Using Temporal Session Types to Analyze Time Complexities of Concurrent Programs

Joseph Moonan Walbran
walbr037@morris.umn.edu
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA

Abstract

Das et al. develop a method for analyzing the time complexity of concurrent, message-passing algorithms. Their method is based on adding timing information to datatypes. Specifically, they use a family of datatypes called *session types*; these constrain the structure of interactions that may take place over a channel of communication. In Das's system, the timing properties of an algorithm can be verified by a typechecker: if the timing information in the session types is mismatched, the computer will report a type error. In their paper, Das et al. develop the theory for such a typechecker, but do not provide an implementation.

Keywords: Formal methods, concurrent algorithms, pi calculus, process calculi, formal systems, type systems.

1 Introduction

A common question in computer science is "how long does this algorithm take to run?" Answering that question can mean implementing the algorithm, running it, and timing how long it takes. But trying to directly measure the runtime of an algorithm is tricky—the answer will depend on the hardware the program is running on, and on how many background processes were competing with the program for resources. Instead of measuring algorithms empirically, it's often more meaningful to look at an algorithm written down and count how many steps are involved. The number of steps is called the *time-complexity* of the algorithm.

Das et al. are interested in analyzing the time complexity of concurrent algorithms. An algorithm is said to be *concurrent* if it consists of two or more parts that execute independently of each other [8]. For example, a concurrent algorithm might have several threads, each of which does some part of the computation. Analyzing the time complexity of concurrent algorithms can be difficult; one needs to consider how the pieces of a concurrent program interact with each other. Sometimes, two pieces can run simultaneously. Other times, one piece of the program needs to wait, idling until another piece is ready.

Das's paper focuses on *message-passing* concurrent systems. These are concurrent systems where the independent parts only interact by sending data between each other over channels [2]. This model can describe multithreaded programs where the threads communicate over shared queues, or Unix processes that run in parallel, sending data over pipes. However, the message-passing model is less suitable for describing multithreaded programs that make use of shared, global memory.

Das et al. provide a formal method for analyzing time complexities of concurrent, message-passing algorithms. For example, suppose an algorithm reads data from an input channel, processes that data, and then writes the result to an output channel. Das's method can be used to answer questions like "When the algorithm reads a message, how long is the delay before it writes a result to the output channel?" (This is called the *latency* of the algorithm.) It can also answer questions like "how many messages can the algorithm process per second?" (This is the *maximum message rate* of the algorithm.) [2] Knowing these quantities is important when trying to send data over a network, for example. If the latency is too high, the network will be slow and unresponsive. If someone tries to send data faster than the maximum message rate, the network won't be able to handle it, and it will start dropping packets. [7]

To answer these questions, Das et al. develop a notation for adding timing information to datatypes. Using the timing information of each piece of the program, one can often reconstruct the time cost of the whole algorithm.

There are two modes in which one can use Das's method. First, one can *find* the time complexity of a concurrent algorithm by annotating each step of the algorithm with timing information, and then aggregating the different pieces of timing information to determine the total time complexity. This process is difficult to automate, and generally needs to be done by hand. [2]

Second, once a time complexity has been found in this manner, one can *verify* that the solution is correct. It's easy for programmers to make mistakes when analyzing concurrent programs, so the fact that Das et al. provide a way to check for errors is valuable. Furthermore, this verification can be automated; there's a completely mechanical procedure for checking whether an algorithm is annotated with correct time complexity. The key idea behind this procedure is that if the timing information is given incorrectly, the algorithm will contain a type error somewhere; the timing

information in the datatypes will be mismatched. In this way, Das's method takes the problem of verifying that an algorithm has a particular time complexity, and turns it into a typechecking problem. [2]

Das's system has three layers to it:

- In the first layer, Das et al. lay out formalization for how concurrent computation works. The formalization they use is called *π-calculus*, and it dates back to the 1990s. [6]
- In the second layer, Das et al. describe a way to typecheck concurrent algorithms. There are a few different approaches to typechecking concurrent programs; Das et al. use a specific system called *session types*, which adds typechecking on top of π-calculus by constraining what messages can be sent over channels. [1][4][5]
- In the third layer, Das et al. introduce a way to add timing information to datatypes. This is their novel contribution; they call the resulting system *temporal session types*. [2]

We will discuss each of these layers in turn.

2 π-calculus

Computer scientists often write algorithms in pseudocode, but for their system, Das et al. write algorithms in a particular notation with exact specifications. This helps keep their analysis clear and rigorous, which is important when developing a general method for analyzing concurrent algorithms. There are several competing systems for notating concurrent algorithms, but Das et al. use a system called π-calculus, which was developed in the early 1990s by Milner et al. [6]

π-calculus represents concurrent computation using the notions of *processes* and *channels*. Processes are sequences of instructions that execute in parallel with other processes. Channels are communication links that processes use to send data to each other. In π-calculus, there are a few elementary operations that processes can perform, such as spawning new processes and sending a piece of data over a channel. The set of elementary operations is small, which helps keep programs expressed in π-calculus easy to reason about; there are only a few cases to consider. More complex operations can be expressed using simple ones as building blocks. [6]

Let's look at the elementary operations of π-calculus.[1]

[1]The details of π-calculus vary significantly from author to author. Where there are differences, we will prefer the formulation given in Das et al, with one exception regarding the treatment of channels.

Das et al. distinguish between a process's primary channel, which it *provides*, and secondary channels, which it *uses*, but which belong to other processes. Milner et al., Honda, and Caires do not make this distinction [1][2][5][6]. There are reasons why this distinction is useful, but they are not relevant to our topic, so we will ignore the distinction and just refer to "channels". [2]

2.1 Defining a process

To define a process in π-calculus, one gives it a name; a list of arguments, where each argument is a channel that the process will use; and a sequence of operations that the process will perform [6]. Process definitions are written like this:

```
processName(channel1, channel2, ..., channelN) =
  operation1;
  operation2;
  ...;
  operationN
```

Here, `channel1` through `channelN` are the arguments of a process called `processName`.

2.2 Spawning a process

Processes can spawn other processes; the new processes run in parallel with the original. Spawning a process is written `processName(channel1, channel2, ...)`, like a function call. [6]

For example, the following is a process that spawns two copies of itself before terminating.

```
a() = a(); a()
```

Running `a()` will cause the number of active processes to grow exponentially.

2.3 Closing a channel

If two processes are connected by a channel, one of them can *close* the channel; this destroys the channel, so that neither process can use it again [1]. Closing a channel `c` is written "`close c`".

Additionally, a process can wait for a channel to be closed by the process on the other end [1][2]. Waiting for channel `c` to close is written "`wait c`".

2.4 Sending a label

The main type of data that processes send to each other are *labels*. A label is one of a predefined, finite set of symbols [2]. For example, a program that sends messages in Morse code might have the following set of labels:

```
{ DOT, DASH, NEXT_LETTER, $ }
```

where the label `$` indicates the end of a message.

Sending a label `L` over channel `c` is written `c.L`, with a period. For example, the following example defines a process that sends the message "HI" ("• • • • • •" in Morse code).

```
sayHi(outChannel) =
  outChannel.DOT;
  outChannel.DOT;
  outChannel.DOT;
  outChannel.DOT;
  outChannel.NEXT_LETTER;
  outChannel.DOT;
  outChannel.DOT;
```

```
  outChannel.$;
  close outChannel
```

Here, `sayHi` takes one argument, the channel it should send messages over. Then, `sayHi` sends eight labels over that channel, one at a time; these labels encode the text "HI" in Morse code. Finally, `sayHi` closes the channel, since it's finished using it.

2.5 Receiving and branching on a label

When a process receives a label over a channel, it needs to be able to take different behaviors depending on the value of that label. In the Morse code example, a process needs to be able to inspect a label to determine whether it's a `DOT` or a `DASH`.

In π-calculus, this is accomplished using the "`case`" operation, which reads the next label from a channel and branches on its value [2]. The `case` operation is written as follows:

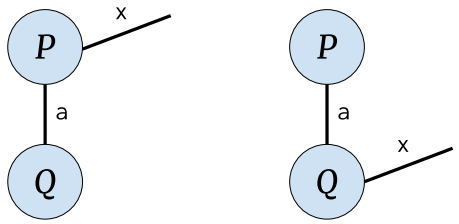
```
case c
| label1 => operation1; ...
| label2 => operation2; ...
| ...
| labelN => operationN; ...
```

The above process waits until it receives a label from `c`, and then looks at what label it received. If it received `label1`, it continues with `operation1`; if it received `label2`, it continues with `operation2`; and so on. We leave undefined what happens if the value received doesn't match any of `label1` through `labelN`.

As an example of the use of the `case` operation, the following process takes a message in Morse code and changes all the dots to dashes, and vice versa.

```
invert(inChannel, outChannel) =
  case inChannel
  | DOT =>
      outChannel.DASH;
      invert(inChannel, outChannel)
  | DASH =>
      outChannel.DOT;
      invert(inChannel, outChannel)
  | NEXT_LETTER =>
      outChannel.NEXT_LETTER;
      invert(inChannel, outChannel)
  | $ =>
      outChannel.$;
      wait inChannel;
      close outChannel
```

The `invert` process uses recursion to loop over the entire input. The first instance of `invert` reads one label from `inChannel`, and then spawns a new instance of `invert` that's responsible for reading the next label. In this way, `invert` continues to call itself recursively until it receives the `$` label, which is the base case of the recursion. Upon receiving `$`, `invert` makes sure all the channels close properly, and then shuts down.



(a) Before sending `x` **(b)** After sending `x`

Figure 1. Sending channel `x` from process P to process Q

2.6 Sending and receiving channels

The characteristic feature of π-calculus is the ability to send one channel over another channel. This lets processes pass channels between each other, so that the network of channels changes over time. [6]

Suppose processes P and Q are connected by a shared channel `a`, and P has a channel `x` with incoming data. P can give Q access to the data by sending the whole channel `x` over `a`. This operation is illustrated in figure 1.

Sending a channel `x` over a channel `a` is written "`send a x`". The complementary operation, receiving a new channel from `a` and giving it the name `x`, is written "`x <- recv a`".

Once a process sends a channel `x`, it loses access to `x`. Only the process that receives `x` can use it from then on.

The following example demonstrates the use of `recv`. In this example, `server` is a process that loops forever; in the body of the loop, `server` receives two channels, one for input and one for output. The server spawns an `invert` process, as described in section 2.5, to read a Morse code message from the first channel, process it, and write the result to the second channel.

```
server(listeningConnection) =
  inChannel <- recv listeningConnection;
  outChannel <- recv listeningConnection;
  invert(inChannel, outChannel);
  server(listeningConnection)
```

A nice benefit of using the `invert` process this way is that each instance of `invert` runs in parallel with `server`, so `server` does not have to wait for `invert` to finish before it can accept more channels.

2.7 Creating channels

Most formulations of π-calculus provide some method for creating new channels. How this operation works, and what its limitations are, vary from author to author [1][2][6]. Due to lack of space, we won't discuss creating new channels in this paper.

2.8 Next steps

π-calculus gives a precise way to describe message-passing concurrent algorithms using a small set of rules. But, without further tools, it's difficult to figure out the time complexity of a π-calculus process.

When analyzing the time complexity of a synchronous procedure, one can count just the number of operations it performs. This approach is not sufficient to find the time complexity of concurrent processes, like those of π-calculus. In concurrent programs, a process will sometimes accrue a time cost just by sitting idle, while it waits for some other process to be ready; when analyzing time complexity, one needs a way to account for these delays.

For example, when a process uses an operation like `case` or `recv` to read from a channel, it blocks, sitting idly until the process on the other end of the channel is ready to send it data. This type of communication between processes creates challenges for determining the exact timing of messages. [2]

To figure out where these delays occur, Das et al. need a way to describe the *structure of interactions between processes*. This is what session types provide.

3 Session types

Session types are a family of datatypes that describe channels. Each session type describes the structure of the interactions that can take place over a particular channel.

Session types can be more detailed than types like "string" or "array of integers", but they fulfill the same role. The point of typechecking a program is to make sure that values all have the form that the programmer expects them to have. If the programmer wants a variable to have the form "array of integers", but assigns it a value with the form "string", then there's a mismatch; a typechecker reports these mismatches as type errors.

In the same way that arrays have a particular type, it's possible to speak of channels as having a particular type. Channels are defined by the kinds of messages that one is allowed to send over them. If users of channel `x` are expected to send three labels and then close `x`, but users of channel `y` are expected to send an infinite stream of labels, then channels `x` and `y` have different types.

Session types formalize this idea by describing what a sequence of interactions over a channel should look like. For example, in the `server` process in section 2.6, the session type for the channel `listeningConnection` would contain the information "receive a channel for reading Morse code messages; then receive a channel for writing Morse code messages; then repeat". In this way, a session type is like a very small network protocol; it's a contract that processes talking over the channel need to abide by. [5]

Let's look at some of the different forms a session type can take.

3.1 Closing a channel; waiting for a channel to close

The simplest session types are **1**, which describes a channel that should be closed immediately, and $\bot$, which describes a channel that is about to be closed from the other end.[2] [1] [2]

If channel `x` has type **1**, the only allowed operation on `x` is "`close x`". Similarly, if `x` has type $\bot$, the only allowed operation on `x` is "`wait x`".

We say that two session types are *duals* of each other if they describe the same interaction as seen from opposite sides of the channel. For example, **1** and $\bot$ are duals, since they each describe a different side of the same interaction.

3.2 Internal choice

An *internal choice* type, written with the $\oplus$ symbol, asserts that one label from a particular set will be sent over a channel. Let `A` and `B` be labels. Then, if a channel `x` has the session type $\oplus\{\texttt{A} : T_1,\ \texttt{B} : T_2\}$, that means the only allowed operations on `x` are "`x.A`" and "`x.B`". If the process sends label `A`, the rest of its interactions over `x` must adhere to the session type T_1; if it sends `B`, the rest of its actions must adhere to T_2. [2]

For example, consider the process `sayHi(outChannel)` from section 2.4. Here, the channel `outChannel` can be described by the recursively-defined session type

$$
\begin{aligned}
&\textit{sendMessage} = \oplus\{\\
&\quad \texttt{DOT} : \textit{sendMessage},\\
&\quad \texttt{DASH} : \textit{sendMessage},\\
&\quad \texttt{NEXT\_LETTER} : \textit{sendMessage},\\
&\quad \$: \mathbf{1}\\
&\}
\end{aligned}
$$

The type *sendMessage* asserts that `sayHi` can choose whether to send `DOT`, `DASH`, `NEXT_LETTER`, or `$` over `outChannel`. If `sayHi` sends the `$` label, it needs to close `outChannel` afterwards. But, if `sayHi` sends `DOT`, `DASH`, or `NEXT_LETTER`, it should take another action of type *sendMessage*—that is, it should send more labels.

3.3 External choice

The dual of internal choice types are *external choice* types, written with the & symbol, which assert that a process is prepared to receive any label from a certain set. If a channel `x` has the session type $\&\{\texttt{A} : T_1,\ \texttt{B} : T_2\}$, then the only allowed operation on `x` is "`case x | A => ... | B => ...`". Here, if the process reading from `x` receives `A`, it should proceed as described by the session type T_1, and if it receives `B`, it should proceed as described by T_2. [2]

For example, in section 2.5, the process

```
invert(inChannel, outChannel)
```

[2]The notation used for session types comes from a branch of mathematics called linear logic, which studies constructions involving consumable resources. [3][1]

uses two channels. Of these, `outChannel` can be described using the session type *sendMessage* defined above, but `inChannel` has the external choice type

readMessage = &{
  `DOT` : *readMessage*,
  `DASH` : *readMessage*,
  `NEXT_LETTER` : *readMessage*,
  `$` : ⊥
}

which says, roughly, that if `invert` sees one of `DOT`, `DASH`, or `NEXT_LETTER`, it should keep reading; but, if it sees a `$` label, then it has reached the end of the message, and it should wait for `inChannel` to close.

3.4 Sending a channel; receiving a channel

The final two session types describe sending and receiving a channel.

Sending a channel is written with a ⊗ symbol. If a channel `x` has session type $T \otimes T_{rest}$, then the only allowed operation on `x` is "`send x f`". Here, `f` must be a channel of type T, and after sending `f` over `x`, the rest of the process's messages over `x` must follow the session type T_{rest}. [2]

Receiving a channel is the dual of sending a channel, and is written with the ⊸ symbol. If a channel `x` has session type $T \multimap T_{rest}$, then the only allowed operation on `x` is "`f <- recv x`". Here, `f` is a newly-assigned channel of type T, and the rest of the process's messages over `x` must follow the session type T_{rest}. [2]

As an example, consider the process

```
server(listeningConnection)
```

from section 2.6. Here, the channel `listeningConnection` can be described by the recursively-defined session type

$$listen = readMessage \multimap (sendMessage \multimap listen)\,.$$

This type says that the server receives a channel for reading Morse code messages; then it receives a channel for for writing Morse code messages; and then the server repeats this process.

3.5 Next steps

Recall Das's motivation for using session types: once one knows how interactions over a channel are structured, one can figure out the timing of these messages, and from there, one can figure out the time complexity of the program.

The final layer of Das's method, *temporal session types*, allows one to find the timing of messages.

4 Temporal session types

The core idea of temporal session types is to introduce one additional session type, $\circ T$, which delays the session type T by one unit of time. As a shorthand, one writes $\circ^n T$ to indicate T delayed by n units of time. [2]

Which operations incur a delay is a choice that will vary from application to application—Das's system is flexible enough to work with different cost models. A reasonable choice is to decide that every operation that does I/O incurs a time cost of one unit, and all other operations take negligible time. Under this cost model, the operations

- `close channel`
- `wait channel`
- `channel.LABEL`
- `case channel`
- `send channel x`
- `x <- recv channel`

all produce a delay. Spawning a process doesn't do any I/O, so it happens instantaneously.

Using this cost model, one can annotate the source code to show where the delays are. By convention, delays occur after the operation that produces them; you can think of the delay as a cooldown period. Let's look at `sayHi(outChannel)`:

```
sayHi(outChannel) =
  outChannel.DOT;           (delay 1)
  outChannel.DOT;           (delay 1)
  outChannel.DOT;           (delay 1)
  outChannel.DOT;           (delay 1)
  outChannel.NEXT_LETTER;   (delay 1)
  outChannel.DOT;           (delay 1)
  outChannel.DOT;           (delay 1)
  outChannel.$;             (delay 1)
  close outChannel          (delay 1)
```

After `sayHi` sends a label, it must wait one time unit before proceeding. So, the temporal session type of `outChannel` is

timedSendMessage = ⊕{
  `DOT` : ∘*timedSendMessage*,
  `DASH` : ∘*timedSendMessage*,
  `NEXT_LETTER` : ∘*timedSendMessage*,
  `$` : ∘**1**
}

which is resembles the *sendMessage* type from section 3.2, but with a delay after each label is sent.

The process `invert(inChannel, outChannel)` is more interesting to analyze, since the temporal session type of `outChannel` will depend on the message rate of `inChannel`: if `invert` receives labels at a very slow rate, it will send labels `outChannel` at a very slow rate.

Suppose `inChannel` has a delay of n time units between labels. Then, the temporal session type of `inChannel` is

$$
\begin{aligned}
&\textit{readMessageSlowly} = \&\{\\
&\quad \texttt{DOT} : \circ^n \textit{readMessageSlowly},\\
&\quad \texttt{DASH} : \circ^n \textit{readMessageSlowly},\\
&\quad \texttt{NEXT\_LETTER} : \circ^n \textit{readMessageSlowly},\\
&\quad \$: \circ^n \bot\\
&\}
\end{aligned}
$$

If the message rate of `inChannel` is known, one can annotate the source code of `invert` to show where the delays are:

```
invert(inChannel, outChannel) =
  case inChannel
  | DOT =>                          (delay 1)
      outChannel.DASH;              (delay 1)
                                    (delay k)
      invert(inChannel, outChannel)
  | DASH =>                         (delay 1)
      outChannel.DOT;               (delay 1)
                                    (delay k)
      invert(inChannel, outChannel)
  | NEXT_LETTER =>                  (delay 1)
      outChannel.NEXT_LETTER;       (delay 1)
                                    (delay k)
      invert(inChannel, outChannel)
  | $ =>                            (delay 1)
      outChannel.$;                 (delay 1)
                                    (delay k)
      wait inChannel;               (delay 1)
      close outChannel              (delay 1)
```

Here, the delay of 1 after each `=>` and `.` are necessary because sending and receiving a label each take one unit of time. By contrast, the delays of k represent time that `invert` spends idling while it waits for the next label over `inChannel`.

The total delay between consecutive reads from *inChannel* needs to equal n, the delay between messages. So one can solve the equation $1 + 1 + k = n$ to find that the time spent idling is $k = n - 2$. In particular, since k needs to be a nonnegative time, it must be true that $n \geq 2$—that is, the maximum message rate of `inChannel` is 1 label every 2 time units. If the message rate were any faster, `invert` would not have time to process a label before the next one arrives.

Once the time spent idling is known, it's possible to find the temporal session type of `outChannel`: one can look through the annotated source code, find what messages get sent over `outChannel`, and find what the delay is between those messages. We find that the temporal session type of `outChannel` is $\circ$*sendMessageSlowly*, where *sendMessageSlowly* is defined as follows:

$$
\begin{aligned}
&\textit{sendMessageSlowly} = \oplus\{\\
&\quad \texttt{DOT} : \circ^n \textit{sendMessageSlowly},\\
&\quad \texttt{DASH} : \circ^n \textit{sendMessageSlowly},\\
&\quad \texttt{NEXT\_LETTER} : \circ^n \textit{sendMessageSlowly},\\
&\quad \$: \circ^n \mathbf{1}\\
&\}
\end{aligned}
$$

Note that $\circ$*sendMessageSlowly* has a $\circ$ at the beginning, unlike *readMessageSlowly*. This is because `invert` reads from `inChannel` right away, but it doesn't write to `outChannel` until one unit of time has already passed. The $\circ$ at the beginning of $\circ$*sendMessageSlowly* reflects this initial delay.

Because of the initial delay, $\circ$*sendMessageSlowly* is offset from *readMessageSlowly* by one time unit. As a consequence, the latency of `invert` is one time unit; that's how much time elapses between when `invert` reads a label and when it passes that label along.

Finally, from the definition of *sendMessageSlowly*, it can be seen that `outChannel` has the same message rate as `inChannel`; they both have a delay of n time units between messages.

To summarize, this analysis shows:

- The maximum message rate of `inChannel` is 1 message every 2 time units.
- The message rate of `outChannel` is the same as that of `inChannel`.
- The latency between `inChannel` and `outChannel` is 1 time unit.

5 Conclusion

In time-sensitive applications, it's valuable to know how quickly an algorithm can process messages over a channel. To address this problem, Das et al. develop temporal session types as a way to analyze the timing properties of a message-passing concurrent algorithm, including the latency of the algorithm and the maximum message rates of channels.

Since these timing properties are built into the type system of π-calculus, a typechecker can mechanically verify that the timings are correct. Once each channel is assigned a temporal session type, a typechecker will be able to look at the source code and determine whether the channels actually have the types indicated.

Das et al. have not yet implemented such a typechecker, for π-calculus or for any programming language; their 2018 paper just develops the theory behind temporal session types. In future work, they would like to see if a programming language with temporal session types can be implemented and made practical. Additionally, although Das describe how to typecheck temporal session types, they don't yet have a way to infer the temporal session type of a channel automatically. In future research, they'd like to try to add type inference to this type system. [2]

Acknowledgments

I'd like to thank Dr. Elena Machkasova for providing guidance as my advisor, and Dr. Stephen Adams for reviewing an earlier draft of this paper.

References

[1] Luís Caires. 2014. *Types and Logic, Concurrency and Non-Determinism.* Technical Report MSR-TR-2014-104. Microsoft Research. 69–84 pages. https://www.microsoft.com/en-us/research/publication/essays-for-the-luca-cardelli-fest/

[2] Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Parallel Complexity Analysis with Temporal Session Types. *Proc. ACM Program. Lang.* 2, ICFP, Article 91 (July 2018), 30 pages. https://doi.org/10.1145/3236786

[3] Roberto Di Cosmo and Dale Miller. 2019. Linear Logic. https://plato.stanford.edu/archives/sum2019/entries/logic-linear/

[4] Deepak Garg and Frank Pfenning. 2005. Type-Directed Concurrency. In *CONCUR 2005 – Concurrency Theory*, Martín Abadi and Luca de Alfaro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 6–20. https://doi.org/10.1007/11539452_5

[5] Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, Eike Best (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 509–523. https://doi.org/10.1007/3-540-57208-2_35

[6] Robin Milner, Joachim Parrow, and David Walker. 1992. A calculus of mobile processes, I. *Information and Computation* 100, 1 (1992), 1–40. https://doi.org/10.1016/0890-5401(92)90008-4

[7] Jerome H. Saltzer and M. Frans Kaashoek. 2009. *Principles of Computer System Design: An Introduction.* Morgan Kaufmann, Chapter 7.

[8] Wikipedia contributors. 2021. Concurrent computing — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Concurrent_computing&oldid=1040057524 [Online; accessed 29-November-2021].