text{and } C \notin \text{df}(S) \\ \text{undefined} & \text{otherwise} \end{cases} \\ \hat{T}(C) &= \begin{cases} (\text{out}_{\hat{S}(B)}(s_{\hat{S}(B)}))_2 & \text{if } B = C \\ T(C) & \text{if } C \in \text{df}(S) \setminus \{B\} \\ \text{undefined} & \text{otherwise} \end{cases}\end{aligned}$$

$$\hat{M} = (M \setminus \{\textcolor{blue}{AB}!\textcolor{violet}{m}\}) \cup \left(\text{out}_{\hat{S}(\textcolor{blue}{B})}(s_{\hat{S}(\textcolor{blue}{B})}) \cap \mathcal{T}^{\textcolor{blue}{B}} \right)$$

A LABELLED TRANSITION between DETCS-states $\langle S, W, T, M \rangle$ and $\langle \hat{S}, \hat{W}, \hat{T}, \hat{M} \rangle$ is a quadruple

$$\left\langle \langle S, W, T, M \rangle, \Xi, t, \langle \hat{S}, \hat{W}, \hat{T}, \hat{M} \rangle \right\rangle$$

such that $\langle \hat{S}, \hat{W}, \hat{T}, \hat{M} \rangle$ is the Ξ - t -successor of $\langle S, M \rangle$ where $\Xi \in \{\textcolor{blue}{B}, \textcolor{blue}{AB}!\textcolor{violet}{m}\}$. The LTS of ECTMMs is the LTS $\langle X, \rightarrow \rangle$ whose set of states X is the set of DETCS-states and the labelled transition relation \rightarrow is the set of all labelled transitions between DETCS-states.