Benedikt Maderbacher, BSc

Proof-Based Development of Actor Systems

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Bernhard Aichernig

Institute of Software Technology

Graz, December 2019

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date	Signature

Abstract

Concurrent and distributed software is widespread, but is inherently complex. There exists many styles to program such software, while trying to keep the complexity reasonable. One such paradigm is actors, it is used in the programming language Erlang, the Akka framework, and many others. It avoids the common pitfall of shared mutable state and interprocess communication is done via asynchronous message passing between processes, known as actors. Even though actor systems avoid many common errors, for critical systems we might still want to formally verify them.

This thesis describes a technique to model and verify actor systems using the formal method Event-B. When formally verifying a program, in most formalisms, one writes a specification and then proves that the program conforms to this specification. Event-B deviates from this pattern. The program is developed in multiple steps, where each step is a program itself. Each subsequent program is a refinement of the previous one, meaning that it is a more concrete and detailed description of the same system. This relationship needs to be formally proven in a proof assistant. Instead of one specification for the system, each program acts as the specification of the next one. Because of transitivity, we can conclude that the last program is a correct refinement of the initial one. The stepwise process of creating the models gives a high confidence into the validity of the specification.

The thesis covers three main topics. How can actor systems be modeled with the techniques available in Event-B and two broad strategies of how to develop an actor system. In the first case a mathematical specification is expanded into a concurrent actor program. In the second case an actor program is developed from a minimal actor program by adding more details in each step. Both of those are presented as a case study.

Kurzfassung

Verteilte Computerprogramme sind heute weit verbreitet. Sie weisen jedoch eine hohe inhärente Komplexität auf. Es werden verschiedene Paradigmen eingesetzt, um dieser entgegen zu wirken und die Entwicklung von verteilten Programmen beherrschbarer zu machen. Ein weitverbreitetes Paradigma sind Aktorensysteme. Sie werden in der Programmiersprache Erlang, dem Akka Framework und in anderen Plattformen verwendet. Die häufigsten Probleme, die durch gemeinsame und veränderbare Speicher verursacht werden, lassen sich damit umgehen. Aktorensysteme beruhen auf Kommunikation via asynchronen Nachrichten. Obwohl sich viele Fehlerquellen durch die Verwendung von Aktorensystemen ausschließen lassen, ist es für sicherheitskritische Systeme empfehlenswert, diese formal zu verifizieren.

Diese Arbeit präsentiert Techniken, um Aktorensysteme mit der formalen Methode Event-B zu modellieren und zu verifizieren. In den meisten formalen Methoden, wird eine Spezifikation erstellt und dann bewiesen, dass ein Programm diese Spezifikation erfüllt. Event-B unterscheidet sich in diesem Punkt von anderen Techniken. Ein Programm wird in mehreren Schritten entwickelt, wobei jede Zwischenstufe selbst ein Programm ist. Jede Stufe ergänzt das Programm um weitere Details, im Englischen wird dies Refinement genannt. Die Korrektheit dieser Refinement Relation wird mit einem interaktiven Theorem-Beweiser formal bewiesen. Anstatt die Spezifikation für das gesamte Programm auf einmal zu schreiben, wird diese in mehren Schritten aufgebaut. Jedes Programm dient als Spezifikation für das Programm der nächsten Stufe. Die Transitivität der Refinement Relation erlaubt es uns den Schluss zu ziehen, dass das fertige Programm eine korrekte Implementierung des ersten Programms ist. Durch die schrittweise Entwicklung der Programme kann ein hohes Vertrauen in die Richtigkeit der Spezifikation gesetzt werden.

In dieser Arbeit werden drei Hauptthemen behandelt. Zuerst wird untersucht wie Aktorensysteme in Event-B modelliert werden können. In weiterer Folge werden zwei Techniken für die Verifikation von Aktorensystemen beschrieben. Die erste Technik erlaubt es, ein Aktorensystem aus einer mathematischen Spezifikation zu entwickeln. Bei der zweiten Technik wird mit einem minimalen Aktorenprogramm begonnen. Das vollständige Programm wird aus diesem entwickelt, indem in mehreren Schritten weitere Details ergänzt werden. Beide Techniken werden anhand von Fallstudien präsentiert.

Acknowledgements

I want to thank several people who helped me during my studies and while preparing this thesis. At first, my thesis supervisor Bernhard K. Aichernig, he first showed the beauty and elegance of functional programming and formal methods during my second year of study and has supported me ever since. He suggested this interesting topic at the intersection of formal methods and programming languages and was always open for inspiring conversations.

I also want to thank Christoph Maurer and Rudolf Wörndle for listening to my endless explanations of my work. This helped me to organize my thoughts in a way that is (hopefully) understandable to computer scientists, who are not already head deep into the topic.

Last but not least, I am grateful to my family, especially my parents Elisabeth and Roland, for their encouragement and constant support during my studies. Additionally, I would like to thank my mother for proofreading this thesis. A great THANK YOU to all of you.

Benedikt Maderbacher
Graz, Austria, December 12, 2019

Contents

List of Figures

List of Listings

1. Introduction

1.1. Motivation

Modern computer systems rely heavily on concurrent and distributed software. Classic techniques using shared mutable state and explicit synchronization mechanisms are not ideal for these tasks. Instead, many systems are written using techniques that are designed to handle the challenges inherent to concurrent programs. A model that is widely used in this area are actor systems [52]. They are based on asynchronous communication via message passing. Each actor or process has its own memory and state that is isolated from the rest of the world. All interaction is done by sending messages between actors. This concept has been implemented in various programming languages such as Erlang [12, 11] as well as in frameworks for other languages such as Akka [69] for Scala [80] and Java [13]. Many well-known distributed systems use various actor implementations in their backend. This includes network infrastructure by Cisco [24] and Ericsson's telecommunication systems [50]. The messenger WhatsApp uses Erlang on its servers [94, 73]. Other usages of actors include various online games, for example LeagueOfLegends [35]. Actor systems can also be used to describe other distributed systems such as IoT devices.

Actor systems can help to prevent many common bugs in concurrent programming, such as data races or deadlocks, but they do not guarantee that the software is correct. There are still many possibilities to introduce errors in software written with actors. The usage of such systems in critical areas such as communication systems makes them an attractive target for formal methods.

Formal methods use mathematics and logic to model and analyze hardware and software. They aim to find errors or certify the conformance to a specification. This techniques help to create software with fewer errors. Large companies, such as Amazon [77] and Microsoft [18], use formal methods to improve the quality of their software.

In this thesis we will explore how the formal method Event-B [2] can be used to verify actor systems.

Actor Systems

The main component of the actor systems concurrency model are so called actors. These are similar to processes or threads but they cannot access any shared memory. Each actor can have its own local memory. Actor communicate by sending messages. An actor who receives a message can do three kinds of actions. It can send messages to other actors. Create new

actors. Or change its own state. While an actor performs computations triggered by one message, no other message can interrupt it. This allows actors to avoid the classic data race problem [86, 25].

Event-B

Event-B is a modeling language and formal method based on set theory. One writes a model that captures the important behaviors of a system. Instead of directly verifying a computer program. An Event-B model contains machines. A machine has a state and guarded events that can change this state. The model is developed by using step wise refinement. At each step a new machine is created that is a refinement of the previous one. It is a more concrete version that contains more details and is closer tp the modeled system. For each step a formal proof is required to demonstrate that this refinement relation holds.

1.2. Problem Statement

In this thesis we want to answer the following research questions.

Can actor systems be represented in Event-B? We want to model actor systems that can use the distinguishing features, spawning actors and dynamic network topology. Can a model of an actor system in Even-B support those features? If yes what is a good representation of actors and messages? What are advantages and disadvantages of different encodings? Is there a single best way to describe actors in Event-B? Or should the choice of the actor encoding depend on the program we want to verify? How can we ensure that a Even-B model corresponds to an actor system? What operations are allowed or forbidden in Event-B actor systems?

How to use refinement to build actor systems? We will not develop and define a refinement relation for actor systems, based on a formal semantics for them. Instead we want to use the refinement relation for Event-B machines. This question should be seen as more applied. We do not want to answer the question is one actor system a refinement of another one. For this we use Event-B's semantics and formalized refinement relation. Instead we want to explore refinement strategies for actor systems. Event-B advocates the usage of a stepwise refinement process. There is not only a specification and an implementation, but many steps in between. Each of them adds new details to the model. A refinement strategy describes, which details are added in a step and how one model relates to the one above it in the refinement hierarchy. Abrial [2] categorizes refinements in Event-B into two classes, namely horizontal and vertical refinement. Horizontal refinements extend the model with new functionality whereas vertical refinements transform a model to be closer to an implementation. We plan to explore how these two classes of refinement can be applied to actors.

Can these techniques be used to model and verify non-trivial actor systems? We want to apply our techniques to two case studies. One that uses mostly a vertical refinement strategy and another one that uses primarily horizontal refinement. The cases studies

should be non-trivial in the sense that they require all features of actor systems. This should include creating actors at runtime, exchanging actor addresses to achieve a dynamic topology and sending complex messages with multiple data fields.

Are there correctness properties that are not covered by this verification approach? If yes, what are they and how can we ensure they hold anyway?

1.3. Contribution

We developed multiple encodings for actor systems into Event-B. They differ in the features they support. For some systems a simpler encoding can be used if certain features are not required. An actor that will never receive two messages at the same time can use a simpler mailbox than an actor who must support multiple concurrent messages. All our encodings use Event-B state variables to store mailboxes and actor states. Receiving and sending of messages is done by events that access the mailbox variables. To send a message to some other actor an action modifies the receivers mailbox variable to include the newly sent message. The most complete actor representation supports creating actors, sending composite messages that can also contain actor addresses, and local actor states.

Based on these encodings for actors we developed two general refinement strategies. One corresponds to vertical and one to horizontal refinement. In the first case an actor system is derived from a mathematical specification. In the second case we start with a simple actor system and add more features in each refinement. We implemented and proved an case study using each of the two techniques.

For the vertical refinement we implemented an actor system to compute the factorial function. This is inspired by an example in Agha's PhD thesis [6]. The model demonstrates the dynamic creation of actors, representing the stack, by an iterative algorithm. The iteration is done by sending a message to itself. For this case study we created two versions, a sequential and a concurrent implementation. In both cases we start with a mathematical specification. The refinements turn this into an iterative program, an iterative program with a stack, and finally into an actor system. The sequential case only models a single execution of the algorithm. It contains a formal termination proof. The concurrent version can handle arbitrary many concurrent requests. It was developed using the insights gained from the sequential version. For the concurrent model no formal termination or liveness proofs where done.

To demonstrate the horizontal refinement strategy we implemented a chat server. This case study was inspired by an example from the Akka documentation [69]. The system consists of one server and multiple clients. Clients can subscribe to the server and subscribed clients can send messages. These messages are then distributed by the server to all subscribed clients. When a client subscribes, a new session actor is created that mediates all future communication between the server and this client. The refinement process starts with a minimal client server case study. A client can send a message to the server who sends the same message back. In the refinement steps we introduce multiple clients, user names, an asynchronous subscribe feature, and the session actors. This model contains all actor

features. New actors are created and the topology is changed to include them. A session stores the address of its client and the server, and the server keeps an updated list of all subscribed clients.

We created a formal termination proof for the sequential factorial model. This proof uses variants and convergent events. For the concurrent factorial case study and the chat server no termination proof is possible, because the systems are supposed to run forever. Instead the relevant correctness property is liveness. The formal proofs guarantee that if a message is send it will be the correct message. However, it might happen that some message is not sent. This would be a liveness violation. We used manual testing and the ProB [67] model checker to check relevant liveness properties of our models.

1.4. Structure of this Thesis

This thesis is structured as follows:

Chapter 2 covers the semantics of classic actor systems, how actors are implemented in different programming languages and what actor semantics will be used throughout the thesis. In this section we also explain the Scala library Akka and UML sequence diagrams. These are later used to demonstrate and visualize actor systems.

Chapter 3 explains the Event-B formal method. This includes notations that are required to read and understand Event-B models. We also describe Event-B's semantics and refinement model. The final part of this section encompasses a brief overview of the ProB model checker and temporal logic. As we will use them to analyze liveness properties of our systems.

Chapter 4 describes different representations for actors in Event-B and criteria for deciding if an Event-B model is an actor systems. This chapter also contains general descriptions of our refinement strategies and how to best perform the required proofs.

Chapter 5 covers the first case study. A program to compute the factorial function, that is derived from a mathematical specification. This chapter contains two major parts. In Section 5.2 a sequential version is developed, including a formal termination proof. In Section 5.3 the previous model is adapted to a concurrent version.

Chapter 6 contains the second case study. A chat server that supports multiple clients. The clients can subscribe to the server and communication is buffered by session actors.

Chapter 7 summarizes the thesis and relates it to preexisting work. These cover different verification techniques for actors as well as prior work in Event-B that is similar to actor systems. This chapter concludes the thesis and suggests possible extensions.

There are two appendices. Appendix A encompasses the complete model of the sequential factorial case study and the last machine of the concurrent variant. Appendix B contains the final machine of the chat server case study.

2. Actor Model

The actor model is a paradigm for concurrent and distributed programming. It was first described by Hewitt et al. [55, 53, 54, 7]. The central idea is that computation can be modeled by multiple agents or actors communicating with each other. This communication is done by sending asynchronous messages. Upon receiving a message, an actor processes it and may send new messages to other actors. Thus, an actor system is a network of actors that perform computations and exchange messages. Actors are unable to access or influence other actors, except by sending them a message. This most importantly includes that no actor can read or write the memory of another actor. As a consequence, no locking of memory is required. Race conditions and many deadlock situations are eliminated. However, a ring of actor may still wait for receiving a message in a circular way.

The advantages of actor systems over low level concurrency primitives (threads and mutexes) have led to many modern programming languages including support for actors. Most famous is the Erlang programming language which includes actors as a language feature. Other languages that are based on actors are Elixir [91] which runs on the same virtual machine as Erlang [11], or the object-oriented language Pony [43]. Not all languages include actors as a language primitive, a wide range of languages provides libraries or frameworks to support actor style concurrency. Two better known implementations of that kind are Akka [69] for Scala and Java, and Orleans [33] for .NET.

Having so many implementations of actors, means that it is sometimes difficult to understand what is actually meant by the term. De Koster et al. [44] give a taxonomy of actor systems. In the remaining chapter we present the classic model of actors as developed by Hewitt [52] and Agha [6] (Section 2.1). We also give a short overview of practical implementations and how they differ from the theoretical framework (Section 2.2). In Section 2.3 we present and justify the actor semantics used in this work. To present examples in an executable form, we use Scala and the typed version of Akka, instead of pseudo code throughout this thesis. Section 2.4 gives a short overview of Akka. To visualize computations in an actor system, we use UML sequence diagrams, these are explained in Section 2.5.

2.1. Classic Actors

In this section we explain classic actor systems as defined by Agha [6] and Hewitt [52]. An actor system consists of two kinds of components, actors and messages. Each actor has a unique name, or address. A message can only be sent to an actor if its address is known to the sender. There is no way to forge an address, guess it or query the system for it.

How an actor reacts, upon receiving a message, is defined by its behavior. It may do one or all of the following three operations:

Send a message to another actor whose address is known. An actor may also send multiple messages to the same or different targets.

Change its behavior for future messages. This also includes changing an internal state.

Create a new actor. When creating a new actor, the parent actor automatically learns its address. It is also possible to create more than one actor at a time.

An actor can only perform computations or execute one of these operations, when it receives a message. It may not start doing some work on its own volition. However, messages from outside the system can start actions.

All operations that happen while handling a message are atomic. An outside observer or another actor cannot observe the computations happening. The actor will also complete all computations necessary to handle a message, before it starts processing the next message. There is no way to interrupt the handling of a message.

Messages are sent asynchronously, meaning that an actor is free to continue its computations immediately, after a message was sent. It does not block till it receives an answer, as would happen with a regular function call. An actor may receive multiple messages before it finished handling the first message. To handle this gracefully, actors include message buffers, so called mailboxes. They store all messages sent to an actor until it is able to process them. Agha generally assumes the presence of such a mailbox [6]. Hewitt on the other hand argues that mailboxes are not a fundamental part of an actor, instead they can be modeled as actors themselves [52].

Sending a message to another actor requires that the sender knows the address of the recipient. It can learn of it by two ways. An actor spawning a new one automatically knows the address of the newly spawned actor. The reverse is not true, a child actor does not automatically know the name of its parent. The other way is to send an address via a message to another actor. Every actor can include any name it knows in a message, the receiver then learns this address. In systems where no new actors are created or where a set of actors exists at the start of the program, we often assume that the addresses have been exchanged before, or are simply hard coded. We define that at startup actors know exactly the addresses they will need.

To ensure that actor addresses are unique in a distributed system without a central authority, Agha proposed the following scheme [6] to generate names. The name of an actor is actually a path that encodes how it was created. Every system starts with a root actor, responsible for creating the other actors required at the beginning. It has some defined name for example ”root”. Every actor also contains a counter that it increments, whenever it spawns a new actor. The newly created actor gets the address of its parent with the current state of the counter appended. A specific actor might thus have the address ”root/actor1/actor5”, it is the fifth actor created by the first actor that was created by root. In this process every actor is only responsible for ensuring that the last part of the name of each of its children is unique. A globally unique name is ensured by the fact the creation path is part of the address.

In the classic actor model there are certain assumptions about the delivery of messages and when they are processed. There is no message loss, a message sent to an actor is always delivered. There is no restriction on the order of messages, messages may arrive in any order. Even messages sent between the same two actors can arrive in a different order than they were sent. An actor processes messages in the order in which it receives them, though removing this constraint would not change the behavior of a system because there are no guarantees on message order. Many concurrent algorithms require some form of fairness to ensure termination or liveness [66]. In the actor model the only fairness assumption is that messages are eventually delivered. There is no guarantee when an actor will process a message in its mailbox or if it will ever process the message. If this property is necessary, to prove the correctness of a system, the formalism can be extended by a more appropriate fairness assumption.

2.2. Alternative Formulations

Other concurrency models that belong to the wider category of actor systems are active objects and Erlang's [11] processes. The more niche model of communicating event-loops that is used in the programming languages E [74] and AmbientTalk [92], will not be discussed in this work.

Active Objects

Active objects were developed by Yonezawa [97]. Instances of this formalism are the original implementation ABCL/1 [97], Asynchronous Sequential Processes [34], and the frameworks SALSA [93] and Orleans [33]. There are also modeling languages in formal methods such as Johnsen et al.'s Creol [63] and ABS [62] languages, or the language and model checker Rebeca [90, 88]. De Boer et al. [46] provided a review of active object and actor languages, from the perspective of the formal methods community.

The active objects version of actor systems is based on imperative and object-oriented programming. Each actor or active object has its own isolated heap. Changes of the state are not done by changing the behavior, as in the classic actor model, instead each active object runs an imperative program that can use mutating assignments on its own heap. Messages are function calls on an active object, they are always performed asynchronously. Passive objects are objects that do not support asynchronous messages, when a message is sent, these passive object are passed by copy between the active objects. Even though the handling of a message is done by an imperative program, it cannot be interrupted. This means for an outside observer, such as another actor, the processing of a message is still atomic.

Processes

The processes used by Erlang [12, 9, 11] and other languages running on the same virtual machine, for example Elixir [91], are very similar to classic actors. They are based on functional programming. Each process can use a receive-statement with a pattern matching. The process blocks until it receives a message that fits one of the expected patterns. All other messages are stored in the mailbox. When a message is received, a functional program is executed and it cannot be interrupted. After the message is processed, the process terminates, alternatively it can recursively call itself at the end, to continue processing messages.

A paradigm that originated in Erlang is ”let it crash” [10]. It is often associated with actor systems, but is not really part of the core concepts of actors. In Erlang processes are very cheap in terms of memory footprint and other resources. A common strategy is to let a process crash, when something unexpected happened and simply restart it. This is done by using supervisor actors that monitor the actors they spawned and restart them if required. A supervisor can have a supervisor of its own, a system often contains a supervisor tree that is a hierarchy of supervisors.

2.3. Used Actor Semantics

In this work we will use an actor semantics that is mostly based on the classic model of actor systems.

Every actor contains an unbounded mailbox. This follows Agha’s interpretation, but deviates from Hewitt’s. It also does not match the reality of actual programs as no computer has infinite memory to store those messages. We assume, however, that in a realistic system an actor will always have enough memory to store the messages it receives. Avoiding an arbitrary bound on the mailbox size greatly simplifies proofs because detecting and handling an overflowing mailbox can be ignored.

Updating the state of an actor is atomic. All actor variants ensure that temporary state changes made while processing a message are not observable from outside the actor. Making all state changes in one step allows us to model all those systems, except for complicated computation inside an actor. We however aim at modeling and verifying the communication between actors and not the actual processing of data. Not being able to execute arbitrary sequential programs inside an actor does not affect this goal.

We assume that the interface of each actor is determined at creation and cannot change. Meaning that an actor always accepts the same type of messages. These messages can be algebraic data types. Actors can hold state and perform different operations based on the values in the state and the message it received. The value of a state variable can be changed, but no new variables can be created later on. State variables can also not be removed, the set of variables is determined statically. This somewhat restricts what can be modeled with our actor system. We argue, that programs, where an actor needs to

completely change its behavior, are rare. Many interesting problems can be modeled with these assumptions in place.

Actors cannot crash or be destroyed. Sending a message to an actor that it cannot handle results in an invalid model that does not pass verification. Instead of modeling crashing and restarting actors, we aim to create and verify protocols where this cannot happen.

Messages are delivered to the mailbox of the receiver immediately. There is no possibility of message loss. Processing of messages can be done out of order, it is done nondeterministically. This does not match the semantics of the actor systems described above. However, the combination of unbounded mailboxes, no message loss, out-of-order message processing, immediate delivery, and the fact that only the actor itself can see the content of its mailbox, means that this is indistinguishable from the classic actor model. It does not matter if a message takes a long time to be delivered, or if it waits in the mailbox for a long time. For any observer those are the same. Event-B does not guarantee that a message is actually processed. If this property is required, we use explicit fairness assumptions, this is explained in Section 3.6.1.

The assumption that a message cannot be lost is the same as in the classic model. This limits our ability to model distributed systems that cannot guarantee this property. In this work we assume that resending messages based on, for example, a handshake protocol, is done on a layer below our actor system.

The constraint that an actor can only do something if it receives a message, is relaxed. We allow certain actors to send messages at any time. This is used to model actors at the boundary of the system, such a spontaneous message is assumed to be triggered by something outside the model, for example a user input.

2.4. Akka Typed

To present small actor programs inside this thesis, we use Scala code and the Akka typed library [70]. We use these instead of pseudo code because there is no (semi)standardized pseudo code for actors and to give executable examples. The Scala code is also very terse and requires minimal boilerplate. The examples in this thesis were tested with Scala in version 2.12.7 and Akka in version 2.5.18.

This section gives a short overview of the used subset of the Akka typed library. Listing 2.1 shows a simple program where two actors named ping and pong exchange a counter that gets incremented each time a message is sent.

In contrast to many actor languages the messages are typed. We need to create a case class for each type of message we want to send. Every actor and every actor address is parameterized with the type of messages it accepts. The type system ensures that only correct messages are sent. This matches our assumption, from the previous section, that wrong messages cannot be sent.

```
object Example {
  final case class Message(value: Int, replyTo: ActorRef[Message])

  def actor(name: String): Behavior[Message] =
    Behaviors.receive { (context, message) =>
      context.log.info("{} received: {}", name, message.value)
      message.replyTo ! Message(message.value+1, context.self)
      actor(name)
    }

  final case class Start()

  val main: Behavior[Start] =
    Behaviors.setup { context =>
      val ping = context.spawn(actor("Ping"), "Ping")
      val pong = context.spawnAnonymous(actor("Pong"))

      Behaviors.receiveMessage { message =>
        context.log.info("Send message \"1\" to Ping")
        ping ! Message(1, pong)
        Behaviors.same
      }
    }
}

val system: ActorSystem[Example.Start] =
  ActorSystem(Example.main, "main")

system ! Example.Start()
```

Listing 2.1: Akka example

Looking at the case class `Message`, we see that it has two fields. An integer value and a field name `replyTo` of type `ActorRef[Message]`, this is the address of an actor with the information what type of messages it accepts, encoded at the type level.

The behavior of our actor is named `actor` and has one parameter `name` of type string, this is the state of this actor. The type `Behavior[Message]` tells us that this is a behavior for an actor that can receive messages of type `Message`. A behavior is some kind of blueprint from witch an actor instance can be created. How a message is processed is defined with `Behaviors.receive`, it takes a function as an argument. This function has two parameters, the context and the received message. It will be called whenever the actor receives a message.

The context object can be used to create log messages or to create new actors. To send a message to another actor the bang (`!`) infix operator is used. The first argument is an actor address and the second one the message, the types of both must be compatible. At the end of the receive-block we need to define the new behavior of our actor. This can be done by calling a behavior of the same type, possibly with different parameters, or we can

use the shorthand `Behaviors.same` to not change the behavior.

The second behavior in this example is `main`, it accepts a start message and is the root actor of our system. It uses the context object to create two instances of our first actor behavior. This is done using `context.spawn( )`, we can either provide a name for the new actor ourselves or let the system choose one by using the `spawnAnonymous` version. Both take a behavior as parameter, all parameters of this behavior must be provided.

Using `Behaviors.setup` and `Behaviors.receiveMessage` allows us to define code that only gets executed when the actor is created and not on every message it receives.

Finally, we create an actor system with our root actor and send it the start message to kick of a game of ping pong.

In later examples we will omit the root actor and the code to create all actors. Only the message types and the behavior definitions are included in the other examples of this work.

2.5. UML Sequence Diagrams

To better visualize the execution of an actor system, we use UML sequence diagrams [85, 51]. They are similar to the actor event diagrams used by Agha [6]. Figure 2.1 shows the first three messages of the ping pong example from the previous section. (The setup part is omitted.) These diagrams show all actor instances and sent messages in chronological order, from top to bottom.

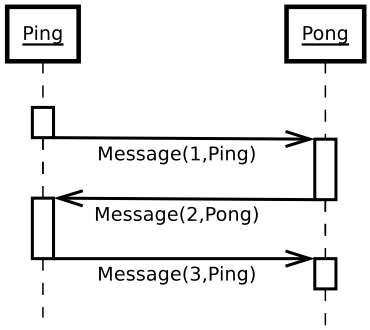

Figure 2.1.: Ping pong example

Each instance of an actor is represented by a box containing the name (address) of the actor and one vertical live line. Actor messages are drawn as arrows with an open head, asynchronous messages in UML, the message name and its parameters are shown as the label. Small boxes on the live line show, when an actor is active and processing a message. They start with an incoming message, or are activated by something outside the model, e.g. a user action on a client. We assume that an actor does not receive a message, while it is processing another message. Messages may only arrive at the start of an active

block. Sending messages is allowed anywhere in the block and it ends after the last action performed in response to the incoming message.

Creating or spawning an actor is done, using an arrow with a solid head and the label `<<create>>` that points to the box at the top of the new actor. This is shown in Figure 2.2. When an actor contains state variables, they are shown below the name of the actor.

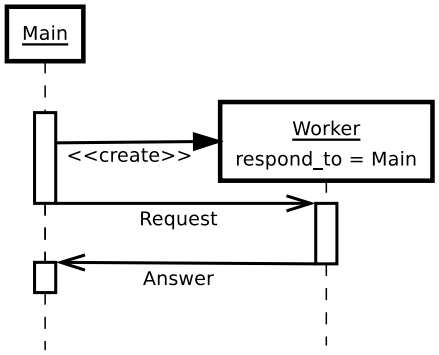

Figure 2.2.: Actor creation example

Some diagrams show models that do not completely follow the actor style because they are from an early stage of the refinement process. In such a model an actor might directly write to the memory of another actor; this is drawn as a synchronous communication: in UML an arrow with a solid head.

When an actor receives a message and just removes it from its mailbox, this is drawn as a synchronous message to itself, when it is important to assign a label to this action. This might model the actor displaying the content of the message to a user.

The addresses of other actors are only included in the state if they are created at runtime.

3. Event-B

Event-B is a formal method and modeling language. It is used to model systems on a more abstract level. One builds a model that describes the features that are most relevant to the system and the used specification. This is in the tradition of the Vienna Development Method [64], other similar formal methods are Abrial's B [3], Lamport's TLA+ [66], and Jackson's Alloy [60]. All of those formal methods also use set theory as their primary way of description. This is in contrast to proof assistants that often focus on logic and type theory. The prime examples of this other school are Coq [38] and Isabelle [79, 78].

Focusing on a smaller model that only captures the key properties of the system, reduces the complexity of verification. Omitted computations and data processing cannot interfere with proofs. Using this smaller more abstract models, allows one to handle bigger systems that would be infeasible to verify on the code level.

The Event-B method has evolved as a combination of Abrial's B [3] with Back's action systems [14]. A model consists of a global state that can be modified by nondeterministically executed guarded events. So it is also related to Dijkstra's guarded command languages [48] and abstract state machines [26]. The distinguishing feature of Event-B is the focus on refinement between models. Instead of writing a single model and verifying it against a specification, one writes multiple models from abstract to more concrete ones. Each model needs to be a refinement of the previous one. The abstract model acts as the specification of the concrete one. This is called a stepwise refinement process. At first one defines the set of requirements for the complete model. Then a refinement strategy is devised where in each step a model is created that is a refinement of the previous one and also incorporates another requirement. Figure 3.1 illustrates this process. In reality one refinement step might model more than one requirement, or a requirement might be developed over multiple steps.

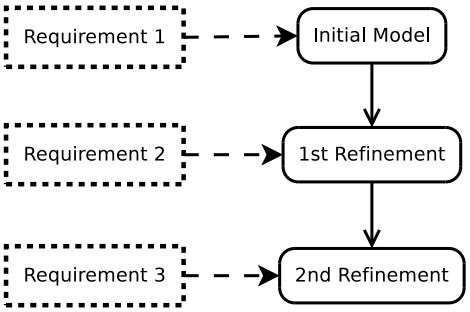

Figure 3.1.: The stepwise refinement process.

The remainder of this section will contain an overview of the set theory notation used in Event-B (Section 3.1). A description of the syntax of machines and contexts is provided in Section 3.2. An informal explanation of Event-B's operational semantics is given in Section 3.3. Section 3.4 defines the used refinement approach. In Section 3.5 the Rodin IDE and its tools are presented.

3.1. Set Theory

When writing a model in Event-B, one has access to the full range of set theory and predicate logic. This section describes the available operations, with a special focus on those that are not commonly encountered. The definitions in this section are from Robinson [84].

Event-B supports integers and natural numbers as primitives. This includes the full range of arithmetic: addition, multiplication, exponentiation, etc. These are written as infix operations, the same way as in school mathematics. We therefore will not show a list of these operations.

First order logic is also supported. This includes Booleans, with the operations *and*, *or*, *negation*, *implication*, etc., as well as *for all* and *exists* quantifiers. Event-B supports unicode, the logic operations are displayed the same way as in mathematics.

The set theory portion is based around carrier sets. These sets contain abstract objects, that have no properties except the inclusion in their carrier set. There is a primitive type checker to ensure that objects of different carrier sets cannot be mixed. If nothing else is defined, a carrier set is infinite. To define a finite set or a set containing a specific finite set of object, one can define their own axioms. Axioms are also used to define properties like an order relation. The available operations include among others: *union*, *intersection*, *difference*, *membership*, and *subset*. All of them are written as in mathematics.

To build compound structures one can use the cartesian product of two or more sets. This is written as $S \times T \times U$, as usual. The operation is left associative. For constructing a tuple of objects, a relatively uncommon syntax is used: $s \mapsto t \mapsto u$. There are two ways to deconstruct them. The functions fst and snd are projections of the first or second element respectively. The $\mapsto$ operation is left associative, meaning that $snd(s \mapsto t \mapsto u) = t \mapsto u$. Another more comfortable way is to write an equation that works like pattern matching: $tuple = s \mapsto t \mapsto u$. The variables s, t, u must be introduced as parameters or by a quantifier for this to work.

Relations are sets of ordered pairs. Event-B allows us to assert many properties of relations, by using different syntax variants. For relations the following styles are available:

relations	$S \leftrightarrow T$
total relations	$S \nleftrightarrow T$
surjective relations	$S \twoheadleftarrow T$
total surjective relations	$S \nleftrightarrow T$

where S, T are sets.

Functions are a specific kind of relation in which the first elements of the pair do not contain duplicates. The following styles of functions are supported:

partial functions	$S \pfun T$
total functions	$S \fun T$
partial injective functions	$S \pinj T$
total injective functions	$S \inj T$
partial surjective functions	$S \psur T$
total surjective functions	$S \sur T$
bijective functions	$S \bij T$

where S, T are sets.

When writing a model, it is often advantageous to specify the exact properties of a function or relation. These properties can then be used in proofs.

Relations and functions can be combined, using normal set operations, as long as the properties are obeyed. Apart from the normal set operations there are specific operations for relations and functions. These are not as common as previously covered operations, therefore we include them and their definitions below.

domain restriction	$S \domres r$	$=$	$\{x \mapsto y \mid x \mapsto y \in r \land x \in S\}$
domain subtraction	$S \domsub r$	$=$	$\{x \mapsto y \mid x \mapsto y \in r \land x \notin S\}$
range restriction	$r \ranres T$	$=$	$\{x \mapsto y \mid x \mapsto y \in r \land y \in T\}$
range subtraction	$r \ransub T$	$=$	$\{x \mapsto y \mid x \mapsto y \in r \land y \notin T\}$
overriding	$r \ovl r'$	$=$	$r' \cup (\mathrm{dom}(r') \domsub r)$

where S, T are sets and $r, r' \in S \leftrightarrow T$

A comprehensive list of the special syntax available in Rodin is available in Robinson [84], the presentation and the selection of operators in the tables above was inspired by Kann [65].

3.2. Machines and Contexts

An Event-B model can consist of two building blocks: machines and contexts. A model can contain multiple instances of both. Contexts are used to extend the mathematical theory used in a model. It contains the definitions of carrier sets and constants. A constant is a named value that is independent of the state of the current execution. Constants can also be used to define functions that can be recursive. In all those cases the values and properties of constants and sets are stated as axioms. These predicates are defined to be always true. For example, an axiom might state that $one = 1$. To use the definitions from a context, we can import it into a machine or another context, using a *sees* definition.

Machines are the part of the model describing the actual system. They consist of state variables, invariants, variants, and events. State variables define the possible states a machine can be in. Each variable can have different values depending on the point of the execution. The state of the machine is formed by all the state variables.

Invariants are predicates that restrict and describe the state. They contain the state variables and define what values they might have. An invariant must always hold and it might talk about multiple state variables. For example $x > y$. Each state variable must have an invariant that assigns it to a specific set. This is also referred to as its type. Most often, this is an expression like $x \in S$, but it can also be inferred from more complex expressions.

Events consist of parameters, guards, and actions. Parameters are similar to function parameters in a programming language, but they are optional. When an event is executed, valid values for the parameters are chosen, and they can be used inside the event. Guards are predicates that determine if an event is allowed to be executed. They can refer to state variables and event parameters. If an event contains a parameter, there must be a guard that describes its type. Similar to how invariants determine the type of a variable.

Actions form the body of an event. They determine the state after the execution of an event, based on the previous state and the parameters. Each action determines the value of one state variable. Each state variable may only be set by one action. If a variable is not set by any action, it is unchanged. The most commonly used action is assignment. Such as $x := v$, this assigns the value v to the variable x. Actions can also be a nondeterministic choice. The notation $x :| \mathcal{P}$, where x is a variable and $\mathcal{P}$ is a predicate involving x, assigns to x any value such that $\mathcal{P}$ is true. This is the most general form of an action. It can be used to express all other forms of actions. Assignment could be written as $x :| x = v$.

There are also more specialized actions. To assign any value from a set to a variable, we can use the action $x :\in S$. A value from S is nondeterministically chosen and assigned to x. This is equivalent to $x :| x \in S$. Another specialized action is function overriding. It can be used to change a single point of a function state variable. The notation is $f(a) := b$; with f being a state variable of a function type, a being a value from the domain of f and b a value from the codomain of f. This changes the function such that a evaluates to b and it evaluates the same for all other values. It is equivalent to $f := f \Leftarrow \{a \mapsto b\}$.

There is a special event *Initialisation*; it has no guards and no parameters. It must assign all state variables and defines the initial state of the machine.

Listing 3.1 shows a simple Event-B machine with one variable and three events. The variable is called *counter* and is of type $\mathbb{N}$ as defined by $inv0$. The second invariant restricts *counter* to the range $[0, 3]$. The *Initialisation* event assigns the initial value 0 to *counter*. The event *Increment* has no parameter, the guard requires *counter* to be smaller than 3 and the action increments counter by one. The *Subtract* event has one parameter *amount*. The guards define that *counter* must be at least 3 and that *amount* must be in the range $[1, counter]$. The action reduces *counter* by *amount*.

VARIABLES
counter
INVARIANTS
inv0: $counter \in \mathbb{N}$
inv1: $counter \geq 0 \land counter \leq 3$
EVENTS
Initialisation
begin
act0: $counter := 0$
end
Increment ⟨ordinary⟩ $\widehat{=}$
when
grd0: $counter < 3$
then
act0: $counter := counter + 1$
end

Subtract ⟨ordinary⟩ $\widehat{=}$
any
amount
where
grd0: $counter \geq 3$
grd1: $amount > 0 \land amount \leq counter$
then
act0: $counter := counter - amount$
end

Listing 3.1: Event-B example

3.3. Semantics

This section gives an informal description of the semantics of Event-B machines.

When a machine is executed, at first, the *Initialisation* event is applied. This assigns all state variables to a fixed value. This initial state must satisfy all the invariants.

After the initialization and for as long as the machine is running, one event after another gets executed. To do so, one event, where all its guards are true, is chosen nondeterministically. If the event contains parameters, values are chosen for them nondeterministically, such that the guards are still true. An event in which all guards are satisfied by the current state, is called to be enabled.

To execute the chosen event, all its actions are executed. If an expression in an action contains a variable, the value of this variable is always the one defined by the previous state. There is no sequencing between actions, all of them happen at the same time.

Subsequently, the next action is chosen; an action which is enabled in the updated state. This action is executed as well. This continues possibly ad infinitum. When at some point in time, there is no enabled action, the system is called to be deadlocked and the execution stops.

In terms of a denotational semantics, we can see the actions of an event as a before-after-relation of the machine's state. The state of the machine is a value from the sets of all possible machine states defined by the state variables and invariants. The before-after-relation of the whole machine is the conjunction of all event relations and their respective guards. This means that each Event-B machine is an (infinite) state machine, with the events as transitions.

3.4. Refinement

A core concept of Event-B is refinement. We can create new machines that are more concrete versions of existing machines. These concrete machines refine their abstract counter part. The behavior of the concrete machine must still conform to the one of the abstract machine. The notion of refinement used by Event-B is based on the refinement calculus [15] by Back and von Wright for action systems [14, 16].

This section contains a summary of Event-B's refinement as described in Abrial's "Modeling in Event-B" [2] Chapter 14. The complete and formal definitions can be found there.

3.4.1. Simple Refinement

This section gives an informal definition of refinement based on traces. A trace is a finite sequence of states. Two consecutive states are related by the before-after-relation of the machine. The before-after-relation is explained in Section 3.3. The first state of a trace is created by the *Initialisation* event.

For this simplified version of refinement, we assume that the state space of both machines is the same. In that case refinement is trace inclusion, with some limitations. The set of possible traces of the concrete machine must be included in the set of possible traces of the abstract machine. This definition would imply that a machine without any traces refines every other machine. But we still want the concrete machine to be similar to the abstract machine, so this would be too weak. Instead, we use the following notion: Given a concrete trace, we cannot tell if it originated from the abstract or the concrete machine. The reverse is not necessarily true. Additionally, we require a property called relative deadlock freedom. If an abstract trace can be extended, that means, we can execute more events after the last state, but the concrete trace cannot be extended the relative deadlock freedom property does not hold. In this case the concrete machine is not a refinement of the abstract one.

3.4.2. Proof Obligations

The semantics refinement property, we just discussed, is a property about the complete system. To prove it we would need to reason about the before-after-relationship formed by all events and also reason about arbitrary long traces. Fortunately such proofs can be split. It is only necessary to check some properties of the events. This can be independently done for each event and only involves the states before and after the event was executed. We call these properties proof obligations. Abrial's Modeling in Event-B [2], in Chapter 14, contains a proof that the sum of all proof obligations implies the semantics refinement property.

The most important proof obligations are:

Guard strengthening The guard of a concrete event must imply the guard of the abstract event. Guard means in this case the conjunction of all the guards of an event. This rule signifies that a concrete event might be enabled less often, but when it is enabled, the abstract event will also be enabled. So the number of potential traces can be reduced, but no new traces will be introduced.

Invariant preservation Given that all invariants hold in the state before an event is executed and all guards are true, then all the invariants must hold after the execution of the event. This means no event may lead to a state that is not part of the valid state space.

Simulation This proof obligation states that actions in the abstract event are properly simulated by the concrete event's actions. For example, if a state variable exists in both, the abstract and the concrete machine, the values of this variable must be the same. This is only relevant in the complete refinement theory, where the two state spaces are not required to be equal.

Well-Definedness This includes a wide range of properties, all of them guaranteeing that some kind of undefined behavior is avoided. Some examples: When dividing, we need to prove that we will never divide by zero. Or, when a partial function is called, we must demonstrate that we will only call it with values for which the function is defined.

There are other more specialized proof obligations that we will not cover in this summary. There are rules for *variants*, *witnesses*, and *nondeterministic assignments*.

Another interesting type of proof obligations are theorems. They are written similar to invariants but marked as theorems. This means they are not proven via the invariant preservation rule. Instead, they must be implied by the other invariants. For example, this can be used to state deadlock freedom; there must be always at least one event enabled.

3.4.3. Advanced Refinements

Actual Event-B models will require more than what was described as simple refinement above. We want to add new state variables or even represent the same information with

different state variables. This is possible in Event-B. When state variables are changed between refinements, the simulation rule becomes relevant. Adding new state variables requires that the old variables still match, new ones represent just additional state. When variables are removed, we need to use special invariants, so called gluing invariants. They relate the removed variables to the variables in the refined machine. The values of the disappeared variables, as determined by the gluing invariants, must match the values of the abstract variables.

It is also possible to introduce new events, these events are implicitly a refinement of a skip event. This is an event without guards and without actions. It can always be executed, but does not change the state. It is also possible to split or merge an event, but this is not used in this thesis.

The theory of refinement in terms of traces can be extended to include all of this [2].

For structuring our models, we differentiate between two types of refinement: horizontal and vertical. Vertical refinement is the simple refinement, but also includes changing variables to more accurately describe the state. The amount of possible interactions with the system stays the same. The other kind is horizontal refinement. This includes adding events and state variables to a machine. We extend what our model includes, but do not make the descriptions of the existing parts more accurate. Most real models include both of those. A typical strategy is to use horizontal refinement till all functionality is included and then use vertical refinement to make it more concrete.

3.5. Rodin

Rodin [4] is an IDE, based on Eclipse, to write Event-B models. It includes an editor to create contexts and machines. These are written in a structured format that can be unergonomic to use. There is a plugin called Camille [23], a text editor that allows one to write models as plain text. They are then translated into the internal Rodin format.

Rodin also includes an interactive theorem prover. It displays all proof obligations that need to be proven. We can then interactively expand definitions, rewrite expressions, define lemmas, and perform other proof steps. The proof-view shows the current goal at the bottom and all assumptions above.

Rodin supports, via plugins, many automatic theorem provers. This means for many proof obligations it is not necessary to prove anything by hand. Even proofs that need to be done interactively, greatly benefit from the automatic solvers. The two big solver plugins are Atelier-B provers [42] and the SMT solver plugin [47]. It supports the following SMT solvers: CVC3 [21], CVC4 [20], Z3 [45], and veriT [27].

The versions of Rodin, its plugins, and the standalone ProB version used in this thesis are given in Table 3.1.

3.6. ProB Model Checker

Another useful plugin is the ProB [67, 68] model checker. It can be used to simulate machines. There is a table showing the values of all state variables, including those of abstract machines. It also displays all enabled events and one can pick one of those and its parameters and execute it. A panel shows the trace of all previously executed events and we can roll back the state to any previous point in time.

ProB also supports random execution, invariant checking as well as deadlock checking. All this functionality can help to find counterexamples and improve the model. This is very important to decide if a proof is just difficult or if we might try to prove something wrong.

The most powerful feature of ProB is its ability to model check machines using a specification in linear temporal logic (LTL). Model checking means that it exhaustively explores all states and checks if they confirm to the given specification. A temporal logic allows us to describe the order in which events should happen in our model and if certain events are required to occur or shall never happen.

There are currently two different frontends for ProB that provide a different set of features. The older Rodin plugin and a new JavaFx based standalone client. The Rodin plugin can also start a standalone client with the machine preloaded. Both frontends can animate machines, by executing one event after another. Checking for deadlocks or invariant violations also works well in both versions. The standalone client shows how many states are already explored and how many are still unexplored. On the contrary, the Rodin plugin gives no feedback while running. The other big difference lies in the supported LTL model checking functionality. On the one hand the Rodin plugin can create a graph that visualizes a liveness counterexample. On the other hand the standalone client supports LTL formulas with event parameters. None of the two supports both functionalities.

Software	**Version**
Rodin	3.4.0
Atelier B provers	2.2.1
Camille TextEditor	3.3.0
ProB for Rodin3	3.0.10
SMT Solvers	1.4.0
ProB 2.0 UI	1.0.1
ProB 2.0 kernel	4.0.0

Table 3.1.: Software Versions

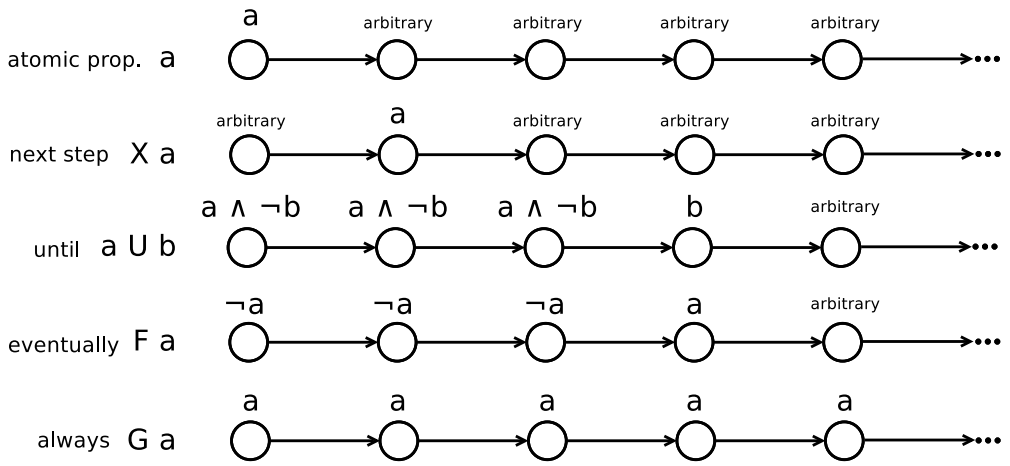

Figure 3.2.: Intuitive semantics of temporal modalities.
Adapted from Principles of Model Checking by Baier and Katoen [17].

3.6.1. Temporal Logic

When analyzing computer programs, we often require them to terminate and to return the correct result when doing so. However, many systems do not fit this simple structure. A web server for example, runs for a very long time and does not produce a meaningfull output at the end. We call these reactive systems. They are assumed to run forever and accept input and produce outputs while doing so. To reason about a reactive system, we need a logic that can talk about how a system behaves over time. Such a logic is called a temporal logic.

A widely used temporal logic, introduced by Pnueli [82], is linear temporal logic (LTL). It can be used to describe a relative order of states in an infinite trace. LTL uses temporal operators to describe when some state shall occur. It can express things like this property shall always be true, or that property shall be true in some future state. The grammar of LTL , as defined by Baier and Katoen [17], is shown in Equation (3.1), where φ is an LTL formula and a is an atomic proposition about a state of the system.

$$\varphi ::= \mathit{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathbf{X}\,\varphi \mid \varphi_1\,\mathbf{U}\,\varphi_2 \tag{3.1}$$

The temporal operators, used in this work, are *next* (**X**), *until* (**U**), *eventually* (**F**), and *always* (**G**). *Next* means that something holds in the next state of the execution. *Until* means that a proposition holds, until at some point in time another proposition becomes true. The derived operator *eventually* is used to say that a proposition holds in some future state. The second derived operator *always* means that something holds in all time steps. *Eventually* and *always* can be defined using the existing operators: $\mathbf{F}\,\varphi \stackrel{\text{def}}{=} \mathit{true}\,\mathbf{U}\,\varphi$ and $\mathbf{G}\,\varphi \stackrel{\text{def}}{=} \neg\mathbf{F}\,\neg\varphi$. Figure 3.2 contains a diagram that visualizes the different operators.

For verification purposes temporal properties are divided into two groups: safety and liveness properties.

Safety

A safety property states that something bad shall never happen. They often are of the form $\mathbf{G}(\neg bad)$. These are invariants of the system. In this work we will not use the model checker to verify invariants because they are formally proven.

Liveness

The other type of property are liveness properties. They state that something good will happen in the future. For example, a server shall send an answer in some future state. The simplest liveness property is $\mathbf{F}\ good$. A common liveness property is that we want our system to respond to all inputs it receives.

ProB supports an extension to LTL that not only allows us to write formulas about states, but also about events. The notation $[event]$ means an event will be executed in the next step and $e(event)$ means that an event is enabled in the current state. Using this notation, we can express the property that every request shall receive a response as $\mathbf{G}([request] \Rightarrow \mathbf{F}[response])$.

Most properties we use in this thesis are of a similar form. An actor system receives some message and it must send an answer back. Formally proving liveness properties is more involved than proving invariant preservation. Event-B proof obligations for certain classes of liveness properties have been developed by Yadav and Butler [95] and Hoang and Abrial [56]. Instead of performing formal proofs, we use ProB and manual testing to check these properties.

Fairness

Suppose we have a system that correctly implements the logic to reply to some message. However, it still violates the liveness property stating that every message receives an answer. The system could accept a new message in every step, but it never executes the code that replies to them.

Many liveness properties only hold if the system behaves in a fair way. An event that is enabled shall not be stalled forever, but instead must be executed at some point.

There are two different formulations of fairness, we use the same definitions as ProB [49]:

Weak fairness:

An event that is eventually always enabled will be executed infinitely often.
$\mathbf{FG}\ e(event) \Rightarrow \mathbf{GF}\ [event]$

Strong fairness:

An event that is enabled infinitely often will be executed infinitely often.
$\mathbf{GF}\ e(event) \Rightarrow \mathbf{GF}\ [event]$

The difference between the two is that under strong fairness an event needs to be executed even if it is not enabled in some states. As long as it is enabled in infinitely many states. For example, an event that is only enabled in every second time step.

In an actor system the two kinds of fairness are often equivalent. If the event handles a message from its mailbox, it is enabled as long as there is a message in there. The only way to remove the message from the mailbox is to execute the event. This cannot be true if the event guard also depends on the state of the actor that can be changed by other messages. Another situation where weak and strong fairness behave differently is if the guard depends on some global state.

ProB contains algorithms to handle fairness more efficiently than encoding them in LTL [49]. They also provide special syntax to specify fairness constraints. To state that an event is weakly fair, we can write $WF(event)$ and $SF(event)$ if it should be strongly fair. Additionally there is WEF and SEF to express that all events shall be weakly or strongly fair. This fairness constraints are written with an implication in front of the actual LTL formula. For example, $SEF \Rightarrow \varphi$ means the formula φ holds if all events are strongly fair.

4. Modeling Techniques

4.1. Representations for Actors and Messages

To model actor systems in Event-B [2], it is necessary to define when a model represents an actor system. This requires us to assign Event-B constructs to all components of an actor system, we want to study. The two most important of these components are actors and messages. A useful model resembles code in a programming language with support for actors, even if the syntax is quite different. Event-B models are also more abstract, meaning that we avoid defining details that are not essential to the task at hand. To demonstrate the similarities between our models and an actual program, we use Scala [80] code using the Akka library [70], described in Chapter 2.

4.1.1. Oneplace Mailboxes

In this section we explore different representations using a small client server example. The actor system consists of one server and one client. A message can be sent by the client to the server which will reply a message with the same content to the client. The Scala code for this example is shown in Listing 4.1. There are two types of messages `Send` and `Reply`, both contain a parameter `value:Int` as the actual message content. In the case of `Send` there is also a parameter `replyTo:ActorRef[Reply]` that contains the address of the client. Whenever the server receives a message, it sends a new message with the same `value` to this address. The client simply logs the messages it receives. Figure 4.1 shows an execution of a single message by the client and the reply by the server as a UML sequence diagram [85, 51].

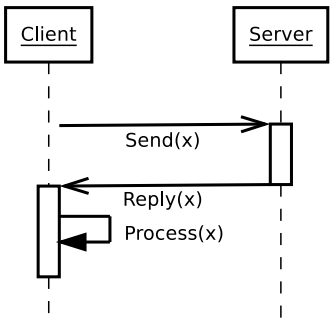

Figure 4.1.: Simulation of a simple client server system

```
final case class Send(value: Int, replyTo: ActorRef[Reply])
final case class Reply(value: Int)

val server: Behavior[Send] = Behaviors.receive { (context, message) =>
  context.log.info("Server received: {}", message.value)
  message.replyTo ! Reply(message.value)
  Behaviors.same
}

val client: Behavior[Reply] = Behaviors.receive{ (context, message) =>
  context.log.info("Client received: {}", message.value)
  Behaviors.same
}
```

Listing 4.1: Simple client server system in Akka

At the most abstract level, the model uses a context to define the type of messages. Instead of using integer as in the Scala program, we define a carrier set `MSG_CONTENT`. When modeling an actor system, we are often more interested in the protocol than in the computations that are performed on the transmitted data, or how the memory layout of a message looks like. It is however important to track messages in the network. In our example, we want the server to reply the same content, it got from the client and not just anything. Carrier sets can be infinite and the only operation supported by their members is equality testing. This means, we can check if a message that has been sent at one point is the same as the message that is received later on. The proof search is also simpler, as the prover only must try equality constraints and functions to prove its goal. It is not necessary to employ an arithmetic solver. For small models, without complicated computations, modern SMT solvers [45, 22] are so fast that this will rarely matter. When the model requires some computation with the message content, as in the case study in Chapter 5, we can change the message type to a more concrete one.

Now that we have defined the message type, we need a representation for actors. Different representations rely on different assumptions about what is possible in the modeled system. Simpler representations need more and stronger assumptions, than more complex ones. All actors consist of their mailbox, events describing their behavior and optionally an internal state. The modeling techniques presented here, differ in the way these parts are implemented.

In the first model each mailbox is a simple state variable of the message type and there is no internal state. The carrier set `MSG_CONTENT` contains a special value `empty_msg`, it signals that there is currently no message in a mailbox. There are two state variables `client_mail` and `server_mail`, both are initialized to `empty_msg`. The behaviors are described by the events in Listing 4.2. There are two events associated with the client `ClientSend` and `ClientReceive`, both take a single parameter of type `MSG_CONTENT`. In `ClientSend` this parameter is an input, meaning that the client chooses a message content, to send. The parameter in `ClientReceive` is different, it is an output and its value is completely defined

Initialisation
begin
act0: $client\_mail := empty\_msg$
act1: $server\_mail := empty\_msg$
end
ClientSend ⟨ordinary⟩ $\widehat{=}$
any
content
where
grd0: $content \in MSG\_CONTENT$
grd1: $content \neq empty\_msg$
grd2: $server\_mail = empty\_msg$
then
act0: $server\_mail := content$
end

ServerReply ⟨ordinary⟩ $\widehat{=}$
when
grd0: $server\_mail \neq empty\_msg$
grd1: $client\_mail = empty\_msg$
then
act0: $server\_mail := empty\_msg$
act1: $client\_mail := server\_mail$
end
ClientReceive ⟨ordinary⟩ $\widehat{=}$
any
content
where
grd0: $client\_mail \neq empty\_msg$
grd1: $content = client\_mail$
then
act0: $client\_mail := empty\_msg$
end

Listing 4.2: Actor model with oneplace mailboxes

by the state variables. This signifies that the client accesses this value and processes it in a way that is outside the scope of the model. A concrete implementation could print the value to the screen.

To send a message to the server, the client directly accesses the server's mailbox and writes its message into it, this is done with action `act0` of the `ClientSend` event. This approach poses an important question: what happens if there is already a message in there? There are three ways how this can be resolved. The most naive solution would be to override the previous message, replacing it with the new one. This is almost never the expected behavior. Slightly better is the approach used in the sample code above. A Guard is used to ensure that an mailbox is empty before writing to it. After a message is read, the mailbox is reset to empty. A variation of this, is to use an additional Boolean state variable for each mailbox that tells if the mailbox is empty or not. Using these additional variables, allows us to avoid extending the message type with a dummy value. Dummy values for empty mailboxes are bad when a later refinement uses a different mailbox representation. The dummy value is no longer needed, but still clutters the data type. The third approach is to use a unbounded mailbox, as will be discussed later on.

The approach described above might seem naive, but it still can be useful if certain assumptions hold.

- The number of actors must be finite and known when writing the model.
- No actors can be created at runtime.
- It is impossible to send an actor id from one actor to another. The layout of the actor system cannot be changed.
- An actor must always process a message before it can receive a new one.
- There is only one message type that can hold a single value.

The first three of these exist, whenever a model is used, where each actor has its own

mailbox variable. The forth constraint is specific to oneplace mailboxes. They can still be used, when a single execution of a protocol is described. In this case there will often be actors that only receive a single message while the program is running. However, the problem with this model is that an actor is blocked from sending, which is contrary to the asynchronous communication pattern.

4.1.2. Unbounded Mailboxes

A major feature of actor systems is their way of asynchronous communication. This is severely hindered if we require that each mailbox can contain at most one message. Imagine a system with three actors, two of them are clients and one is a server. The clients send requests to a server which processes them and sends back an answer. If we use a model, as described in Section 4.1.1, the server artificially synchronizes both clients. One cannot send a new request while the request of the other one is still unprocessed. In a concrete implementation the clients would not have access to the state of the server mailbox. They could not determine if the mailbox is empty or not, thus one message would inevitably get lost when both try to send at the same time. In order to prevent this, actors have a mailbox that can hold multiple messages. This section explores three different ways to model such mailboxes.

Sets

The first model uses a set of messages. Sets are directly supported by the logic of Event-B, thus we do not need to find an encoding. Adapting the model is relatively straight forward. The state variables `client_mail` and `sever_mail` are of type set of `MSG_CONTENT`. This is written as the variable is an element of the powerset of `MSG_CONTENT`. The guards and actions use the elementary set operations *element of*, *union*, and *difference*. The resulting model is given in Listing 4.3. We skipped the variable declarations, as the used variables are also enumerated in the invariants and the initialization event.

A new parameter was introduced to the event `ServerReply`, it models the server nondeterministically choosing one of the available messages. Event-B also supports a nondeterministic assignment operator. In this case we cannot use it because the two actions of the server are linked; they must always use the same value for `content`. We want the server to remove the same message from its mailbox that it sends to the client. When using nondeterministic assignment, both actions would choose their message independently. Another reason to use a parameter is to give a name to this object. Event-B sadly does not support let expressions or synonyms, so the only way to assign a name to something is to make it either an event parameter or a state variable.

Another difference to the previous model is that the event `ClientSend` is always enabled, its only guard merely specifies the type of the parameter `content`. To send the message, the client directly accesses the mailbox of the server. The new server mailbox is the old with the message added. Event-B does not support messages, everything is based on manipulating

INVARIANTS
inv0: $client\_mail \in \mathbb{P}(MSG\_CONTENT)$
inv1: $server\_mail \in \mathbb{P}(MSG\_CONTENT)$
Initialisation
begin
act0: $client\_mail := \varnothing$
act1: $server\_mail := \varnothing$
end
ClientSend ⟨ordinary⟩ $\widehat{=}$
any
content
where
grd0: $content \in MSG\_CONTENT$
then
act0: $server\_mail := server\_mail \cup \{content\}$
end

ClientReceive ⟨ordinary⟩ $\widehat{=}$
any
content
where
grd0: $content \in client\_mail$
then
act0: $client\_mail := client\_mail \setminus \{content\}$
end
ServerReply ⟨ordinary⟩ $\widehat{=}$
any
content
where
grd0: $content \in server\_mail$
then
act0: $server\_mail := server\_mail \setminus \{content\}$
act1: $client\_mail := client\_mail \cup \{content\}$
end

Listing 4.3: Actor model with sets as mailboxes

a global state. There are extensions that allow the usages of multiple models that are linked via shared state [1, 5] or shared events [29]. None of them support communication via actor-like messages. We therefore stick to using state variables as mailboxes in this model and all later ones. For a model to qualify as an actor system, it shall adhere to the following style guide.

- Event names start with the name of the actor or actor behavior that they are part of followed by a name that describes what they are doing.
- An event can access a single message out of its actors mailbox. When doing so, this message must be removed, from the mailbox, by the actions of this event.
- An event may never read two messages or leave a message in the mailbox after reading it.
- Events that do not read a message are only allowed if they either represent an outside influence or an observation about the system. The event `ClientSend` is a case of an outside influence, something outside the modeled system triggers the client to send a new message. All other events are triggered by receiving a message.

While processing a message, an event may write one or multiple messages to mailboxes of actors, its actor knows. For the current model the relationship of knowing an actor is hard coded into the model. The client knows the server and vice versa. Section 4.1.4 goes into more detail about what knowing another actor means in systems with many actors. An event may never remove messages from other mailboxes or alter messages in other mailboxes.

Using sets makes this model asynchronous. The server can receive arbitrary many messages, before it responds to one of them. In the previous model on the other hand the server is always required to reply before a new message could be sent. Processing of messages is not bound to the order in which they were sent. The guards only require it to be a message from the mailbox which is unordered.

However, there is one big limitation in modeling mailboxes as sets. It assumes that each message is only sent ones. More precisely, a message may not be sent to an actor whose mailbox currently contains the same message. This might be sensible if each sender is required to include a unique identifier in the message by the protocol and it will not send a new message before it received an answer to the previous one. Most often though this cannot be guaranteed. To overcome this problem, we need a way to allow multiple messages with the same content in the same mailbox.

Arrays

One approach is to model mailboxes not as sets, but as dynamic arrays. In logic, arrays are often modeled as functions from the natural numbers to the content of the array, due to McCarthy's array theory [71]. The mailboxes are changed to be of type $\mathbb{N} \pfun MSG\_CONTENT$, a function from the naturals to the content of the message. Each mailbox is extended by a new state variable with the suffix `_next_id` instead of `_mail`. This is a counter that stores how many messages have been sent to this actor and is used to supply a fresh and unused identifier for each message. We will sometimes refer to natural numbered identifiers as indices, as they represent an index into the mailbox array. The functions are partial, non injective, and non surjective. They are non injective because that is the reason we introduced them. We want them to potentially hold multiple messages with the same content. There are also more potential messages than there are in a mailbox, so the functions cannot be surjective. Finally, the function must be partial because there are infinitely many natural numbers, but each mailbox only stores a finite number of messages. Most indices do not point to a message. When more information, about the domain and range, is available, it is often a good idea to encode this in the type of the mailbox function. For example, if we know a set of all message contents that are currently sent but not yet read, we can change the range to this set and mark the function as surjective. This can be necessary to discharge some proof obligations.

The model using array functions is shown in Listing 4.4.

Using functions instead of sets, allows us to store multiple messages of the same value in the same mailbox, as long as an index is not repeated. This is guaranteed, by always incrementing the `_next_id` variables when a message is sent. The domain of the mailbox functions will gradually shift towards higher numbers. Old messages with lower indices, or identifier, get processed and removed and new messages with higher indices get added.

To choose a message from a mailbox, we add a message id parameter and a guard stating that it must be in the domain of the mailbox function. When we also need access the value of the message, we can add it as a parameter and use a guard expressing that the pair of index and value is an element of the mailbox function. This is done in the `ClientReceive` event. Sending a message is done by using the function override syntax, it can be seen in `act0` of the `ClientSend` event. The value of the function variable is updated with a new value that is the same as the old in all points, but the one supplied on the left side of the assignment. This value now points to the value on the right side of the assignment. If the

INVARIANTS
inv0: $client\_mail \in \mathbb{N} \pfun MSG\_CONTENT$
inv1: $client\_next\_id \in \mathbb{N}$
inv2: $server\_mail \in \mathbb{N} \pfun MSG\_CONTENT$
inv3: $server\_next\_id \in \mathbb{N}$

Initialisation
begin
act0: $client\_mail := \varnothing$
act1: $client\_next\_id := 0$
act2: $server\_mail := \varnothing$
act3: $server\_next\_id := 0$
end

ClientSend ⟨ordinary⟩ $\widehat{=}$
any
content
where
grd0: $content \in MSG\_CONTENT$
then
act0: $server\_mail(server\_next\_id) := content$
act1: $server\_next\_id := server\_next\_id + 1$
end

ServerReply ⟨ordinary⟩ $\widehat{=}$
any
msg_id
where
grd0: $msg\_id \in dom(server\_mail)$
grd1: $\forall i \cdot i \in dom(server\_mail) \Rightarrow msg\_id \leq i$
then
act0: $server\_mail := \{msg\_id\} \ndres server\_mail$
act1: $client\_mail(client\_next\_id) := server\_mail(msg\_id)$
act2: $client\_next\_id := client\_next\_id + 1$
end

ClientReceive ⟨ordinary⟩ $\widehat{=}$
any
content, msg_id
where
grd0: $msg\_id \mapsto content \in client\_mail$
then
act0: $client\_mail := \{msg\_id\} \ndres client\_mail$
end

Listing 4.4: Actor model with array mailboxes

value was not previously in the domain of the function, it now is. In case, the value is already in the domain, its image will be overwritten, but this can not happen in our model because we only write to unused indices. To remove a message from the mailbox, we use the domain reduction operator as in `act0` of `ServerReply` or `ClientReceive`.

For some applications, there is the assumption that messages will be received in the same order as they are sent. Mailboxes are first in first out queues in this case. To model this behavior, we add a guard to an event that states that the message index must be the smallest one currently in the mailbox. This is via the guard `grd1` in the `ServerReply` event. Messages sent to the server are answered in the same order as they were sent. The client on the other hand receives its messages in a nondeterministic order, independent of the time they were sent.

A disadvantage of using indices that are always increasing is that it is nearly impossible to exhaustively model check [39, 41, 17] such models. They are essentially infinite state machines that will never reach a previous state again. Model checkers rely on clever exploration techniques to exhaustively check a finite state machine. Without specific knowledge about the modeling style used in one of these models, a model checker can only try random paths. This might discover errors, but the absence of errors cannot be shown in this way. A specialized checker could be built that employs specific knowledge about our model. We can observe that there are states where all mailboxes hold the same message content, but the indices are different. A special symmetry reduction [40, 72] or state abstraction could be used that marks states as equal if they only differ in the values of the indices. This would allow for exhaustively checking small models.

Abstract identifiers

In cases where we want to use model checking and do not rely on the order of messages we can replace the natural numbered indices by identifiers from a carrier set. We name this set `MSG_ID`, it is an infinite set because we always want to be able to send messages no matter how many are already in some mailboxes. This would still be impossible to model check, but the model checker can use a finite subset and exhaustively check this. Not all possible violations will be found this way, but we can at least say, this property holds when there are at most n messages in mailboxes at the same time.

The model with identifiers of type `MSG_ID` is available in Listing 4.5. It is no longer necessary to keep a counter to supply fresh identifiers. This is now done in the events. Each event that wants to send a message has a parameter `fresh_id` and a guard stating that this identifier is not in the domain of the target mailbox. An actor that sends two or more messages in a single event needs one parameter and one guard for each message. The model is the same as the one in Listing 4.4 in all parts that are not related to the message identifiers.

INVARIANTS
inv0: $client\_mail \in MSG\_ID \pfun MSG\_CONTENT$
inv1: $server\_mail \in MSG\_ID \pfun MSG\_CONTENT$

Initialisation
begin
act0: $client\_mail := \varnothing$
act1: $server\_mail := \varnothing$
end

ClientSend ⟨ordinary⟩ $\widehat{=}$
any
content, fresh_id
where
grd0: $content \in MSG\_CONTENT$
grd1: $fresh\_id \notin dom(server\_mail)$
then
act0: $server\_mail(fresh\_id) := content$
end

ServerReply ⟨ordinary⟩ $\widehat{=}$
any
msg_id, content, fresh_id
where
grd0: $msg\_id \mapsto content \in server\_mail$
grd1: $fresh\_id \notin dom(client\_mail)$
then
act0: $server\_mail := \{msg\_id\} \domsub server\_mail$
act1: $client\_mail(fresh\_id) := content$
end

ClientReceive ⟨ordinary⟩ $\widehat{=}$
any
content, msg_id
where
grd0: $msg\_id \mapsto content \in client\_mail$
then
act0: $client\_mail := \{msg\_id\} \domsub client\_mail$
end

Listing 4.5: Actor model with abstract identifier mailboxes

Using unbounded mailboxes, allows us to soften the assumptions we rely on. The remaining assumptions are:

- The number of actors must be finite and known when writing the model.
- No actors can be created at runtime.
- It is impossible to send an actor id from one actor to another. The layout of the actor system cannot be changed.
- There is only one message type that can hold a single value.
- The order of message processing can be either first in - first out or nondeterministic.

4.1.3. Algebraic Message Types

In this section we discuss how messages with multiple fields can be modeled. To better demonstrate this, we change the server in our Scala example as shown in Listing 4.6. The `Send` message now contains two integer parameters and the server replies with the smaller of these two numbers. This will also help us to demonstrate how branching inside an actor can be modeled. Before the server can reply, it needs to compare the two numbers and then reply with one or the other. All of this is done in one atomic action.

```
1  final case class Send(n1: Int, n2: Int, replyTo: ActorRef[Reply])
2
3  val server: Behavior[Send] = Behaviors.receive { (context, message) =>
4    context.log.info("Server received: {}", message.value)
5    if (message.n1 < message.n2) {
6      message.replyTo ! Reply(message.n1)
7    } else {
8      message.replyTo ! Reply(message.n2)
9    }
10   Behaviors.same
11 }
```

Listing 4.6: Simple server calculating the smaller function in Akka

A message that contains multiple fields is a product type. Event-B has a product operator that builds the Cartesian product of two sets. We can use it to create an ordered pair of two natural numbers. This is used for the `server_mail` variable in Listing 4.7. Another possibility would be to adopt Event-B's theory plugin [32] which supports algebraic data types. With it we can create product types where the fields are not identified by position, but by a name. Algebraic data types are an essential part of strongly typed functional programming languages such as SML [75] or Haskell [81].

Sending and receiving messages with multiple fields is the same as with messages consisting of only one value. We simply replace the single value with the pair of two values.

There is no if-then-else in Even-B, as it does not support control structures in expressions. All control flow is handled via event guards or nondeterministic assignments. In Listing 4.7 the server is split into two events: `ServerReply1` and `ServerReply2`. The decision which number is sent back to the client is done by the guards `grd2`. Both events only differ in this guard and the resulting action `act1`. To access the individual fields of the message, the build in functions `prj1` and `prj2` are used. They calculate the first or second projection of a product type, meaning the first or second element of the pair.

Using product types for messages, has a detrimental effect on automatic theorem provers and complicates manual proofs. The problem is that Event-B generates proof obligations for each state variable. This results in one complicated proof obligation for the whole message, even if we only access one field of the message, the whole structure is brought into scope. A common recommendation for modeling languages that rely heavily on SMT solvers,

INVARIANTS

inv0: $client\_mail \in MSG\_ID \nrightarrow \mathbb{N}$
inv1: $server\_mail \in MSG\_ID \nrightarrow (\mathbb{N} \times \mathbb{N})$

EVENTS
Initialisation

begin
act0: $client\_mail := \varnothing$
act1: $server\_mail := \varnothing$
end

ClientSend ⟨ordinary⟩ $\widehat{=}$

any
n1, n2, fresh_id
where
grd0: $n1 \mapsto n2 \in \mathbb{N} \times \mathbb{N}$
grd1: $fresh\_id \notin dom(server\_mail)$
then
act0: $server\_mail(fresh\_id) := (n1 \mapsto n2)$
end

ClientReceive ⟨ordinary⟩ $\widehat{=}$

any
content, msg_id
where
grd0: $msg\_id \mapsto content \in client\_mail$
then
act0: $client\_mail := \{msg\_id\} \lhd client\_mail$
end

ServerReply1 ⟨ordinary⟩ $\widehat{=}$

any
msg_id, content, fresh_id
where
grd0: $msg\_id \mapsto content \in server\_mail$
grd1: $fresh\_id \notin dom(client\_mail)$
grd2: $prj1(content) < prj2(content)$
then
act0: $server\_mail := \{msg\_id\} \lhd server\_mail$
act1: $client\_mail(fresh\_id) := prj1(content)$
end

ServerReply2 ⟨ordinary⟩ $\widehat{=}$

any
msg_id, content, fresh_id
where
grd0: $msg\_id \mapsto content \in server\_mail$
grd1: $fresh\_id \notin dom(client\_mail)$
grd2: $prj1(content) \geq prj2(content)$
then
act0: $server\_mail := \{msg\_id\} \lhd server\_mail$
act1: $client\_mail(fresh\_id) := prj2(content)$
end

Listing 4.7: Actor system calculating the smaller function.

INVARIANTS

inv0: $client\_mail \in MSG\_ID \nrightarrow \mathbb{N}$
inv1: $server\_mail\_n1 \in MSG\_ID \nrightarrow \mathbb{N}$
inv2: $server\_mail\_n2 \in MSG\_ID \nrightarrow \mathbb{N}$
inv3: $dom(server\_mail\_n1) = dom(server\_mail\_n2)$

Listing 4.8: Type declarations for messages consisting of multiple fields

such as Alloy [59], is to keep the model as flat as possible [60]. Following this advice, we split the state variables for the server in two; this is shown in Listing 4.8. Each field is represented by its own state variable. We employ a naming convention that messages with multiple fields have a suffix with the field name. An additional invariant requires that the domain of both mailboxes is the same. This means that always both fields must be sent. One field can be marked as optional by changing the equality between the domains to a subset equal relation. The events for this representation are straight forward and therefore omitted from Listing 4.8. The difference from Listing 4.7 is that instead of the projections the individual variables are used and that sending and receiving the message requires two actions, one for each field.

Sometimes one actor is supposed to handle not only one type of message, but multiple. Listing 4.9 shows the type declarations for a model that combines the one from Section 4.1.2

INVARIANTS

inv0: $client\_mail\_smaller \in MSG\_ID \nrightarrow \mathbb{N}$
inv1: $client\_mail\_ping \in MSG\_ID \nrightarrow MSG\_CONTENT$
inv2: $server\_mail\_smaller\_n1 \in MSG\_ID \nrightarrow \mathbb{N}$
inv3: $server\_mail\_smaller\_n2 \in MSG\_ID \nrightarrow \mathbb{N}$
inv4: $dom(server\_mail\_smaller\_n1) = dom(server\_mail\_smaller\_n2)$
inv5: $server\_mail\_ping \in MSG\_ID \nrightarrow MSG\_CONTENT$

Listing 4.9: Type declarations for a system with multiple different messages

and from this section. The client can either send a ping or two numbers to the server. In case of a ping, the server replies with the same content as it received. When the server receives two numbers, it returns the smaller one. These messages could be modeled in the theory plugin [32] as a sum type of product types. We will not follow this route. Instead, we create individual mailboxes for each message type.

The naming convention for messages with multiple message types is as follows: first comes the name of the actor, followed by the name of the message and at last the name of the message fields. If a message consists of only one field, the field name can be omitted. Also if an actor only accepts one type of message, the message name can be omitted.

4.1.4. Dynamic Topology

What distinguishes actor systems from many other concurrency models is that new actors can be created and that actor identifiers can be exchanged between actors. This allows the topology of the network to change based on messages.

In order to illustrate these properties, we again modify our Scala example. Instead of replying to the client that sent the request, each request now contains a list of recipients. Note that the previous Scala example is actually dynamic as well, the `replyTo` field could be filled with a reference to any client, not just the one which sent the messages. This was just ignored in our previous models. The Scala code of the new server is shown in Listing 4.10. The server iterates over all `ActorRef`s in `replyTo` and sends a reply to each of the entries. When creating the actor system, it is not yet known to whom the server will send its messages. There are no established channels from the server to the clients. Instead, the server decides, based on the message content, who shall receive a reply; it also learns the identifiers or `ActorRef`s through this message.

In Figure 4.2 an execution of this system with three clients is shown. The clients are numbered from one to three. Initially the first client sends a message to the server and names as recipients itself and client two. Client three will not receive a reply. On receiving the message, the server sends the message content to all clients listed as recipients, namely one and two. Who should receive these messages was only defined by the previous message and was not known when the system was created. At that point, the server did not even know that the clients exist.

```
final case class Send(value: Int, replyTo: List[ActorRef[Reply]])

val server: Behavior[Send] = Behaviors.receive { (context, message) =>
  context.log.info("Server received: {}", message.value)
  message.replyTo.foreach(_ ! Reply(message.value))
  Behaviors.same
}
```

Listing 4.10: Simple server with replyTo list in Akka

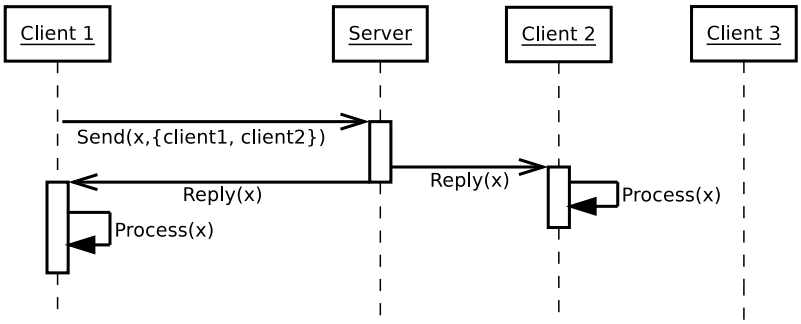

Figure 4.2.: Simulation of a simple client server system with three clients

Our current modeling technique does not allow us to pass references to actors in messages. Each mailbox is represented by its own state variable and Event-B does not contain pointers. So we need to build this functionality into the model ourselves. We introduce another carrier set named `CLIENT_ID`. It contains abstract identifiers for clients. Instead of one state variable per actor, we now have one state variable per actor type, `client` in this case. This mailbox is a function of type $(CLIENT\_ID \times MSG\_ID) \pfun MSG\_CONTENT$. It maps an actor identifier and a message identifier to the message with the given identifier in the mailbox of the given actor. One might ask why we do not use a curried function of type $CLIENT\_ID \pfun (MSG\_ID \pfun MSG\_CONTENT)$ instead. This notation closer resembles our understanding of an actor system. Each client is associated with one mailbox and this mailbox contains one message for each message identifier. At a first glance the two representations look equivalent, seeming to be sets of ordered triples. Though the two versions are not equivalent because the functions are partial. In case of the curried one, a client identifier could be absent, or it could point to an empty set. The uncurried version cannot distinguish these two cases. This difference leads to different syntactic constructs being available for each of them. Using the curried version makes it easy to get the whole mailbox, for one client. Unfortunately, everything else becomes nearly impossible. We cannot use function assignment to send messages; instead, we must explicitly use relational overwriting. It is not possible to use the syntax $client\_id \mapsto msg\_id \mapsto content \in mailbox$ to access a message, we would need to explicitly name the mailbox and write one predicate that the mailbox is the one of the client and another predicate that the message identifier and content are part of this mailbox. The uncurried version allows both of these operations, greatly simplifying the guards and actions. At the minor cost that the type signature looks

INVARIANTS

inv0: $client\_mail\_ping \in (CLIENT\_ID \times MSG\_ID) \pfun MSG\_CONTENT$
inv1: $server\_mail\_ping\_content \in MSG\_ID \pfun MSG\_CONTENT$
inv2: $server\_mail\_ping\_receivers \in MSG\_ID \pfun \mathbb{P}(CLIENT\_ID)$
inv3: $dom(server\_mail\_ping\_content) = dom(server\_mail\_ping\_receivers)$

ClientSend ⟨ordinary⟩ $\widehat{=}$
any
client_id, content, receivers, fresh_id
where
grd0: $client\_id \in CLIENT\_ID$
grd1: $content \in MSG\_CONTENT$
grd2: $receivers \in \mathbb{P}(CLIENT\_ID)$
grd3: $fresh\_id \notin dom(server\_mail\_ping\_content)$
then
act0: $server\_mail\_ping\_content(fresh\_id) := content$
act1: $server\_mail\_ping\_receivers(fresh\_id) := receivers$
end

ClientReceive ⟨ordinary⟩ $\widehat{=}$
any
client_id, content, msg_id
where
grd0: $client\_id \mapsto msg\_id \mapsto content \in client\_mail\_ping$
then
act0: $client\_mail\_ping := \{client\_id \mapsto msg\_id\} \ndres client\_mail\_ping$
end

ServerReply ⟨ordinary⟩ $\widehat{=}$
any
msg_id, content, receivers, fresh_id
where
grd0: $msg\_id \mapsto content \in server\_mail\_ping\_content$
grd1: $msg\_id \mapsto receivers \in server\_mail\_ping\_receivers$
grd2: $fresh\_id \notin \{m\_id, r \cdot r \in receivers \land r \mapsto m\_id \in dom(client\_mail\_ping) | m\_id\}$
then
act0: $server\_mail\_ping\_content := \{msg\_id\} \ndres server\_mail\_ping\_content$
act1: $server\_mail\_ping\_receivers := \{msg\_id\} \ndres server\_mail\_ping\_receivers$
act2: $client\_mail\_ping := client\_mail\_ping \nrres \{r \cdot r \in receivers | r \mapsto fresh\_id \mapsto content\}$
end

Listing 4.11: Actor system with dynamic topology

a bit less intuitive.

Listing 4.11 displays the model for the dynamic topology example. The client mailbox is now parameterized with a client identifier. There is still only one server and no need to add a server identifier. The list of receivers is changed to a set because that is easier to model. We assume that no client is supposed to receive the message twice, as would be possible with a list.

The `ClientSend` event is nearly unchanged. It only contains the additional field receivers in the message to the server. The content for this field is a parameter chosen from all possible sets of client identifiers. In the event `ClientReceive`, we can see, how a message is read from a mailbox if there are multiple actors with the same behavior. Using the uncurried function representation, makes it possible to write a guard stating that the triple consisting of client identifier, message identifier and message content shall be an element of the mailbox function. To remove the message from the mailbox, the domain restriction operator is used. Instead of just the message identifier, we now supply a pair of

client identifier and message identifier. This removes this one message and leaves all other mailboxes and other messages in the same mailbox unchanged. To send a message to a single actor, we would use the same pair together with the function override syntax.

The most interesting part of this model is the event `ServerReply`. Recall that the server needs to send messages to all clients, listed in the message, it receives. There is no reason that message identifiers must be globally unique. It is sufficient if they are unique in each mailbox. So the first part of sending the messages is finding an identifier that does not correspond to an existing message in any of the mailboxes we want to write to. This is done by `ServerReply`'s the guard `grd2`. A set comprehension is used to construct a set of all identifiers that are currently used in one of the affected mailboxes. Then we can state that the `fresh_id` shall be not one of those.

Sending to all chosen clients is done by updating the client mailbox state variable with a relational override. A set with all sent messages is constructed, again using a set comprehension. It contains one triple for each receiver, where all triples share the same message identifier and message content. This set is used to update the client mailboxes. Instead of a relational override, a set union could be used as well. From the guards it is clear that there will be no overwriting because we have chosen a message identifier accordingly. So union and overwriting are equivalent in this case.

The introduction of composite messages in the previous section and actor identifiers in this section, lets us represent most actor systems. Of the assumptions stated in Section 4.1.1 only one remains: Using unbounded mailboxes allows us to soften the assumptions, we rely on. The remaining assumptions are:

- No actors can be created at runtime.

We are now able to handle an infinite number of actors because our set of identifiers is infinite. It is also possible to send actor identifiers from one actor to another, as we demonstrated in Listing 4.11. A message can consist of multiple fields, this was introduced in Section 4.1.3.

4.1.5. Creating Actors with Internal State

The behavior of an actor can also depend on state variables. This means that an actor can store information it got at creation or received by a message. In this section we introduce two concepts at once: actors with state and creating actors. A real system will need to adapt while running, this often includes creating an actor for a specific purpose. It might also be necessary to update the state of an actor, for example to increment some counter.

We adapt our running example to include a proxy actor. When the server receives a message, it will not reply directly to the given address, instead it creates a new actor. This proxy forwards all messages it receives. The receiver of messages sent by the proxy is determined by a state variable. It holds the address of the receiving actor and is set when the proxy is created. After creating the proxy actor, the server immediately sends the reply message to

the proxy. The message is the same that would have been sent directly to the client. In this model it passes through proxy instead.

Listing 4.12 shows the Scala code of the server and the proxy. The message type definitions `Send` and `Reply` are unchanged from Listing 4.1. The proxy has one parameter named `client`, it holds a reference, of type `ActorRef[Reply]`, of the client it sends its messages to. When it receives a message, it forwards the exact same message to the client. No rewrapping is required because client and proxy accept the same type of messages. The server still receives a `Send` message containing the actual value and the `replyTo` address of the actor which the message shall be delivered to. Using the context object, the server creates a new actor with the `proxy` behavior that captures the `message.replyTo` value in a closure. In the next step, it sends the `Reply` message to the newly created actor `msg_proxy`.

```
1  val server: Behavior[Send] = Behaviors.receive { (context, message) =>
2    context.log.info("Server received: {}", message.value)
3    val msg_proxy = context.spawnAnonymous(proxy(message.replyTo))
4    msg_proxy ! Reply(message.value)
5    Behaviors.same
6  }
7
8  def proxy(client: ActorRef[Reply]): Behavior[Reply] =
9    Behaviors.receive{ (context, message) =>
10     context.log.info("Proxy received: {}", message.value)
11     client ! message
12     Behaviors.same
13   }
```

Listing 4.12: Server with proxy in Akka

A simulation of this system, with a single client actor and one message, is shown in Figure 4.3. Initially the client sends a message to the server containing a value x and its own address. Upon receiving this message, the server spawns a new actor named `Proxy`. This actor has a state that stores the address `Client 1`. At the same time the server sends the reply message, containing the value x, to the proxy actor it created. These two events happen sequentially in Akka, but only one update is done in the Event-B model. When the proxy receives a message, it forwards it to the address, it has stored in its state. Finally, the reply message is received and processed by the client.

Listing 4.13 shows the Event-B model of the example discussed previously. We will explain the modeling techniques required to handle spawning new actors and actor state based on this example.

To spawn actors of a certain type in our Event-B model, we first need a set of identifiers for this type of actors. In this case we use a new carrier set `PROXY_ID`. What is different to previous actors is that we cannot assume that there exists an actor, with a specific identifier from this set, when the system starts. There must be unused identifiers that can be used for the actors that will be created at runtime. We need to introduce a state

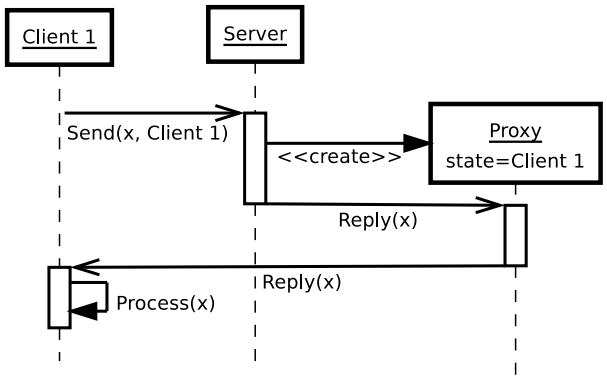

Figure 4.3.: Simulation of a simple client server system with a proxy

variable, to the Event-B model, that stores all identifiers that are currently in use. It will be a subset of `PROXY_ID`. Instead of using an explicit state variable for this purpose, we will combine it with the state of our actors.

When creating actors at runtime, actors of the same type often have a state that distinguishes them from each other. In our example, this is the `client` which shall receive the messages. State is stored by a state variable of function type, from the actor identifier to the type of the stored information, `CLIENT_ID` in this case. Each field of the actor state is modeled as its own function. We adhere to a similar naming convention as the one used for mailboxes. The state variables start with the name of the actor type, followed by the keyword `state` and finally by the name of the field, these three components are separated by underscores. There are two ways the type of this function can be written, either as a total function or as a partial function. When using a total function, we introduce a new state variable that holds the set of all active actors of this type and use it as the domain of the state function. Otherwise, we can write a partial function directly in terms of the carrier sets of identifiers and use the domain of the function to store the set of active actors. In Listing 4.13 the partial function approach is used. Both versions are based on the assumption that actors that do not exist cannot have a state. They also assume that all state variables are always defined for each existing actor. This restriction can be circumvented by using the same technique described in Section 4.1.3 for optional message fields. It is also necessary to ensure that only existing actors can have messages in their mailbox. This is enforced by the invariant `inv6`.

To create a new instance of an actor, as done in the `ServerReply` event, we need to correctly update the state variables for this actor type. By using a parameter, a fresh identifier for the new actor is chosen, it must be in the correct carrier set, but not already in use. We also need message identifiers for all messages, we want to send. In this case we can pick any because we know that the mailbox of the new actor will be empty, so there can be no conflict. The variable holding the actor state is updated by a function override action to include the new actor and correctly set its state. This creates the actor in our model. It is now eligible to receive messages. The next action sends the reply message to the newly created actor, this is done the same way as discussed in Section 4.1.2.

INVARIANTS

inv0: $client\_mail \in (CLIENT\_ID \times MSG\_ID) \nrightarrow MSG\_CONTENT$
inv1: $server\_mail\_content \in MSG\_ID \nrightarrow MSG\_CONTENT$
inv2: $server\_mail\_receiver \in MSG\_ID \nrightarrow CLIENT\_ID$
inv3: $dom(server\_mail\_content) = dom(server\_mail\_receiver)$
inv4: $proxy\_mail \in (PROXY\_ID \times MSG\_ID) \nrightarrow MSG\_CONTENT$
inv5: $proxy\_state\_client \in PROXY\_ID \nrightarrow CLIENT\_ID$
inv6: $\{p, i \cdot p \mapsto i \in dom(proxy\_mail) | p\} \subseteq dom(proxy\_state\_client)$

ServerReply ⟨ordinary⟩ $\widehat{=}$

any
msg_id, content, receiver, fresh_id, fresh_proxy_id
where
grd0: $msg\_id \mapsto content \in server\_mail\_content$
grd1: $msg\_id \mapsto receiver \in server\_mail\_receiver$
grd2: $fresh\_proxy\_id \notin dom(proxy\_state\_client)$
grd3: $fresh\_id \in MSG\_ID$
then
act0: $server\_mail\_content := \{msg\_id\} \lhd server\_mail\_content$
act1: $server\_mail\_receiver := \{msg\_id\} \lhd server\_mail\_receiver$
act2: $proxy\_state\_client(fresh\_proxy\_id) := receiver$
act3: $proxy\_mail(fresh\_proxy\_id \mapsto fresh\_id) := content$
end

ProxyForward ⟨ordinary⟩ $\widehat{=}$

any
proxy, msg_id, content, receiver, fresh_id
where
grd0: $proxy \mapsto msg\_id \mapsto content \in proxy\_mail$
grd1: $receiver = proxy\_state\_client(proxy)$
grd2: $fresh\_id \notin \{i \cdot receiver \mapsto i \in dom(client\_mail) | i\}$
then
act0: $proxy\_mail := \{proxy \mapsto msg\_id\} \lhd proxy\_mail$
act1: $client\_mail(receiver \mapsto fresh\_id) := content$
end

Listing 4.13: Actor system with actor creation

An actor state can be accessed by calling the corresponding function with the actor's identifiers. This is done in the `ProxyForward` event. A new parameter is introduced, holding the current state, to make it more convenient to send the reply message to the client.

When using state actor states and spawning new actors, certain conventions need to be obeyed. Similar to the rules for mailboxes, these are not checked by the proof system and need to be enforced by coding, or modeling, discipline. The following list summarizes all these rules, including the ones for mailboxes.

- No actor may access the state of another actor. The value, stored in the state function, may only be read or written to if the event is processing a message for the corresponding actor. It may also be written to, when the actor is created.
- No actor is allowed to modify the mailbox of another actor, except to send a message to it. A message may only be sent to an actor if the identifier of the receiver is known.

This means, it must be part of the currently processed message, or be part of the state of the current actor.
- Removing messages from the mailbox is only allowed by the actor processing the message. Only one message may be removed at a time. A message that is processed by an event, must be removed in the same event.
- Message and actor identifiers are chosen nondeterministically, but appropriate guards need to be used to ensure that there can be no name conflicts.

The techniques described in this section allow us to model all kinds of actor systems that are in scope for this thesis.

4.2. Refinement Strategies for Actor Systems

We propose two general refinement strategies to develop an actor system. The first one can be used to develop an actor system that can compute a mathematical function; starting from a recursive definition of this function. The second strategy is for building a reactive system. It starts with a minimal actor system and each refinement fulfills another requirement. These two strategies correspond to vertical and horizontal refinement [2].

4.2.1. Refinement Towards an Actor System

This section describes a vertical refinement technique to create an actor system model. The initial model contains a mathematical specification of an algorithm, expressed as a recursive function. From this we can derive an actor system in three major refinement steps. In a more complex model it might be necessary to implement each of these steps as multiple smaller steps.

While this strategy can be used to derive an actor system from a recursive function, one needs to consider if the performance characteristics are desired. On the one hand the memory overhead is linear in terms of the recursion depth and there is a constant overhead for handling the messages. On the other hand this architecture allows a long running computation to be interrupted by a shorter one. A function that can be implemented tail recursively could be implemented with constant memory usage as an imperative or functional program inside an actor. In this case on needs to decide if the possibility to interrupt a computation outweighs the large memory usage.

This strategy is indented as a theoretical exploration of how non-trivial actor systems can be derived from a recursive definition. It is generalization of the strategy used for the factorial in Chapter 5.

Mathematical Specification

The initial model contains a definition of the mathematical function. In Event-B this would be part of a context. There is a single event that uses this function to calculate the correct solution. This captures the indent of the system, given an input it will calculate the correct result. However, it has no resemblance to an actor system.

Iterative Solution

As a first refinement the recursive algorithm, that is calculated in a single step, needs to be broken up into multiple steps. Each recursive call needs to be its own event invocation. The resulting model is conceptually similar to an implementation in an imperative programming language. Executing an event corresponds to one execution of a loop body. The gluing invariant for this step is the same as one would need to prove the imperative program being correct.

At a later point, we want to compute everything by sending messages. The individual actors should not need to execute any loops or use other forms of iteration. This first transformation makes the iteration explicit in our model.

Explicit Stacks

In the next step, we introduce a stack. We no longer mutate the variables that are used in the iterative version of our algorithm. Some parameters of a recursive function are changed in a recursive call and will converge towards a base case. We use a model that is very similar to the execution of a recursive function in C (and other languages). When the function is called all parameters are pushed on the stack, and when the function returns they are popped again. The same strategy is used here. Each new value of a parameter, or a changing variable as we are refining a iterative model, is pushed onto the stack. Once the base case is reached, a different event pops one element after another from the stack and uses them to calculate the final result.

For the refinement it is important that the event building the stack is newly introduced, and the event reducing the stack, refines the previous compute event. The gluing invariants must specify how large the stack is at any point and what the values are at each position of the stack.

Emulating the Stack with Actors

The final step is to turn the model into an actual actor system. The role of the stack is now filled by dynamically created actors. Pushing a value onto the stack, gets replaced by spawning an actor. This actors state is the value from the stack frame and the address of the previously created actor. The actors, created in this way, form something similar to a linked list but with actor addresses instead of pointers. The actor that was created first,

and therefore has no predecessor, instead stores the address of a consumer who will receive the final result. When all actors are created, the last created one gets a message starting the actual computation. Each actor performs one step of the computation and sends the temporary result to the next actor in the chain.

The gluing invariants for this step need to link the actors and the stack. Each actor state corresponds to one entry in the stack. It also needs to be ensured that the linked list formed by the actors is in the same order as the stack entries.

4.2.2. Refinement Between Actor System Models

In this section we describe another strategy that uses actor system models from the beginning. Starting from a simple actor system that only describes the interaction in the most abstract sense, more details are added in each refinement step. That may require splitting a single actor into multiple, or introducing intermediate actors that forward messages. This strategy is mostly a horizontal refinement strategy and is used for the chat server in Chapter 6.

We explain a few techniques that can be used to extend the functionality of an actor system.

Extending at the Start or End of a Protocol

The easiest way to extend an actor system is to add new messages at the start or at the end. This is purely a horizontal refinement. Adding a new actor, to a system that receives a newly introduced message, does not require any refinement proofs. We can always add sending a new message to any event, as long as the receiver mailbox is introduced in the same refinement.

A message that was previously assumed to be coming from outside the system can be changed to come from a newly introduced actor. The message handling event of this actor needs to refine the event that previously modeled the external message.

Splitting Actors

Occasionally we want to model a group of actors as a single actor in an early version of our model. This is problematic if the split actor is in the middle of the protocol and needs to respond to messages. If the actor, to be split, only acts as a consumer for messages, we can split it.

The actor who sent the message now must sent a copy of the message to all new actors. We also introduce a new observation event that is executed if all new actors have performed the same tasks that the original actor performed. Most likely this means reading the message. This event refines the original message read event. Another event is added and used by

each new actor to read the message. The observation event can only be executed once every new actor has read its message.

Forwarding

It can be necessary to add a new actor in the middle of the protocol that acts as buffer for messages. However, this transformation only works if either the original receiver or sender is at the end of the protocol. We cannot add a new intermediary at any point of the protocol. There is one event that describes a message being removed from one mailbox and added to another one at the same time. To get a valid refinement of this event, we need to move one of these mailboxes from the original actor to the new intermediary. The original actor then gets a new mailbox. A gluing invariant is required that asserts that the intermediary mailbox is equivalent to the original actors mailbox in the previous machine.

This can be seen as a special case of extending at the start or at the end. A new actor is added and it receives a new message. The difference is how we name the actors. In this case we assume the new actor has the same identity as the original actor and the intermediary is actually a new actor.

These are some general strategies that we hope are useful for building a wide variety of systems. Each individual project might require variations of these techniques or completely new refinements.

4.3. Proof Engineering

In this section we explain how certain decisions while writing a model effect the difficulty of the required proofs. We also discuss how Rodin's interactive theorem prover helps with performing these.

4.3.1. Influences on Proof Complexity

To discuss if one proof is harder than another we need some definition of proof complexity. We will not give a formal definition, but an informal and somewhat subjective one. The general idea is that a proof is more complex if it requires more effort for a user who has access to an interactive theorem prover and automatic solvers.

A proof that can be solved by an automatic solver in reasonable time is easy. The more manual interventions a proof requires, the more complex it becomes. Expanding definitions before evoking an automatic solver is still relatively easy. The most complex proofs are the ones that require creative insights.

The remainder of this section gives some examples of what to avoid when writing a model. Some of this advices where already mentioned in Section 4.1 and incorporated in these models.

- Using compound data types can negatively affect the proof complexity. Instead of using tuples or other compound structures, one should try to split them up wherever possible. If an event or action only uses some part of the information, all other parts are known to stay the same. For separate variables there are no proof obligations about these. Otherwise, we need to prove that all parts that we do not touch really are unchanged.
- Another point is to avoid nested structures. The automatic solvers are relatively good at handling the normal set operators. Using sets of sets, or functions that return sets, brings them to and above their limits. In these cases it often helps to introduce new names for some subexpressions. A syntactically simpler, but semantically equivalent, expression is often better for an automatic solver. Probably because it can easier detect relevant structures.
- Similar to the previous recommendation is to avoid indirections. Calling a function with the result of another function leads to complicated well-definedness proof obligations. One needs to demonstrate that the value returned by the first function is in the domain of the second function.

While it is difficult to completely avoid these when writing a model, keeping them to a minimum means a higher number of proofs can be done automatically.

4.3.2. Performing Proofs

Rodin provides many tools that help with discharging proof obligations. There is an interactive theorem prover and two big automatic solvers that are available as plugins.

The combination of the Atelier B [42] provers and the SMT plugin [47] is able to automatically discharge most of the generated proof obligations. It is recommended to use both solvers as they complement each other. The SMT plugin is able to solve more goals and is especially good at handling arithmetic. Most goals solved by Atelier B are also solved by SMT , a small number of goals is only solved by Atelier B and not by SMT .

To get more proof obligations discharged automatically, one can change the default strategy, used on all goals, to also utilize SMT . On the one hand this slows down the automatic proving in the background. On the other hand it is then no longer necessary to open a proof obligation, just to start the SMT solver which then is able to solve it.

If a proof is not done automatically, the user can help the solver by writing a lemma or hypothesis. This gives a new subgoal that can then be used in the main proof. Often the proofs for different proof obligations are similar and can all benefit from the same lemma. To reuse a lemma in multiple proofs, we can include it in the model as an invariant. Otherwise we need to create and prove the hypothesis for every proof obligation individually.

Writing a lemma as an invariant can also be used to make some proofs automatic. Including these inside the model makes them more explicit. It is then also possible to copy them into a similar project or reuse them in other ways. Another point to keep in mind is that some

refactorings, like renaming a machine, destroy all proofs. If a lemma is written down as an invariant, it will be preserved.

When stuck while proving an invariant, one can use the simulation feature of ProB to test it. If a counterexample is found, one knows that the invariant needs to be changed. This helps to not waste time on trying to prove something that is actually false. If no counterexample can be found, the invariant probably holds, but it might need new invariants to be proven.

The interactive theorem prover is a good tool to find new invariants that are needed to prove some other goal. One can try to reduce a goal and use case splits until only one simple subgoal remains unproven. It is often obvious that it cannot be proven under the current assumptions. This subgoal can then be turned into a new invariant. In many cases the simplifications are not necessary to perform the proof in the end. Once the new invariant is added to the model, the proof works automatically.

Rodin contains support for creating custom solvers or proof tactics. These would be helpful to perform repetitive procedures inside proofs. For example, some goal can be solved by expanding all override operators and then starting the automatic solver. It was not possible to write a proof tactic to automatically do this. The documentation of the available transformations is nonexistent or outdated. Transformations do not seem to do anything, even if the name suggests they should be executable. In the current state[1] the interface for creating custom solvers is unusable.

[1] This was tested with Rodin in version 3.4.0.

5. Case Study: Factorial

The first case study is a program to compute the factorial function. It is important in combinatorics and widely used to demonstrate the basics of recursion. In this chapter we will show a distributed algorithm for calculating this function. We will use the standard definition as function over natural numbers. The factorial of a number n is the product of all natural numbers smaller or equal than n. This can be written as $n! = \Pi_{i=1}^{n} i$ or as a recursive function as shown in Equation (5.1).

$$\begin{aligned} &factorial : \mathbb{N} \to \mathbb{N} \\ &factorial(0) = 1 \\ &factorial(n+1) = (n+1) * factorial(n) \end{aligned} \tag{5.1}$$

The factorial function allows us to study multiple interesting problems. At first, it is necessary to allocate new memory, actors in this case, as the recursive function continues its computation. Second, to compute the factorial, we need to apply basic arithmetic. This allows us to demonstrate how our verification approach handles these challenges.

In this chapter we first describe the actor implementation, by Agha [6] and Hewitt [53], of factorial. It is the target to verify and the last model corresponds to this implementation (Section 5.1). Then we explain the refinement strategy and the resulting models, in case of a single computation. This model uses the simplifying assumption that the actor system will only ever compute one function value (Section 5.2). The last section describes an extended model that is able to handle multiple function applications. Those might even happen, while the previous function is still computed. We demonstrate how the models can be extended to allow this extension and discuss the required proofs (Section 5.3).

5.1. Computing Factorial with Actors

This section explains Agha's recursive factorial algorithm for actors [6]. The algorithm works recursively, but the computation is not solely done by one function. Instead, it works by creating continuations for each step. Each of these continuations is represented as its own actor. Additionally, there is one actor called `fact` that receives requests by customers, to calculate the factorial for a certain number. In response to these requests, it starts the continuation actors to do the actual work.

This protocol contains two types of behaviors `fact` and `cont`. There exists exactly one actor with the behavior `fact`, therefore we will also refer to it as `fact`. It receives as a request the value that shall be processed and the address of the recipient of the result. If the number is 0, it will immediately send the result 1 to recipient, otherwise it creates a new actor with the `cont` behavior. This new actor keeps as state the value and the recipient of the request. After the new actor is created, `fact` sends itself an updated request, containing the decremented value and the newly created continuation as recipient. The `cont` actors await the result of the factorial of the number below the one stored by them. Once this is received, it is multiplied by the stored number and the result is sent to the stored recipient. An implementation of this algorithm, using Scala [80] and Akka Typed [70], can be seen in Listing 5.1.

```
1  final case class Request(value: Int, replyTo: ActorRef[Result])
2  final case class Result(value: Int)
3
4  val fact: Behavior[Request] = Behaviors.receive { (c, m) =>
5    m.value match {
6      case 0 => m.replyTo ! Result(1)
7      case n =>
8        val cont = c.spawnAnonymous(cont(m.value, m.replyTo))
9        c.self ! Request(m.value - 1, cont)
10   }
11   Behaviors.same
12 }
13
14 def cont(i:Int, cust:ActorRef[Result]): Behavior[Result] =
15   Behaviors.receive { (c, m) =>
16     cust ! Result(i * m.value)
17     Behaviors.same
18   }
```

Listing 5.1: Akka factorial

To get a better intuition for this algorithm, we can study a concrete example. Figure 5.1 uses a UML sequence diagram to show, how this actor system computes the factorial of 3. At first, only the actor `Factorial` exists. It receives a request with the number 3 and the recipient address `c`. As a result the continuation actor `m` is created with the state 3 and `c`. The actor `Factorial` also sends itself the new request message containing 2 and the address of `m`. This is repeated two more times and the actors `m'` and `m''` are created. When `Factorial` receives the request with the value 0, it responds to the newest actor `m''` with the result 1. This triggers a chain of result messages. The actor `m''` computes the value 1 and sends it to `m'`. This continues till `m` sends the final result 6 to the customer who sent the original request.

Computing the factorial function in this way is not more efficient than using a sequential program. It might even consume more memory because the number of actors is linear in the size of the problem. One possible advantage of this program is that it can distribute the computation of multiple calls to factorial over multiple processors. Instead of doing them

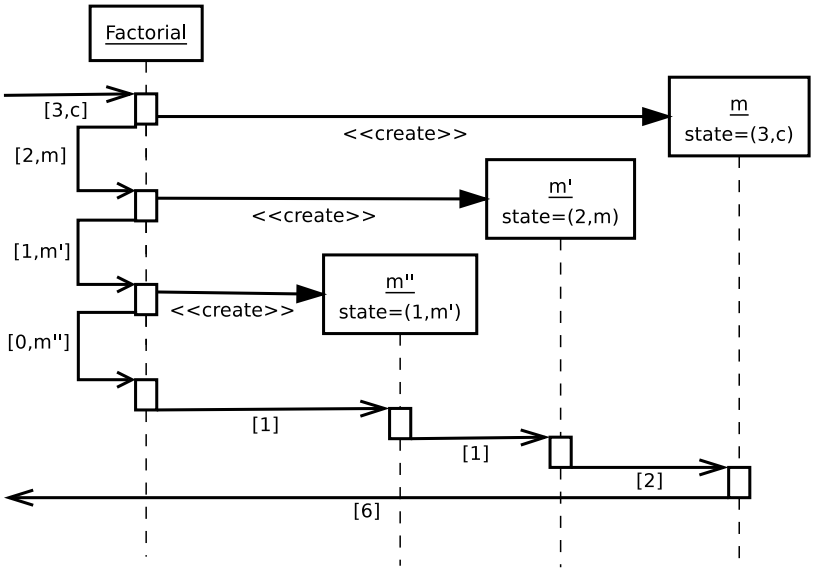

Figure 5.1.: Visualization of a factorial computation
Adapted from Agha [6]

one after another. In this case the factorial actor receives not only a single request, but multiple requests over time. The created continuation actors are distributed and can do all computations independently. The general pattern of using actors to represent continuation can also be used for more complicated computations. Thus, the techniques used to verify this case study, might be applied to other more complex applications.

5.2. Sequential Model

The fundamental assumption of this sequential model is that we are only interested in computing one value. There are no other requests that need to be handled concurrently. We also know the input value throughout the whole computation and can use it in our invariants. The modeling as well as the proofs are done in the Event-B formalism [2] using the Rodin IDE [4]. In this section we give a detailed description of all elements of the model. However, the machines are too large to be included as continuous listings. Instead, the model code is presented inline with explanations interspersed.

5.2.1. Requirements Document

For this case study, we model the computation of the factorial function via an actor system. We assume some kind of customer who sends a request to compute the factorial of some number. The first and primary functional property is that the computation terminates with the correct result.

The customer should get the requested value of the factorial.	FUN-1

Another property is that the computation must be performed by an actor system. This is not exactly a functional property, but it is the reason for including this case study.

The computation is done by an actor system.	FUN-2

The final property is a restriction, we assume about our environment.

There will be only one request in the lifetime of the system.	ENV-1

The proofs in this case study only verify that the computation is correct, given that there is only one computation. No requests are allowed, while the computation is still running, nor are more requests allowed after the first one has been completed. This restriction will be removed for the case study in Section 5.3.

5.2.2. Refinement Strategy

We start with the recursive definition given in Equation (5.1). In five refinements the details necessary for an actor system are added.

The initial model consists of an event that computes the factorial in one step.
The first refinement changes the one step computation into an iterative algorithm.
In the second refinement the used memory is made explicit in the form of a stack. We also separate the creation of the memory cells from performing the computation.
Refinement three is the first one that resembles an actor system. At that point the stack elements are replaced by actors and the computations are triggered by messages. However, the process of creating the actors is still controlled by an iterative program.
Refinement four turns this last part into an actor, controlled by sending updated messages to itself.
Refinement five changes the shared mailbox to one mailbox per actor.

5.2.3. Initial Model

The initial model consists of a context that defines the function `fact` that is used as the functional specification and a machine performing the task by using that function. The context defines two constants `fact` and `input` that represent the factorial function and the input to the computation. The definition is done via the following axioms:

axm0_0: $fact \in \mathbb{N} \to \mathbb{N}_1$
axm0_1: $fact(0) = 1$
axm0_2: $\forall n \cdot n \in \mathbb{N} \Rightarrow fact(n+1) = (n+1) * fact(n)$

axm0_3: $input \in \mathbb{N}$

Note that this definition is equivalent to the one given in Equation (5.1). The definition of `input` as a constant is justified by the environment assumption.

There will be only one request in the lifetime of the system.	ENV-1

As there is only one request, that will never change, the usage of a constant allows us to guarantee this assumption. It also enables us to refer to the value of the request, whenever it is necessary in a model.

The initial machine has only a single variable `result`, of type $\mathbb{N}$. This variable is used to store the final result of the computation. There are three events:

- The `initialization` event sets the value of `result` to 0. This is a convenient value to use for an unfinished computation. It cannot be confused with a valid result because the result needs to be strictly positive.
- The `step` event is marked as anticipated and otherwise empty. In a later refinement, it will represent one step of the computation.
- The `finish` event contains one action

  act0_0: $result := fact(input)$

  that assigns `result` to the result computed by the `fact` function. This event contains no guard and may be repeated.

A simulation of this initial model consists of the execution of the `initialization` event followed by one or more executions of the `finish` event. Looping the `finish` event will not change the value of `result`, it merely replaces it with the same value again. We do not use a guard on `finish` to restrict it to only a single execution. This would make it difficult to express and prove deadlock freedom and termination in the later models. With this approach, we can just assume the computation has terminated, when the `finish` event is executed at least once. All events introduced in later refinements are marked as convergent. Meaning, they cannot loop forever. Because the model is deadlock free at least one event can always be executed. The convergent events cannot be executed infinitely often. Only the `finish` event is non-convergent, i.e. can be executed infinitely often. It follows that every execution of the model will eventually execute the `finish` event and will execute it forever. This can be interpreted as termination of the computation. For the initial model deadlock freedom is trivial as the event `finish` is always enabled. The only invariant, the type definition of `result`, is also obviously true.

Assuming the customer from our requirements document is modeled as the variable `result`. The first functional property is satisfied by this first model with its termination proof.

The customer should get the requested value of the factorial.	FUN-1

5.2.4. First Refinement

The first refinement splits the computation into multiple steps. The new variables `tmp_result` and `val` are introduced to hold the intermediate state. The `result` variable stays part of the machine. The algorithm will do the calculation beginning with the smallest number 1. In each consecutive step the next factorial number is calculated based on the previous one which is stored in `tmp_result`. The number of remaining steps is stored in `val`. The types of this new variables are $\mathbb{N}$ for `val` and $\mathbb{N}_1$ for `tmp_result`. The machine consists of three events:

The `initialization` assigns the variable `val` to `input`. We need to do as many steps as the value of the input. To start the computation properly, `tmp_result` is initialized to 1 the multiplicative identity. As in the previous machine, `result` is set to 0.

The convergent event `ComputeStep` refines the previously anticipated event `Step`. The guard

grd1_0: $val > 0$

states that this event can only be executed if val is not 0, meaning that there is still work to do. It contains two actions

act1_0: $val := val - 1$
act1_1: $tmp\_result := tmp\_result * (input - val + 1)$

they decrement `val` and update `tmp_result` to the next factorial number.

The event `finish` refines the event of the same name. It now contains the guard

grd1_0: $val = 0$

Also, instead of assigning the final result directly, the variable `tmp_result` is assigned to `result`.

act1_0: $result := tmp\_result$

This event can now only be executed if there are no more computations to do and instead of doing the computation itself, the result is just copied.

In order to demonstrate that this machine is indeed a refinement of the previous machine, we need to confirm that all events refine their corresponding abstract event. It is also required to show that all convergent events are really convergent. That is, there exists a variant, an expression bounded from below which is decreased by every execution of the convergent event.

The event `ComputeStep` refines a previously empty event, the refinement relation is thus satisfied. It remains to be shown that this event is convergent. The variant for this machine is the variable `val`. It is a natural number and thus cannot get infinitely small and it is decremented in `ComputeStep`. Thus, it turns out that `ComputeStep` is indeed convergent.

To justify that `finish` refines its predecessor event, we need to demonstrate that whenever `val` is 0, the value in `tmp_result` is the correct final result. To prove this, we need to define two invariants:

inv1_2: $val \leq input$
inv1_3: $tmp\_result = fact(input - val)$

The function `fact`, used in invariant inv1_3, requires that its input is non-negative. We use invariant inv1_2 to show that this is always the case. Invariant inv1_3 expresses that the `tmp_result` is always the factorial number computed up to this point. Using this invariant, we can establish that the refinement for event `finish` is correct. The value of `val` needs to be 0, so `tmp_result` is the required value $fact(input)$.

We need to prove that the introduced invariants are preserved by all events. The initialization satisfies both invariants. More interesting is the event `ComputeStep`. Before the event is executed

$$tmp\_result = fact(input - val)$$

and afterwards

$$tmp\_result' = fact(input - val) * (input - val + 1) = fact(input - val')$$

which proves that the invariant inv1_3 is preserved. The proof of inv1_2 follows from the observation that `val` is only decremented and `input` is a constant.

Finally, we prove the deadlock freedom theorem. It states that at least one event must always be enabled, or expressed differently, the disjunction of all guards must be a valid expression.

thm_DLF: ⟨theorem⟩ $(val > 0) \lor (val = 0)$

This theorem follows directly from the type definition invariant of `val`, stating that $val \in \mathbb{N}$.

All proof obligations for this refinement can be discharged by the included automatic solvers [47, 42]. No manual proofs are required.

5.2.5. Second Refinement

In the second refinement the computation is split into two phases. First all numbers are pushed on a stack, afterwards they are multiplied. This brings us one step closer to the actor system, where first actors are created, then they process messages to perform the actual computation.

For this stack-based model, we need to introduce several new variables: `counter` tracks how many more elements need to be pushed, `stack` is a function that models the stack and `stack_pointer` is the current size of the stack. They are defined by the following invariants:

inv2_1: $stack \in \mathbb{N} \pfun \mathbb{N}_1$
inv2_2: $stack\_pointer \in \mathbb{N}$
inv2_3: $0 \mathinner{..} (stack\_pointer - 1) \subseteq dom(stack)$
inv2_4: $counter \in \mathbb{N}$

This means that `stack` is defined for all $\mathbb{N}$ up to, but not including `stack_pointer` which points to the next free space in the stack. The variables `result` and `tmp_result` are the same as in the previous machine. The variable `val` is no longer visible.

The machine consists of four events:

The event `initialization` sets `counter` to `input`, `stack` to an empty set, `stack_pointer` to 0. The old variables `result` and `tmp_result` are initialized as before to 0 and 1.

The event `call` is a newly introduced convergent event. It is responsible for pushing the numbers on the stack. The guard states that there are numbers left and that the computation has not started. This is needed to satisfy some invariants which will be discussed later.

`grd2_0`: $counter > 0$
`grd2_1`: $tmp\_result = 1$

The actions push the value of counter and decrement it.

`act2_0`: $counter := counter - 1$
`act2_1`: $stack\_pointer := stack\_pointer + 1$
`act2_2`: $stack(stack\_pointer) := counter$

To establish that this event is convergent, the variant `counter` is used.

The event `return` is a refinement of `ComputeStep`. Its guard requires that all elements are pushed and that the stack is non-empty.

`grd2_0`: $counter = 0$
`grd2_1`: $stack\_pointer > 0$

When executed, it pops one element and multiplies it with `tmp_result`.

`act2_0`: $tmp\_result := tmp\_result * stack(stack\_pointer - 1)$
`act2_1`: $stack\_pointer := stack\_pointer - 1$

The value of `tmp_result` is the same as in the previous refinement. The difference is that now it is computed based on a stack element and not based on a simple variable.

The event `finish` is nearly the same as in the previous refinement. Only the guard is slightly different, it still means that all computations are completed.

`grd2_0`: $counter = 0$
`grd2_1`: $stack\_pointer = 0$

The action is the same as before. It copies `tmp_result` into `result`.

To establish the refinement relationship, we need to define several gluing invariants:

`inv2_5`: $\forall n \cdot n \in dom(stack) \Rightarrow stack(n) = input - n$
`inv2_6`: $stack\_pointer + counter = val$
`inv2_7`: $counter = 0 \Rightarrow val = stack\_pointer$
`inv2_8`: $counter \neq 0 \Rightarrow val = input$

The invariant `inv2_5` allows us to know the value on the stack, which is important for the proof obligations related to the `return` event. The other three invariants state how `counter`, `stack_pointer`, and `val` are related. While the event `call` is executed, the value

of `val` stays at `input`. At the same time `counter` and `stack_pointer` are decremented and incremented, but always both. Once `counter` is 0 and the execution of `return` starts, the `stack_pointer` takes the role of `val`. The invariant `inv2_7` follows directly from `inv2_6`, it could be marked as a theorem.

Using these invariants, all proof obligations can be discharged by the automatic solvers [47, 42].

The deadlock freedom theorem

thm_DLF: ⟨theorem⟩ $(counter > 0 \land tmp\_result = 1) \lor (counter = 0 \land stack\_pointer > 0) \lor (counter = 0 \land stack\_pointer = 0)$

is also proven automatically.

5.2.6. Third Refinement

With the third refinement, we start to introduce actors. The stack, used in the previous refinement, is now represented as actors and the computation phase is controlled by messages sent between these actors. To implement these components some additional definitions are required.

An additional context provides the definition of an actor identifier. Each identifier is part of the carrier set `ACTOR_ID`. It contains identifiers for dynamically created actors, represented as natural numbers, and two special identifiers. The first one is `final_id` for the customer who is outside the modeled system and who should receive the final result. To simplify computations, it is represented by the number -1. The other special identifier is `invalid_id`. It is used as a dummy value if a variable of type `ACTOR_ID` is not used at some point in time. These constants are formally defined by the following axioms:

axm3_0: $ACTOR\_ID = \mathbb{N} \cup \{-1, invalid\_id\}$
axm3_1: $final\_id = -1$
axm3_2: $invalid\_id \notin \mathbb{N}$
axm3_3: $final\_id \neq invalid\_id$

To represent the actors, multiple new variables are introduced into the machine. One of them is `num_actors`, it stores the number of existing actors. A related variable is `actor_id`, a set that stores all actor identifiers which are currently in use. The variables are defined by these invariants:

inv3_4: $num\_actors \in \mathbb{N}$
inv3_5: $actor\_id = 0 \mathinner{..} (num\_actors - 1)$

Meaning that exactly the actor identifiers from 0 to $num\_actors - 1$ are valid and a newly created actor will get the next larger number as its identifier.

In this step only the memory for the computation is represented as actors. This corresponds to the behavior `cont` in Listing 5.1. To represent these actors, we need to store their state,

consisting of two values per actor. These values are the stored `value` and the `target`, who will receive the response, when the computation is done. Both of these state variables are represented as functions from `actor_id` to their respective types.

inv3_6: $cont\_actors\_target \in actor\_id \rightarrow (actor\_id \cup \{final\_id\})$
inv3_7: $cont\_actors\_value \in actor\_id \rightarrow \mathbb{N}_1$

The target value can either be one of the dynamically created continuation actors or the customer, who receives the final result.

The messages received and sent by the continuation actors consist of two fields. These are the content or result of the previous computation and the recipient of the new result. This is equivalent to the message type `Request` in Listing 5.1. The assumption that we will only see a single request, means that there exists at most one message at a time. This allows us to use a single set of variables for all of them. The Boolean variable `msg_exists` stores if an unprocessed message exists. The variables `msg_content` and `msg_recipient` store the values of a message. If no message exists, the value of `msg_recipient` will be invalid. The variable `active_actor` stores the actor which receives the current message if a message exists. Otherwise, it is the identifier of the last actor that was created. The formal definition of these variables:

inv3_1: $msg\_exists \in BOOL$
inv3_2: $msg\_recipient \in actor\_id \cup \{final\_id, invalid\_id\}$
inv3_3: $msg\_content \in \mathbb{N}_1$
inv3_9: $msg\_exists = FALSE \Leftrightarrow msg\_recipient = invalid\_id$
inv3_13: $active\_actor = num\_actors - 1$

Two variables are taken from the previous refinement, they are `result` and `counter`.

The machine consists of five events, one more than in the previous refinement.

In the `initialization` event most variables are set to empty or default values. The variable `counter` is set to `input` and the variable `active_actor` is set to -1, meaning that no dynamic actor exists at that point. The usage of -1 allows us to avoid defining a special case for the creation of the first actor.

act3_0: $result := 0$
act3_1: $counter := input$
act3_2: $active\_actor := final\_id$
act3_3: $msg\_exists := FALSE$
act3_4: $msg\_recipient := invalid\_id$
act3_5: $msg\_content := 1$
act3_6: $num\_actors := 0$
act3_7: $actor\_id := \varnothing$
act3_8: $cont\_actors\_target := \varnothing$
act3_9: $cont\_actors\_value := \varnothing$

The event `create` refines the event `call`. It creates continuation actors. The guard

grd3_0: $counter > 0$

is a subset of the guard of `call`. When executed, the counter is decremented and a new continuation actor is created. To create this new actor the `num_actors` variable is incremented, the `actor_id` variable is extended, and the state is added to `cont_actors_target` and `cont_actors_value`. Additionally, the `active_actor` variable is set to the id of the newly created actor.

act3_0: $counter := counter - 1$
act3_1: $actor\_id := 0 \mathinner{..} num\_actors$
act3_2: $cont\_actors\_target(active\_actor + 1) := active\_actor$
act3_3: $cont\_actors\_value(active\_actor + 1) := counter$
act3_4: $active\_actor := active\_actor + 1$
act3_5: $num\_actors := num\_actors + 1$

The newly introduced event `created` is enabled when the counter reaches zero, but no message was sent to a continuation actor.

grd3_0: $counter = 0$
grd3_1: $msg\_exists = FALSE$
grd3_2: $active\_actor = input - 1$

It is responsible for starting the computation by sending the first message to the continuation actor that was created last. To do so, the `msg_exists` flag is set to *true* and the other `msg` variables are filled.

act3_0: $msg\_exists := TRUE$
act3_1: $msg\_recipient := active\_actor$
act3_2: $msg\_content := 1$

This event is introduced in this refinement and marked as convergent. So we need to provide a suitable variant. We know that this event is only executed once and it is the only event that changes `msg_exists`. To build a variant out of this Boolean variable, we need an auxiliary function that turns the Boolean value into an integer and decreases when the input changes from $false$ to $true$. The following definition is part of the context:

axm3_4: $boolToNat \in BOOL \to \mathbb{N}$
axm3_5: $boolToNat(TRUE) = 0$
axm3_6: $boolToNat(FALSE) = 1$

By using it, the variant can be defined as `boolToNat(msg_exists)`.

The event `compute` is the receive function of the continuation actors. It is enabled whenever there exists a message for one of these actors. It also contains two additional guards to keep the system synchronized.

grd3_1: $msg\_exists = TRUE$
grd3_2: $msg\_recipient \neq final\_id$
grd3_3: $msg\_recipient = active\_actor$

When the event is executed, a message is sent to the stored target. The message contains the product of the stored value and the value received via the latest message. This corresponds to the `cont` behavior in the actor algorithm in Listing 5.1.

act3_0: $msg\_recipient := cont\_actors\_target(msg\_recipient)$
act3_1: $msg\_content := msg\_content * cont\_actors\_value(msg\_recipient)$
act3_6: $active\_actor := cont\_actors\_target(msg\_recipient)$

Additionally, the actor who processed the message is deleted, as there will be no more messages for it to process. This is done by removing it from the `cont_actors` functions and from the `actor_id` set.

act3_2: $num\_actors := num\_actors - 1$
act3_3: $actor\_id := 0 \mathinner{..} num\_actors - 2$
act3_4: $cont\_actors\_target := \{msg\_recipient\} \lhd cont\_actors\_target$
act3_5: $cont\_actors\_value := \{msg\_recipient\} \lhd cont\_actors\_value$

The `finish` event corresponds to the customer who receives the final result. It is enabled if a message arrives at this customer.

grd3_1: $msg\_exists = TRUE$
grd3_2: $msg\_recipient = final\_id$
grd3_3: $msg\_recipient = active\_actor$

When executed, the variable `result` is set to the received result in the message.

act3_0: $result := msg\_content$

In order to demonstrate that this third machine is a refinement of the second machine, we need to provide some gluing invariants. These relate the now invisible variables of the stack system, to the new variables of the actor system. The roles of `tmp_result` and `stack_pointer` are now taken by `msg_content` and `num_actor`. In fact, these variables are equivalent to its predecessors, they are just renamed to be a better fit for describing an actor system. The content of the continuation actors is equivalent to stack frames in the second refinement. The gluing invariants are formally stated as:

inv3_8: $msg\_exists = TRUE \Rightarrow counter = 0$
inv3_11: $msg\_content = tmp\_result$
inv3_12: $num\_actors = stack\_pointer$
inv3_14: $\forall x \cdot x \in dom(cont\_actors\_value) \Rightarrow stack(x) = cont\_actors\_value(x)$
inv3_15: $\forall x \cdot x \in dom(cont\_actors\_target) \Rightarrow cont\_actors\_target(x) = x - 1$

As for all the previous machines, we need to provide a deadlock freedom theorem. In this case we need to provide additional invariants to prove it because our existing invariants are not strong enough. The value of `msg_recipient` needs to be derived correctly from the other information known in a guard. There is no way to guarantee its values independently.

inv3_16: $msg\_exists = TRUE \Rightarrow msg\_recipient = active\_actor$
inv3_17: $(counter = 0 \land msg\_exists = FALSE) \Rightarrow active\_actor = input - 1$

With this additional invariants the deadlock freedom theorem can be proven:

thm_DLF: ⟨theorem⟩
$(counter > 0) \lor$
$(counter = 0 \land msg\_exists = FALSE \land active\_actor = input - 1) \lor$

$$(msg\_exists = TRUE \land msg\_recipient \neq final\_id \land msg\_recipient = active\_actor) \lor$$
$$(msg\_exists = TRUE \land msg\_recipient = final\_id \land msg\_recipient = active\_actor)$$

The proof obligations from the invariants and the deadlock freedom theorem are all automatically discharged by the solvers [47, 42]. The only manual intervention was the creation of the two additional invariants for deadlock freedom.

5.2.7. Fourth Refinement

In the fourth refinement, we replace the counter variable by a message. This message is sent by the factorial actor to itself. It corresponds to the `Request` message and the `fact` actor in Listing 5.1. To model this message, we introduce a new channel consisting of the variables `msgC_exists` and `msgC_content`. The `msgC` prefix expresses that these variables belong to the message that sends the counter. The variable `counter` from the previous refinement is removed, all other variables stay the same. The variables are defined, including the gluing invariant, as follows:

inv4_0: $msgC\_exists \in BOOL$
inv4_1: $msgC\_content \in \mathbb{N}$
inv4_2: $msgC\_content = counter$

The number and names of the events are unchanged, compared to the previous refinement. To simplify the explanation in this section, only the changes from Section 5.2.6 are highlighted. Instead of repeating all details, we refer to the previous refinement.

The `initialization` is the same as before, except the new variables that are set to

act4_9: $msgC\_exists := TRUE$
act4_10: $msgC\_content := input$

The event `create` is modified to handle the new message. Instead of checking the value of the `counter`, the existence of the message and its value are checked.

grd4_0: $msgC\_exists = TRUE$
grd4_1: $msgC\_content > 0$

The decremented `counter` is not updated directly, but instead sent as a message. The `msgC_exists` flag is already *true*, thus unchanged, and the message content is written to `msgC_content`.

act4_0: $msgC\_content := msgC\_content - 1$

The event `created` is also modified to work with the counter message. The guard now checks the existence of the message and whether its content is 0.

grd4_0: $msgC\_exists = TRUE$
grd4_1: $msgC\_content = 0$

An additional action supplements the actions of the event. After the last counter message was handled, the channel will be empty, as this event does not create a new one. Thus, the value of the `msgC_exists` flag needs to be changed.

act3_3: $msgC\_exists := FALSE$

The events `compute` and `finish` are completely unchanged compared to the previous refinement.

The proofs for refinement and deadlock freedom are done automatically [47, 42]. To establish the deadlock freedom theorem, we need this additional invariant.

inv4_3: $msgC\_exists = FALSE \Rightarrow msg\_exists = TRUE$

The deadlock freedom theorem looks as follows.

thm_DLF: ⟨theorem⟩
$(msgC\_exists = TRUE \land msgC\_content > 0) \lor$
$(msgC\_exists = TRUE \land msgC\_content = 0 \land msg\_exists = FALSE \land$
$active\_actor = input - 1) \lor$
$(msg\_exists = TRUE \land msg\_recipient \neq final\_id \land msg\_recipient = active\_actor) \lor$
$(msg\_exists = TRUE \land msg\_recipient = final\_id \land msg\_recipient = active\_actor)$

The fourth refinement satisfies the last missing property from the requirements document, although with some limitations.

The computation is done by an actor system.	FUN-2

One actor in this system is the factorial actor. Its message handling is represented by two events, namely `create` and `created`. Two different events are the most idiomatic way to represent the choice required in the message handling. The Scala code in Listing 5.1 uses a match case expression instead. The mailbox of the factorial actor is modeled by the variables `msgC_content` and `msgC_exists`. Unlike in a typical actor implementation this mailbox is not buffered, it can only contain one message at a time. This is acceptable for this model because we know that there cannot be another message at the same time. A future refinement could be used to extend the model to include a buffered mailbox. It could be represented as a set or bag (also known as multiset). This new refinement would not provide any new insights because the set would be either empty or contain exactly one element. We argue that it is therefore better represented by two variables, one of the type of the content and a different Boolean one to show whether it is empty or not.

The continuation actors are dynamically created and all share the same behavior. This behavior is modeled by the event `compute`. The state is stored in the variables `cont_actors_target` and `cont_actors_value`. The mailbox is represented by the variables `msg recipient`, `msg content`, `msg exists`, and `active_actor`. Similar to the factorial actor the mailbox is unbuffered. It is also shared among all actors of the same type, with the variable `active_actor` showing who currently owns the mailbox. This is justified because we know, based on our environment assumption, that there will only be one such message in the system at the same time.

5.2.8. Fifth refinement

The fifth and last refinement changes the mailboxes to arrays and uses separate ones for each actor. This follows the technique described in Section 4.1 and gives a model that better resembles an actor system.

We introduce the new variables for the mailboxes of the `fact` and `cont` actors. They replace the variables `msg_exists`, `msg_content`, `active_actor`, `msgC_exists` and `msgC_content`. Their types are defined by the following invariants.

inv5_0: $fact\_mail\_msgC\_content \in \mathbb{N} \pfun \mathbb{N}$
inv5_3: $fact\_index\_msgC \in \mathbb{N}$
inv5_4: $cont\_mail\_msg\_content \in (ACTOR\_ID \times \mathbb{N}) \pfun \mathbb{N}$
inv5_6: $cont\_index\_msg \in \mathbb{N}$

The state variables as well as the `result` and `actor_id` variables are unchanged compared to the previous refinement. To satisfy the refinement condition, we need gluing invariants to link the old mailbox variables to the new ones. Note that the model still adheres to the restriction that there can be only one message in all the continuation actor mailboxes. This message must be in the mailbox of the actor identified by the now hidden `active_actor` variable. The boolean `exists` flags are replaced by using an empty set instead. This gives us these gluing invariants.

inv5_1: $msgC\_exists = TRUE \Leftrightarrow ran(fact\_mail\_msgC\_content) = \{msgC\_content\}$
inv5_2: $msgC\_exists = FALSE \Leftrightarrow fact\_mail\_msgC\_content = \varnothing$
inv5_7: $msg\_exists = TRUE \Leftrightarrow ran(cont\_mail\_msg\_content) = \{msg\_content\}$
inv5_8: $msg\_exists = FALSE \Leftrightarrow cont\_mail\_msg\_content = \varnothing$
inv5_9: $\exists n \cdot msg\_exists = TRUE \Rightarrow dom(cont\_mail\_msg\_content) = \{active\_actor \mapsto n\}$

The events are adapted to the new message encoding in a relatively straightforward way. There is no change in the processing logic. We will highlight a few of these changes, but will not go over all details.

The new actions of `initialization` initialize the mailboxes with the same message as before.

act5_9: $fact\_mail\_msgC\_content := \{0 \mapsto input\}$
act5_10: $fact\_index\_msgC := 1$
act5_11: $cont\_mail\_msg\_content := \varnothing$
act5_13: $cont\_index\_msg := 0$

In the event `create` a new parameter `index` is used to access the message in the mailbox. The parameter `content` holds the content of the message and must be greater than zero, this is ensured by the guards.

grd5_0: $index \in dom(fact\_mail\_msgC\_content)$
grd5_1: $fact\_mail\_msgC\_content(index) = content$
grd5_2: $content > 0$

The response message is sent by these actions:

act5_0: $fact\_mail\_msgC\_content := \{fact\_index\_msgC \mapsto content - 1\}$
act5_1: $fact\_index\_msgC := fact\_index\_msgC + 1$

We know that the receiving mailbox is empty. This allows us to replace it directly with a new mailbox holding a single message. The actions to update the state are unchanged.

The events `created`, `compute`, and `finish` use the same technique to access the mailboxes and send messages as the `create` message. They are otherwise unchanged from the previous refinement.

The complete code of this model is available in Appendix A.1.

This final refinement fulfills all the requirements we defined at the start of the chapter. It also describes an actor system, according to the definitions from Section 4.1.

5.3. Concurrent Model

The previous model has one major limitation; it can only perform the computation once. Even though the actor program can compute the solution for multiple requests. These requests can also occur while the previous computation is still running. In that case the two computations are performed concurrently. In this section we adapt our factorial model to handle concurrent requests correctly.

The expected strategy is to add a new refinement. Unfortunately this is not possible. When looking at the initial model in Section 5.2.3, we observe that the input parameter is a constant in this model. There is also only one variable to hold the result of the computation. No refinement will allow us to get rid of this input constant, it is baked into this model at a fundamental level.

We need to create a new abstract model that is independent of the previous one. However, it is possible to reuse many of the insights we gained while creating the sequential model. The general strategy is to follow the same structure as before, but to parameterize everything with a request identifier. That associates data with the request or computation it belongs to.

Following the same structure as before, we start by listing the requirements of our model and giving an outline of the refinement strategy. Then a brief description of the model and its refinements is given.

Unlike for the sequential model the concurrent model does not contain a termination proof. The concurrent model is a reactive system that does not terminate and can handle new requests forever. So a termination proof is not possible. Instead, we want the system to fulfill a liveness property. Whenever it gets an request it will eventually answer that request. Intuitively this is a termination guarantee for each requests computation. We will not give a formal proof of this property. In Section 5.3.5, we give a formal definition of the liveness property. Its correctness is justified by using simulations and model checking.

5.3.1. Requirements Document

The requirements of this new version of the factorial model is similar to the one given in Section 5.2.1.

The first requirement is mostly the same as before, but reworded slightly to allow for multiple request. This requirement is essentially the functional correctness of a single request. Someone asks our system for the factorial of some positive number and it must return the correct result by the mathematical definition.

A customer can request the value of the factorial for some positive number and get the correct result.	FUN-1

To actually satisfy this requirement, we need to add an assumption about our system.

The system behaves in a fair way.	ENV-1

This is much weaker than the assumption in the sequential case, but an adversarial environment can stop our system from delivering the correct result. Assume a system that will only accept new requests, but never do the calculations for them. Even through the code that should do the calculation is entirely correct. With this assumption, we state that when the system can do the computation, it will eventually perform it. A more in depth discussion of this phenomenon is included in Section 5.3.5.

Once again the second requirement states that our system must be an actor system.

The computations are done by an actor system.	FUN-2

In the sequential model, there is an assumption about the environment. It states that there is only a single computation. Our goal of this rewrite of the model is to create a concurrent model that can handle multiple requests. Therefore, we remove this assumption and instead replace it by two new requirements.

The third requirement expresses our desire to allow more than one request.

The system supports arbitrary many requests.	FUN-3

The actor system can handle requests that occur while the previous computation is still running. In that case both computations are performed concurrently. We want our model to capture this behavior.

Multiple requests are computed concurrently.	FUN-4

These requirements describe a system that is strictly more general than the sequential one in Section 5.2. The behavior of the sequential system is included in the behavior of the concurrent system. If there is only one request the two will behave the same, modulo some changes to the data structures. But every trace that includes more than one request cannot be produced by the sequential model. This explains why the concurrent model is not a refinement of the sequential model. Instead, we must start over.

5.3.2. Refinement Strategy

While we cannot build on the sequential model, we can reuse its strategy and structure. By building the sequential model, we demonstrated that this strategy can be used to build a correct implementation of the factorial actor system. The used refinements and invariants are appropriate for the task. For the concurrent model, we keep as much as possible of the sequential model the same. However, we change all variables such that they can hold one value for each computation, instead of just a single value. Another major change, is done for the initial model. In that case, we need to include a new event and new variables to represent the requests and results. Starting with this new initial model, we follow the same refinement steps as in the sequential case.

The initial model accepts requests, containing a number, and computes the factorial of this number in a single step. This uses the recursive definition of factorial.

The first refinement changes the one step computation into an iterative algorithm.

In the second refinement the used memory is made explicit in the form of stacks. Each request has its own stack.

Refinement three is the first one that resembles an actor system. At that point the stack elements are replaced by actors and the computations are triggered by messages. However, the process of creating the actors is still controlled by an iterative program. All actors and all messages contain an identifier that links them to the request they are calculating the result for.

Refinement four turns this last part into an actor, controlled by sending updated messages to itself.

Refinement five changes the shared mailbox to one mailbox per actor.

5.3.3. Initial Model

The definition of the recursive factorial function is included in a context. It is equivalent to the one in the sequential model in Section 5.2.3. Additionally, the context defines a carrier set `REQUEST_ID` used to identify the requests sent to the system.

To store the requests and results, the model contains two variables: `tasks` and `results`. Their types are given by these invariants.

inv0_0: $tasks \in REQUEST\_ID \pfun \mathbb{N}$

inv0_1: $results \in REQUEST\_ID \pfun \mathbb{N}_1$

inv0_2: $dom(results) \subseteq dom(tasks)$

The `tasks` variable contains all requests that are sent to the system. It is an array, or function, indexed by the identifier of the request. The `results` variable contains the result of a requested computation if it exists. The last invariant ensures that a result can only exist for a computation that was started, i.e. it is contained in `tasks`. On the other hand not every started computation already has a result, some might still be in progress. Therefore the domain of `results` needs to be a subset of the domain of `tasks`.

Apart from the `initialization` event, the model consists of two events `start` and `finish`. This model does not contain an anticipated step event. Instead, the step events in later refinements are direct refinements of the implicit `skip` event. Using an anticipated event that is later refined by a convergent event, is helpful for the termination proof. As the concurrent model does not contain a termination proof, this is not necessary for this model. It also does not contain any variants, for the same reason. The events are as follows:

Both variables are initialized with an empty set. No requests have been sent and no results calculated.

act0_0: $tasks := \varnothing$
act0_1: $results := \varnothing$

The `start` event models the sending of a new request. It contains two parameters: `input` and `request`. The first is the positive number, the factorial should be computed from. The second is used to choose a fresh identifier for this request.

grd0_0: $input \in \mathbb{N}$
grd0_1: $request \notin dom(tasks)$

When this event is executed, a new entry in `tasks` is created using these two parameters.

act0_0: $tasks(request) := input$

The other event is `finish`. It picks one `task` that does not yet have a result.

grd0_0: $task \in dom(tasks)$
grd0_1: $task \notin dom(results)$

For this task the result is computed using the `fact` function and entered into `results`.

act0_0: $results(task) := fact(tasks(task))$

It is important for future refinements to have separate events for creating and finishing a task. In the initial model both could be done as two actions of the same event. For the later refinements, however, we refine the `finish` event the same way as in the sequential model, while the `start` event takes the place of the initialization event from the sequential model. This corresponds to the sequential model where only a `finish` event exists. Instead of being created by the `start` event, in the sequential model the starting value is a constant.

The initial model already satisfies two of the requirements.

A customer can request the value of the factorial for some positive number and get the correct result.	FUN-1

The customer is assumed to control the `start` event and can call it to create new tasks. To get the results, the customer can access the `results` variable for its own request and read the result. By directly using the mathematical definition of factorial, it is also guaranteed that the result is correct.

This assumes that the fairness assumption holds.

The system behaves in a fair way.	ENV-1

Otherwise the system could forever execute the `start` event but never the `finish` event. In this case the correct results would not be entered into the `results` variable.

The system supports arbitrary many requests.	FUN-3

This requirement is satisfied because we use an unbounded set for `REQUEST_ID`. There always exists a fresh request identifier that can be used to create a new task that will be answered later.

5.3.4. Refinements

This section describes how the concurrent models are derived from their sequential counterparts. Instead of going over all invariants and events in detail, we explain the principles and focus on the consequences of these changes.

The general structure of the models stays the same, including all the invariants and event guards and actions. However, all variables are now parameterized by `REQUEST_ID`. This means that all parts of the model that access them must be adapted. We cannot refer to these variables anymore, we rather need to operate on their value for a specific task.

Invariants are changed to piecewise definitions. For example, the variable `tmp_result` is part of the following invariants in the first refinement of the sequential model (Section 5.2.4). Here `tmp_result` is the result calculated up to this point; `val` is a counter that moves from `input` towards zero.

`inv1_1`: $tmp\_result \in \mathbb{N}_1$
`inv1_3`: $tmp\_result = fact(input - val)$

In the concurrent model the invariants need to be defined for each of the tasks. As mentioned previously the types of `tmp_result` and all other variables are now functions. The invariant `inv1_5` corresponds to the sequential invariant `inv1_3`. A *for all* quantifier, over all existing tasks, is used to express this invariant for all states. To make the function applications in this definition well defined, we need to ensure that all used functions share the same domain.

inv1_2: $tmp\_results \in REQUEST\_ID \pfun \mathbb{N}_1$
inv1_3: $dom(tmp\_results) = dom(tasks)$
inv1_5: $\forall i \cdot i \in dom(tasks) \Rightarrow tmp\_results(i) = fact(tasks(i) - vals(i))$

Instead of using a partial function for the type of `tmp_result` and an invariant to define its domain, we can change the definition to

inv1_2: $tmp\_results \in dom(tasks) \to \mathbb{N}_1$

Events are translated in a similar way. Each event gets a new parameter `task` that determines for which task this step of the computation is performed. It must be a task that has already started, but not yet finished i.e. it is in `tasks` and not in `results`. All other guards are expressed over the variable values for this specific task. Actions also only access the variable values for the current task and update them for the same task. No event accesses the values of more than one task at a time. An example for an action in the first refinement of the concurrent model is:

act1_1: $tmp\_results(task) := tmp\_results(task) * (tasks(task) - vals(task) + 1)$

It is taken from the `compute_step` event. For the given `task` the variable `tmp_result` is updated in the same way as in the sequential case; it contains the corresponding action:

act1_1: $tmp\_result := tmp\_result * (input - val + 1)$

A major consequence of these translations to a concurrent model is that the complexity of the proofs rises significantly. In the sequential model all proofs where done by the automatic solvers [47, 42]. For the concurrent model, a small number of proofs required manual intervention.

One example is from the first refinement. We need to prove that the `compute_step` event preserves the invariant `inv1_5`. The invariant is already described above. The event contains two actions `act1_1` (mentioned earlier) and

act1_0: $vals(task) := vals(task) - 1$

The generated proof obligation is

$$\begin{aligned}&\forall i \cdot i \in dom(tasks) \Rightarrow \\ &\quad (tmp\_results \domsub task \mapsto tmp\_results(task) * (tasks(task) - vals(task) + 1))(i) \\ &\quad = \\ &\quad fact(tasks(i) - (vals \domsub task \mapsto vals(task) - 1)(i))\end{aligned}$$

To prove this, we have to simplify it enough to allow the automatic prover to complete the proof. At first, we eliminate the *for all* quantifier and do a case split on $i = task$. The subgoal for $i \neq task$ is automatically discharged. All updates are done on the values for task, all other entries are unchanged and thus preserve the invariant. The other subgoal for $i = task$ requires us to manually instantiate the *for all* quantifier in the invariant `inv1_5`. By substituting $task$ for i, we obtain $tmp\_results(task) = fact(tasks(task) - vals(task))$.

Element Name	Total	Auto	Manual
ctx0	2	1	1
ctx3	0	0	0
m0	8	8	0
m1	26	24	2
m2	50	47	3
m3	129	110	19
m4	33	33	0
m5	57	53	4
Σ	305	276	29

Table 5.1.: Proof Statistics for Concurrent Factorial

Using this hypothesis, the automatic prover can discharge the final subgoal. In some cases it is also necessary to expand the definition of *override* ($\lhd$) first. We are not sure exactly under what circumstances this is necessary because in many cases the simplifier handles *override* just fine.

The manual proofs for this model required one or multiple of the following techniques:

- Expanding definitions of set operators, mostly *override* ($\lhd$) and *biimplication*.
- Instantiating quantified formulas.
- For equality propositions involving complicated expressions, it can be necessary to add hypotheses that some subexpressions are equal. Take, for example, a goal $\alpha * \beta = \gamma * \delta$, where $\alpha, \beta, \gamma, and \delta$ are subexpressions. It can help to define the two hypothesis $\alpha = \gamma$ and $\beta = \delta$. Of course only if they are actually true. This new subgoals often allow the automatic prover to succeed, even if the original goal was to complicated.
- Create invariants if a hypothesis is needed in multiple proofs. For example, in the third refinement we use the following theorem to help with the proofs of other invariants.

  inv3_21: ⟨theorem⟩ $\forall t \cdot t \in dom(tasks) \Rightarrow dom(cont\_actors\_target(t)) = 0 \mathinner{..} num\_actors(t) - 1$

- Applying equality rewrites. Sometimes the prover needs to be explicitly told to use a certain equality assumption.

The most manual proofs are required for the refinement between the second and third machine. This is where the stacks are turned into actors. The gluing invariants contain properties about deeply nested structures. An array of stacks on the one hand and multiple actors on the other hand.

Table 5.1 contains the number of proof obligations for each machine and context and how many of them where solved automatically or had to be done by hand. The automatic prover was configured to try all solvers including SMT . This means, all proofs that are counted as manual actually involved some intervention. They cannot be solved by just manually invoking a specific solver.

Final Machine

The fifth and final machine uses the following types for the actor states and mailboxes.

inv3_08: $cont\_actors\_target \in dom(tasks) \rightarrow (ACTOR\_ID \nrightarrow ACTOR\_ID)$
inv3_10: $cont\_actors\_value \in dom(tasks) \rightarrow (ACTOR\_ID \nrightarrow \mathbb{N}_1)$
inv5_00: $fact\_mail\_msgC\_content \in dom(tasks) \nrightarrow \mathbb{N}$
inv5_1: $cont\_mail\_msg\_content \in (ACTOR\_ID \times REQUEST\_ID) \nrightarrow \mathbb{N}_1$

The actual actor identifier consists of two parts: the `ACTOR_ID` and the `REQUEST_ID` ($dom(tasks)$). This can be seen as a separate namespace for each computation. The `ACTOR_ID`s can overlap because they are distinguished by the `REQUEST_ID`. Instead of using a tuple for the actor identifier, a curried definition is used. This makes some proofs simpler.

The factorial actor reuses the task identifier as message identifier. Its mailbox will contain at most one message per task, therefore there is no danger of confusing messages.

The mailbox of the continuation actors are one place. This is justified because we know that a continuation actor will only receive a single message in its lifetime. Introducing message identifiers would only complicate the model and the proofs.

The complete code of the final machine is available in Appendix A.2. This last machine fulfills the remaining two requirements. Actors belonging to different tasks can act independent of each other, and thus perform their respective computations at the same time.

The computations are done by an actor system.	FUN-2

Multiple requests are computed concurrently.	FUN-4

5.3.5. Simulation and Model Checking

This section explains how ProB [68] helps to check the resulting model. A special focus lies on testing the liveness properties that where not formally proven.

One difficulty with this model is that when loaded into ProB, as is, the model checker cannot simulate it. The problem is the recursive definition of `fact` in the first context. ProB is not able to find a value for this constant. This stops us from simulating any part of the model. ProB's recommendation to create a simulation refinement does not help because it tries to simulate all machines at once. An unrepresentable constant in the first context blocks everything.

Instead we created a copy of the whole project containing only the machines one to five. We removed all references to the `fact` function and used machine one as the initial one. This allows us to simulate the remaining model. While the changes do not affect the validity of

our testing, it is still unsatisfying that the whole code needs to be duplicated. When doing a change in the original model, we need to manually copy this change to the simulation model. This bears the danger of letting these to get out of sync.

ProB is only able to work with finite state spaces. Therefore we need to define ranges for our variables. We allow inputs in the range $[0, 3]$ and two different request identifiers. The other ranges are inferred to be sufficiently large for this model.

For the sequential model, we require that the computation terminates. In the concurrent case the property is a more complex liveness property. Whenever a request gets sent to the system, it shall respond with the correct result in a finite time.

More formally, this can be expressed using temporal logic as given in Equation (5.2).

$$\forall t \in REQUEST\_ID, n \in \mathbb{N} \cdot \mathbf{G}([Start(n, t)] \Rightarrow \mathbf{F}[Finish(t, final\_id, n!)]) \tag{5.2}$$

For all possible request identifiers and inputs it holds that whenever the `Start` event occurs at some later point the `Finish` event must happen. The request identifier must be the same for both events and the result in `Finish` must be the correct value. This specification only holds for our system if all events are fair. The `Start` event is not actually required to be fair, but we will assume that all events are fair nevertheless. If one event was not fair, the system could accept a new request in each step without doing any computations on the already existing tasks. Thus, these tasks would never receive an answer.

While Equation (5.2) is a formal specification of our system, we cannot check it. ProB does not allow us to write LTL formulas with quantifiers. The Event-B plugin of ProB does also not support event parameters in LTL formulas. They are, however, supported for simulations. The new standalone JavaFx client for ProB2 supports event parameters in LTL , but still no quantifiers. Yet, that is no real problem because we are still limited to finite models. Therefore we can simply unroll the quantifiers.

For some unknown reason (probably a bug) the new ProB client cannot open the last machine of this model (`m5`), the previous refinement `m4` works fine.

In the context, we defined two distinct `REQUEST_ID`s: `r1` and `r2`. Instead of testing the system for all possible inputs, we use the input 3 for both computations. Given the formula in Equation (5.3) ProB is now able to check all interleavings of these two computations.

$$\mathbf{G}(([Start(3, r1)] \Rightarrow \mathbf{F}[Finish(r1)]) \wedge ([Start(3, r2)] \Rightarrow \mathbf{F}[Finish(r2)])) \tag{5.3}$$

According to ProB this property holds. That might be a bit surprising as we did not add a fairness requirement. However, we had to use a finite range for request identifiers. At some point, there is no fresh identifier to create a new request, and the system is forced to complete the remaining tasks. For our finite model the property holds indeed.

To test the final machine, we used the older Event-B plugin of ProB to manually test it. This included runs with two requests, where the second one was started at different points of the execution of the first one.

By testing and model checking finite versions of our model, we can be reasonably certain that the liveness property also holds for the unbounded version.

5.4. Conclusion

In this chapter the development of an actor system to compute the factorial function was described. Factorial serves as a representative example for a recursive algorithm. The model is created in two major phases. First, a sequential version is built; it only describes a single execution of the algorithm. In the second phase, the sequential version is used as a template to create a distributed and concurrent version. The resulting system can handle multiple computations concurrently.

Our model starts from a simple mathematical specification and uses multiple refinements to turn it into an actor system. This follows the strategy from Section 4.2.1. The main challenges in this model are the dynamic creation of actors, the usage of integer arithmetic, finding the proper gluing invariants, and the refactoring into a concurrent model.

The refinement between machine two and machine three was the most complex. It changes the model from using a stack to using actors for the main computation. This required many invariants to properly link the stack frames from the previous model to the actor states and messages in the next model. Although all proofs in the sequential model are performed automatically, the interactive proof editor helped tremendously when creating the model. It can be used to explore why some proofs fail and thus helps to discover new invariants. Often the required connections between some variables are non obvious and can only be found when attempting to prove another invariant.

Integer arithmetic was less of a challenge than initially expected. Modern SMT solvers have no problems with arithmetic expression of the size and complexity used in this model. Only when the arithmetic is performed on complex expressions, it is sometimes necessary to simplify the operands first.

When creating the concurrent model, using the sequential version as a template helped a lot. In contrast to the sequential version the concurrent one required manual proofs. The knowledge that the algorithm is correct and the invariants are strong enough for the sequential case, meant that we could focus on the proofs. The concurrent version essentially contains the sequential version for each task and they cannot interfere with each other. We know that the proofs worked once so it should also work if we perform the same task multiple times concurrently. If a proof obligation required a manual proof we could be sure that all required invariants exist and the proof is indeed possible. Proving these proof obligations while at the same time trying to discover the necessary invariants would be extremely hard.

This chapter demonstrates that our method of proof-based development of actor systems can be used to develop a distributed and concurrent actor system from a recursive specification.

6. Case Study: Chat Server

This chapter covers the development and formal proof of a chat server. It is inspired by an example[1] from the Akka Typed documentation [70]. The focus of this chapter, is on developing a reactive actor system which uses multiple message types and creates actors while running. The model describes a chat server. Clients can subscribe to it and then send messages. These messages are forwarded to all subscribed clients.

Initially, the system consists of one server and multiple clients. A client can try to subscribe to the server and provides a user name while doing so. When the request is accepted, the server creates a new session and sends the sessions address to the client. Otherwise the server sends a rejection message. The client can send a message to its session to be sent to all other subscribed clients. The server distributes the message to all subscribed clients using the sessions as an intermediary.

The remainder of this chapter is structured as follows. In the next section (6.1) we explain why this case study is interesting and how it relates to real world systems. In Section 6.2 the original version, in Scala Akka, of this system is described. This is the target, we want the final machine to correspond to. Section 6.3 gives all the requirements the system should fulfill. Section 6.4 describes the refinement strategy which is implemented in Section 6.5 to Section 6.11. Section 6.12 details how the system was tested and model checked. Finally, Section 6.13 concludes this chapter.

6.1. Motivation

Chat servers are a common use case for actor systems. The used model also can be seen as a simple instance of a publish-subscribe network. Clients can subscribe to topics and then get a notified when a new message is posted on this topic. A commonly used protocol is MQTT [19] which is used in the Internet of Things.

We use session actors to forward messages between the clients and the server. In a purely local system these might not be necessary, yet in a distributed system they can reside on the same node as the server actor and reduce its load. Delivering the messages to the local sessions is very fast and reliable. The sessions would then do the slower operation of sending the message to the clients over a network. This might include a handshake to detect lost messages and resend them if necessary. Doing this in a dedicated actor means

[1] `https://doc.akka.io/docs/akka/2.5/typed/actors.html#a-more-complex-example`

that the server actor can respond faster to important messages. It cannot be caught up in a long handshake with a single client which would stall the whole network.

This model covers many techniques that could be used to model larger real-world systems.

6.2. Akka Version

An implementation of the chat server case study is provided in Listing 6.1. It is to be understood as a minimal executable implementation. A real chat server would need to include many additional functionalities. For example authenticating users, an unsubscribe feature, status information, or multiple chat rooms.

The system contains three different types of actors `server`, `client`, and `session`. Each of them has its own message type. The `server` accepts messages of type `ServerMsg`. They can either be a subscription request, sent by a client, or a `Publish` message from a session. A client can receive three different messages. The messages `Success` and `SubscribeFailed` are sent by the server in response to a `Subscribe` message. The `Posted` message is used to inform a client if someone has sent a new message to the chat room. Sessions are used to forward and buffer messages between the server and their client. The `Send` message is sent by the client and the `Forward` message is sent by the server.

The server's behavior is implemented by the `server` method. It uses a parameter to store all sessions that have been created. There is one session for each subscribed client. Upon receiving a `Subscribe` message the server will always accept it and create a new session for the client. The client gets send a confirmation containing the address of the newly created session. At the end the server's session list is updated. When a `Publish` message is processed the server forwards it to all sessions it has stored.

The behavior of sessions is defined in the `session` method. It has three parameters used to store the state of a session. These are the address of the server, the user name of its client and the address of its client. If it receives a `Send` message from its client it transmits this message and the client's user name to the server. Upon getting a `Forward` message from the server the session delivers the same message to its client.

No implementation of a client is included in this listing. An implementation would first try to subscribe and then send a message when prompted by the user, and display incoming messages.

Figure 6.1 depicts an execution of this system with two clients. First, Client 2 subscribes successfully, and then Client 1. After both clients are subscribed, Client 1 uses its session to send a message. The message is forwarded to the server which distributes it to all sessions. This includes Session 1. The sessions then deliver the message to their respective clients.

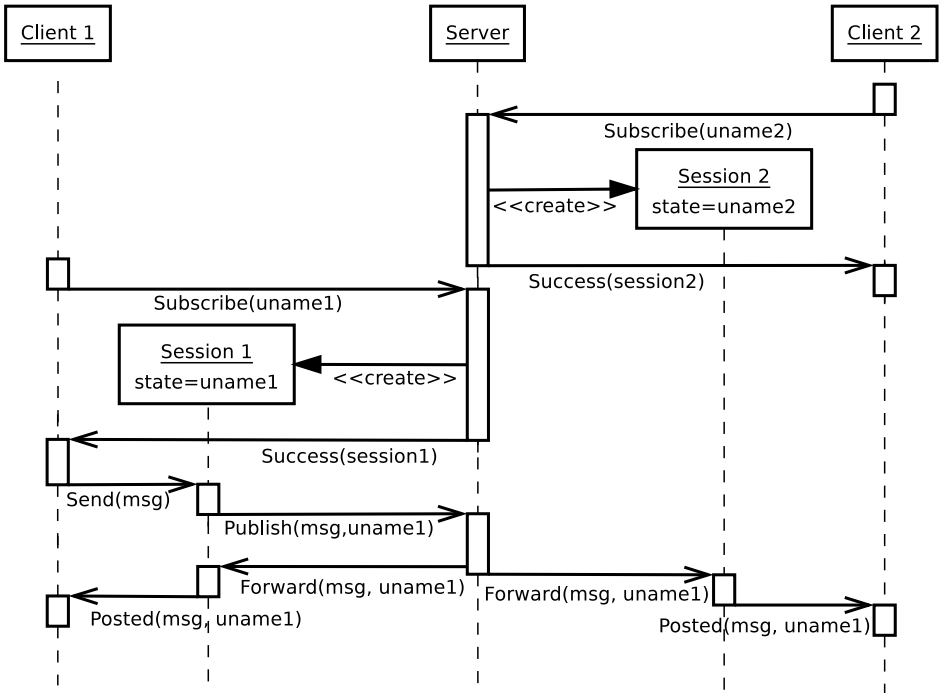

Figure 6.1.: Simulation of the chat server actor system

```
1  sealed trait ServerMsg
2  final case class Subscribe(uname: String,
3                             replyTo: ActorRef[ClientMsg]) extends ServerMsg
4  final case class Publish(uname: String, message: String) extends ServerMsg
5
6  sealed trait ClientMsg
7  final case class Success(session: ActorRef[Send]) extends ClientMsg
8  final case class SubscribeFailed() extends ClientMsg
9  final case class Posted(uname: String, message: String) extends ClientMsg
10
11 trait SessionMsg
12 final case class Send(message: String) extends SessionMsg
13 final case class Forward(uname: String, message: String) extends SessionMsg
14
15 def server(sessions: List[ActorRef[SessionMsg]]) : Behavior[ServerMsg] =
16   Behaviors.receive { (context, message) =>
17     message match {
18       case Subscribe(uname, client) =>
19         val ses = context.spawnAnonymous( session(context.self, uname, client) )
20         client ! Success(ses)
21         server(ses :: sessions)
22       case Publish(screenName, message) =>
23         sessions.foreach(_ ! Forward(screenName, message))
24         Behaviors.same
25     }
26 }
27
28 def session( server: ActorRef[Publish], uname: String,
29              client: ActorRef[ClientMsg]): Behavior[SessionMsg] =
30   Behaviors.receiveMessage {
31     case Send(message) =>
32       server ! Publish(uname, message)
33       Behaviors.same
34     case Forward(uname, message) =>
35       client ! Posted(uname, message)
36       Behaviors.same
37   }
```

Adapted from the Akka Typed documentation [70].

Listing 6.1: Akka chat server

6.3. Requirements Document

As a first step, when creating an Event-B model, we need to list and define all our requirements. We can then check what requirements are fulfilled in each refinement step. In the last refinement all requirements must be realized.

The chat server receives messages and sends them to clients.	FUN-1

This first requirement provides a general context for our project. We want to model a chat server and multiple clients. The clients can communicate with each other by using this chat server. This requirement is still vague the other requirements define the desired behavior in more detail.

Clients need to subscribe, to send or receive messages.	FUN-2

From this requirement we can infer the following things. Not every client can send or receive messages. There is some process for a client to subscribe to the server. It needs to be ensured that only the subscribed clients can send and receive messages.

A client needs to be authenticated before it can subscribe.	FUN-3

Subscribing is not always successful. The server needs to authenticate the client, before it accepts a new subscriber. How this authentication is done will not be part of this model. Instead, the server chooses nondeterministically whether the authentication was successful or not.

When subscribing, the client chooses a unique user name.	FUN-4

When a client enters the chat room, it must pick a user name. This cannot be changed later on and no two clients can have the same user name.

If a subscription fails, the clients gets notified.	FUN-5

If the server rejects a client trying to subscribe, it needs to communicate this to the client.

Messages contain the user name of the sender.	FUN-6

If a client receives a message from the chat room, it must contain the user name of the original sender. This is the user who posted the message to the chat room.

A sent message will eventually be delivered to all subscribed clients.	FUN-7

This is a liveness property of our system. A message that is sent to the chat room must reach all subscribed clients. We count all clients as subscribed that are known as subscribed to the server when it gets the message. A client which subscribes after the message reached the server, will not get this message.

The communication between subscribed clients and the server is done via sessions.	FUN-8

Except for subscribing, clients and server do not communicate directly with each other. For each subscribed client a session is created. The session buffers the communication between client and server in both directions.

The whole system is an actor system.	FUN-9

As for all examples and case studies in this thesis, we want the modeled system to be an actor system. The criteria for deciding if an Event-B model is an actor system are described in Section 4.1.

All actors in the system are fair.	ENV-1

The fairness property in requirement FUN-7 only holds if all actors are fair. Therefore, we need this environmental assumption. An actor is fair if it eventually processes each message in its mailbox.

6.4. Refinement Strategy

This project uses a mostly horizontal refinement strategy; each refinement adds new functionality. The way how something is modeled rarely changes between different refinements. All machines are actor systems. Except some earlier machines which can directly access the state of other actors.

We start with a simple actor system consisting only of the server and one actor for all clients. The clients actor can send a message to the server. The same message gets sent back to the clients actor. The required features are gradually introduced in the following refinements.

The first refinement splits the clients into multiple actors. It also adds a subscribe event. Each client is either subscribed or not.

In the second refinement user names are introduced.

The third refinement makes the subscription process asynchronous. It is now done by sending messages, instead of directly accessing the servers memory. The server can accept or deny a subscription request.

The fourth refinement introduces sessions. However, they are only used to send messages from the client to the server. Not yet in the other direction.

In the fifth refinement sessions are also used for the messages sent from the server to the client.

The models in this chapter are described on a more abstract level, than in the previous chapter. We focus on the different actor messages and states, instead of how they are actually encoded in Event-B. For a detailed description of actor encodings in Event-B refer to Section 4.1. We will mention events or Event-B variables if they are especially interesting.

6.5. Initial Model

We use the following carrier sets: `MSG_CONTENT` for the content of messages instead of string, `CLIENT_ID` is used for the addresses of clients, `UNAME` is the set of all possible user names, `SESSION_ID` is used for session addresses and `MSG_ID` is used to keep messages unique in a mailbox. Many are only used in later refinements.

The initial model consists of two actors: one named `clients` and another one named `server`. None of them contains any state. The model contains three events each corresponding to sending or receiving a message.

ClientsSend
: The `clients` actor can send the message `Send` to the server, whenever it wants. We assume this is triggered by some external user input. The message has one field of type `MSG_CONTENT`.

ClientsReceive
: This event models the `clients` actor receiving a `Reply` message. It does not create a new message, it only removes the message from its mailbox. We assume that the content of the message is displayed to a user, but that is not part of the model.

ServerReply
: When the server receives a `Send` message, it sends a `Reply` message back to the `clients` actor. This message contains a `MSG_CONTENT` field. The content of the reply message must be identically to the one the server received previously.

Figure 6.2 contains a UML sequence diagram visualizing the execution of this model. A single message is sent to the server and returned to the client.

This initial model captures the general structure of the chat server case study. There are two groups: the clients and a server. The clients can send messages and the server replies to them. This fulfills the first requirement.

The chat server receives messages and sends them to clients.	FUN-1

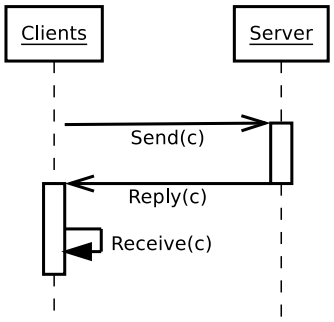

Figure 6.2.: Simulation of chat server model 0

6.6. First Refinement

In the first refinement each client gets its own actor. We include a global state variable to store a set of all subscribed clients. This variable can be accessed by the clients as well as the server. Additionally, we introduce a new data structure called `in_flight`. It stores all messages that have been sent by the server, but have not yet been received by all clients. This structure is not used in the actual computation. It is only required for the refinement proof. A gluing invariant states that `in_flight` is equivalent to the client mailbox in the initial model. The events of this machine are:

ClientSend (refines ClientsSend)
: This event now checks the set of subscribers. It can only be executed if the client is in this set. The sent message is unchanged.

ClientReceive (new)
: Removes a message from the mailbox of a single client. This cannot be a refinement of ClientsReceive because that would require that the message is removed from all mailboxes.

AllClientsReceived (refines ClientsReceive)
: This is an observation about the system and not a real event, but is modeled as an event in Event-B. It describes the state when a message sent from the server was received by all subscribed clients. The event is enabled if a message from the `in_flight` structure is no longer in any subscriber's mailbox. When it is executed, it removes the message from `in_flight`.

ServerReply (refines ServerReply)
: Instead of sending the message to the single client actor, it now sends it to all subscribers. Additionally, the message is added to `in_flight`.

SubscribeSuccess (new)
: This is used to add a client to the subscribers set. It is only allowed to add the client who executes it and only if this client is not already in the set. The `client` parameter of the event describes which actor executes the event.

SubscribeFail (new)
: Currently this event does not contain actions. It can be called with any `CLIENT_ID` and represents a failed subscription request.

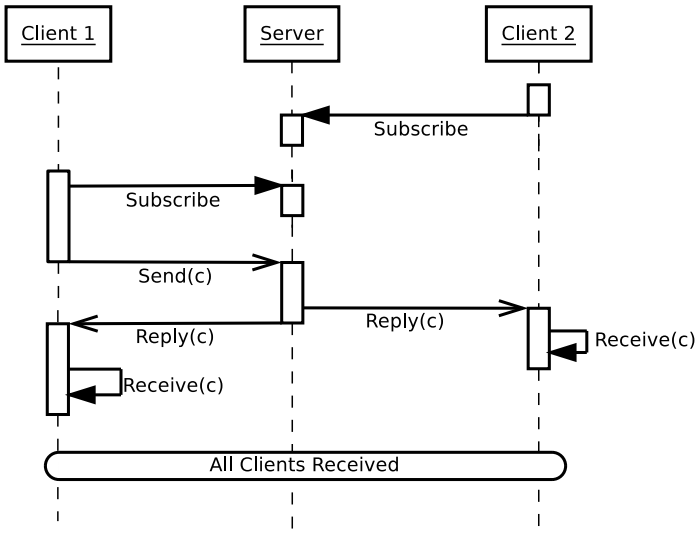

Figure 6.3.: Simulation of chat server model 1

Figure 6.3 shows the simulation of the first refinement with two clients. First, both clients subscribe successfully and then a message is sent to the server. The message is then sent back to both clients. After both clients received the message, the observational event AllClientsReceived is triggered.

The first refinement already satisfies two additional requirements.

Clients need to subscribe, to send or receive messages.	FUN-2

We include a set of subscribers, a method to subscribe, and we check if a client is allowed to send.

A sent message will eventually be delivered to all subscribed clients.	FUN-7

This requirement is arguably already fulfilled by the initial model. However, we include it here because only with multiple distinct clients the requirement is covered completely. The observation AllClientsReceived helps to identify when the condition is met. Of course, this only holds under the environmental assumption.

All actors in the system are fair.	ENV-1

A more detailed evaluation of this liveness condition is done in Section 6.12.

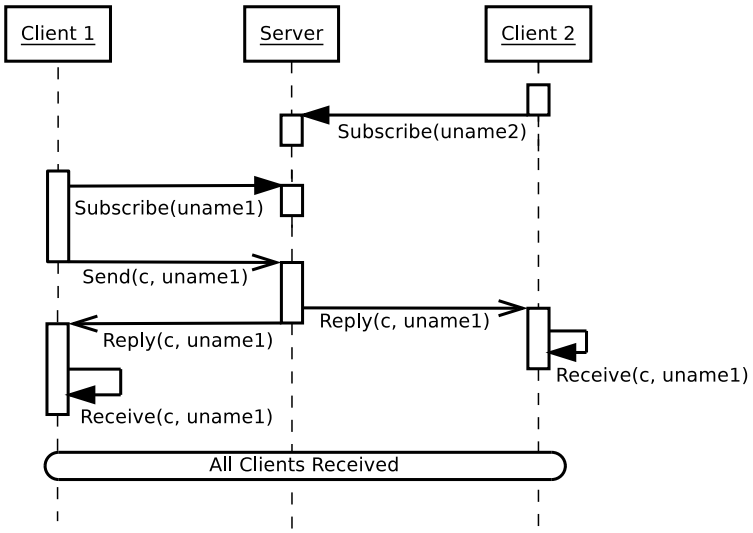

Figure 6.4.: Simulation of chat server model 2

6.7. Second Refinement

This is a relatively minor change compared to the last model. We add a new user name field to all messages. The server also stores a set of user names to ensure that they are unique.

ClientSend (refines ClientSend)
: The message now also contains the user name. The client must use the same user name, it used when subscribing.

ClientReceive (refines ClientReceive)
: The event contains a new parameter for the user name of the sender. This models that the user name is displayed together with the content.

AllClientsReceived (refines AllClientsReceived)
: The user name field is added here as well, to ensure that all clients got the message with the same sender included.

ServerReply (refines ServerReply)
: The server adds the user name, it got in the `Send` message, to the `Reply` message.

SubscribeSuccess (refines SubscribeSuccess)
: We need to check if the user name requested by the client is already in use. The client is only allowed to use a previously unused name.

SubscribeFail (refines SubscribeFail)
: This is unchanged except for a new user name parameter.

Figure 6.4 illustrates the execution of the second refinement; including the newly added user names.

Adding user names realizes two additional requirements.

Messages contain the user name of the sender.	FUN-6

All messages now contain the user name of the original sender.

When subscribing, the client chooses a unique user name.	FUN-4

The `SubscribeSuccess` event contains a guard to ensure that all user names are unique.

6.8. Third Refinement

In the third step, we refine the subscription process. Instead of directly writing to the subscribers set, a client is now required to send a request to the server. The server checks this request and responds with either success or fail.

Previously, we had the requirement that only subscribers can send messages. This gets a little more involved. What about a client where the subscription request was granted, but it has not yet received the notification. The client does not know if it is subscribed or not, only the server knows. The server has a similar problem, it knows to whom it has granted a subscription, but not if this client has received the notification. Clearly, there are some clients in a limbo between subscribed and not subscribed. To solve this, we introduce a new set `confirmed_subscribers`; this are all clients that have received their success notification. The `subscribers` set contains all clients the server has accepted. So `confirmed_subscribers` is a subset of `subscribers`. From now on a client must be a confirmed subscriber to send a message, i.e. it must know that it is subscribed. To receive messages it is enough to be in the `subscribers` set. This means it is possible that a client receives the first message from the chat room before it receives the success message.

The model contains the following events:

ClientSend (refines ClientSend)
: The guard of this event now checks if the client is in `confirmed_subscribers`. This is fine because it strengthens the guard.

ClientReceive (unchanged)

AllClientsReceived (unchanged)

ServerReply (unchanged)

Subscribe (new)
: This newly introduced event represents the client sending a subscribe request to the server. The request contains the user name the client wants to use.

SubscribeSuccess (refines SubscribeSuccess)
: This event is now included in the server. When the server receives a subscribe request, it checks if the user name is allowed. It might also do some other checks that are modeled as a non deterministic choice. If the client is accepted this event is executed and a success message is sent to the client.

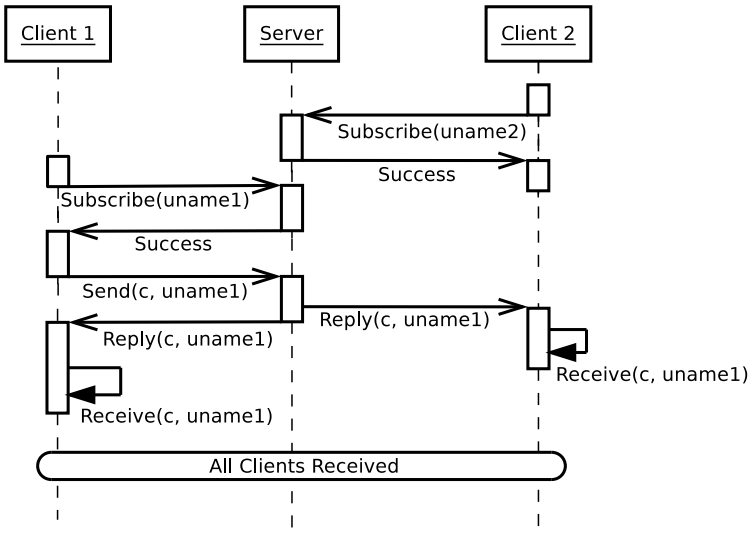

Figure 6.5.: Simulation of chat server model 3

SubscribeFail (refines SubscribeFail)
: This is the counter part of the previous message. If the check failed for any reason, it is not added to the `subscribers` set, and a failure message is sent to the client. In a real system, this would contain some reason why the request was denied.

ReceiveSubResponseSuccess (new)
: This is another new event. It is executed when a client receives its success message. The client is added to `confirmed_subscribers` by this event.

ReceiveSubResponseFail (new)
: Symmetrically this event is used by a client to read a fail message. It does nothing except for removing the message from the mailbox.

The execution of this model for two clients is shown in Figure 6.5. Both clients subscribe successfully and then a message is sent through the system.

The newly satisfied requirements are FUN-3 and FUN-5.

A client needs to be authenticated before it can subscribe.	FUN-3

This is done by the non deterministic choice between the events SubscribeSuccess and SubscribeFail.

If a subscription fails, the clients gets notified.	FUN-5

If the authentication fails, the client gets a fail message generated by SubscribeFail.

6.9. Fourth Refinement

This refinement introduces sessions. They are used by the clients to send messages to the server. Only a client who subscribed successfully gets assigned a session. Clients are no longer allowed to transmit a `Send` message to the server. Instead, they must use their sessions. Therefore sessions can be used to ensure that only subscribed clients can send messages to the chat room. A client without a session cannot send anything to the chat room.

In this refinement we change the events for successful subscription and the sending of messages. The globally accessible sets for user names and subscribers are now changed to private state variables of the server. New state variables for the session mailboxes and the session states are included. Each session stores the address of its client and the client's user name. Another state variable is added to the clients. They are now able to store the address of their sessions. This replaces the global confirmed subscribers set. The events of this model are the following.

SessionForwardSend (refines ClientSend)
: The ClientSend event writes to the mailbox of the server. This means, its refined event must do the same. The client no longer sends a message directly to the server, this event therefore can no longer belong to the client; instead it is now part of the session. The session performs the same write operations the client performed previously. This event is triggered if a session receives a message from its client. The message content together with the clients user name are sent to the server.

ClientSendViaSession (new)
: This event now represents the client sending a message. It sends the message to the corresponding session. Instead of checking if the client is in `confirmed_subscribers`, we now check if the client has stored a session address.

ClientReceive (unchanged)

AllClientsReceived (refines AllClientsReceived)
: Adapted for the name change of the `subscribers` set. We still allow this event to access the set, even though it does not belong to the server. It is only an observation used for the verification and not part of any actor.

ServerReply (refines ServerReply)
: The server now uses its private subscribers set to send the message.

Subscribe (unchanged)

SubscribeSuccess (refines SubscribeSuccess)
: When a subscription is successful, a new session is created. The state of the session is initialized with the client's address and user name. This new session is added to the server's session set. The success message sent to the client now also contains the address of the new session.

SubscribeFail (unchanged)

ReceiveSubResponseSuccess (refines ReceiveSubResponseSuccess)
: A client who receives a subscription success message, now stores the session address it contains.

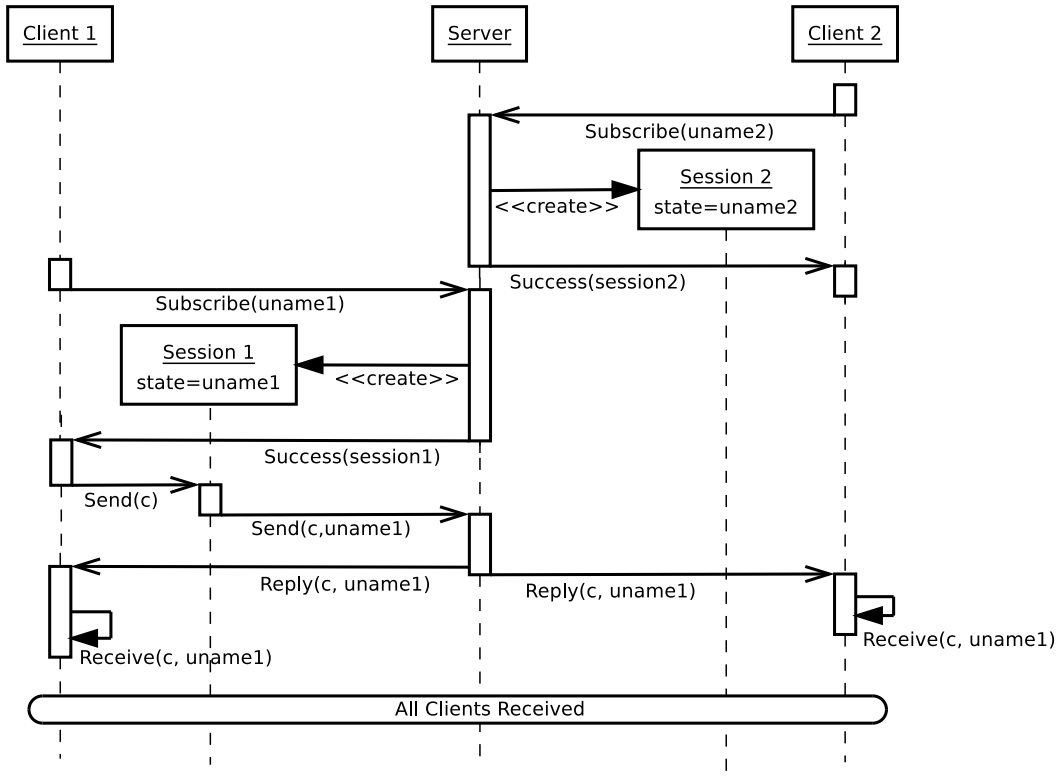

Figure 6.6.: Simulation of chat server model 4

ReceiveSubResponseFail (unchanged)

The same execution with two clients as before is shown in Figure 6.6. This now includes the creation and usage of the session actors.

Although we have introduced sessions, they are not yet used for both directions of the communication. The requirement FUN-4 is not satisfied.

We have eliminated the global sets for subscribers and confirmed subscribers. They have been replaced by private states inside the server and the clients. All communication is now done by passing messages. The system now fully confirms to requirement FUN-9.

The whole system is an actor system.	FUN-9

6.10. Fifth Refinement Version A

This is a first version of the final refinement. The resulting model is more complicated than necessary. An early variant also contained a subtle liveness bug. We decided to include this model anyway, because it provides insides about suboptimal modeling decisions. It

also serves as evidence that the correct refinement is not always obvious. The next section (6.11) contains the final and correct version of this model.

In this refinement we want to use the sessions also to deliver messages from the server to the client. The difficulty in this refinement is that an event that was previously atomic is now performed by multiple events. Some other events can be interleaved with these events. Sending a message from the server to all clients was done as a single event in the earlier models. In this model the server sends the message to all sessions at once, but they forward them one at a time. Instead of one event, there is now one event for the server and one for each session.

To solve this, we introduce a new observation event `AllMessagesDelivered`. This is triggered when all messages haven been forwarded by the sessions. The observation `AllClientsReceived` is still triggered when all clients received the message. The following events are part of this model.

ClientSendViaSession (unchanged)
SessionForwardSend (unchanged)
ClientReceive (unchanged)
AllClientsReceived (unchanged)
ServerReplyToSessions (new)
: This newly introduced event represents the server sending the `Reply` message to the sessions of all subscribers.

SessionForwardReply (new)
: If a session gets a `Reply` message, it forwards the same message to its client.

AllMessagesDelivered (refines ServerReply)
: The new observation event refines the previous ServerReply event. It is triggered after a message was sent by the server to all sessions and they have all forwarded it. This is the state where the message is in all client mailboxes.
We follow the same strategy used in the first refinement for ClientsReceive. This was split into a ClientReceive event and a AllClientsReceived observation.

Subscribe (unchanged)
SubscribeSuccess (unchanged)
SubscribeFail (unchanged)
ReceiveSubResponseSuccess (unchanged)
ReceiveSubResponseFail (unchanged)

The first problem in this model was a liveness bug in the `AllMessagesDelivered` event. If a client subscribes between `ServerReplyToSessions` and `AllMessagesDelivered`, the later would never become enabled. This was because we initially used the server's subscribers set to check if all client mailboxes contain the message. In this scenario the set would have a new member that will never receive the message. This can be solved by keeping a copy of the subscribers set at the time of `ServerReplyToSessions` and use this to determine if all messages have been delivered. All proofs were completed successfully, even through the model was faulty. This highlights the importance of intensive testing of liveness properties. A guard can become too strong if some specific course of events happens. However, this is not part of the refinement proof and needs to be verified separately. See also Section 6.12.

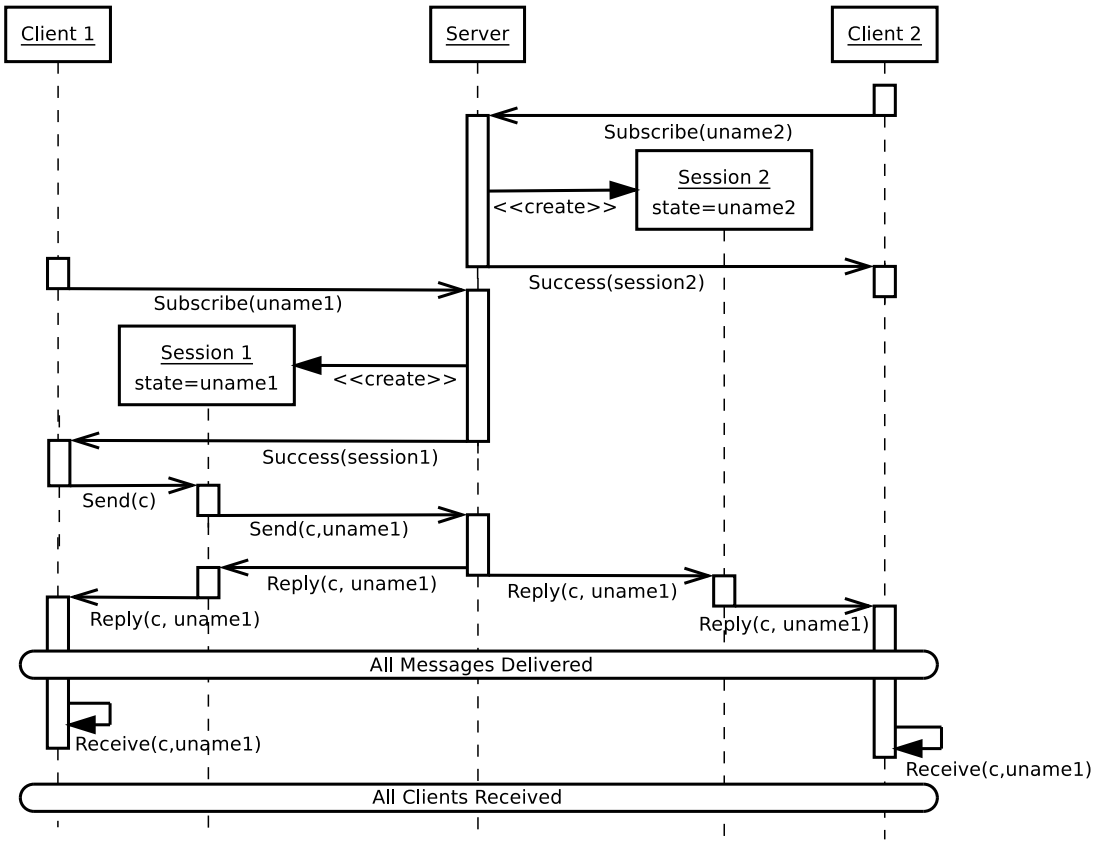

Figure 6.7.: Simulation of chat server model 5a

The major problem of this model is something else. It requires that a message is forwarded by all sessions, before any client can read it. This introduce an arbitrary synchronization barrier. An invisible force prevents clients from reading a message in their mailbox. In an actor system there should be no way for a client to directly observe the mailbox of other clients. This is violated by this version of the model. Figure 6.7 contains a simulation of this model. It includes the synchronization barrier, or observation, `AllMessagesDelivered`.

6.11. Fifth Refinement Version B

Synchronizing all clients by an invisible force is clearly unsatisfactory. This led to the creation of this second version of the final refinement. The goal was to get rid of the `AllMessagesDelivered` observation. Instead, the only observation in the system should be `AllClientsReceived`. Every actor must be able to read any message in its mailbox at any time. The state of other actors may not influence this.

We achieved this by declaring that the sessions refine the clients, and by introducing new mailboxes for the clients. The client mailboxes in this model are unrelated to the client mailboxes in model 4 (Section 6.9).

However, we cannot add an invariant to state that the session mailboxes and the old client mailboxes are equal. Clients and sessions use different identifier types. There exists a one to one correspondence between (subscribed) clients and sessions. This is captured in the state of the session, each session knows its client. The function `session_state_client` is bijective. It can be used to translate between sessions and clients. Using this insight, we added the following invariants.

inv5_6: $\forall s, i \cdot s \mapsto i \in dom(session\_mail\_msgAS\_content) \Rightarrow$
$session\_state\_client(s) \mapsto i \in dom(client\_mail\_msgA\_content)$

inv5_7: $\forall c, i \cdot c \mapsto i \in dom(client\_mail\_msgA\_content) \Rightarrow$
$session\_state\_client^{-1}(c) \mapsto i \in dom(session\_mail\_msgAS\_content)$

inv5_8: $\forall s, i \cdot s \mapsto i \in dom(session\_mail\_msgAS\_content) \Rightarrow$
$session\_mail\_msgAS\_content(s \mapsto i) = client\_mail\_msgA\_content(session\_state\_client(s) \mapsto i)$

They state that the content of old client mailboxes is equivalent to the new session mailboxes if we translate the actor identifiers with `session_state_client`. The mailboxes are not equivalent, but isomorphic.

The events where adapted to this state change as follows.

ClientSendViaSession (unchanged)
SessionForwardSend (unchanged)
SessionForwardReply (refines ClientReceive)
: This event removes a message from the session mailbox and forwards it to the new client mailbox. It is a valid refinement because the message gets removed from the corresponding mailbox. The event does everything ClientReceive did and something more.

ClientReceiveFromSession (new)
: The new receive event for the client. It reads a message from the new client mailbox. Again we assume that the message is displayed to some user.

AllClientsReceived (refines AllClientsReceived)
: This observation checks that the message is no longer in any client's mailbox or in any session's mailbox.

ServerReplyToSessions (refines ServerReply)
: The server now sends the `Reply` message to the sessions instead of the clients.

Subscribe (unchanged)
SubscribeSuccess (unchanged)
SubscribeFail (unchanged)
ReceiveSubResponseSuccess (unchanged)
ReceiveSubResponseFail (unchanged)

Figure 6.8 visualizes the execution of our final model. As before, two clients subscribe and one message is sent.

By using sessions for both directions of client server communication, this model satisfies the last requirement.

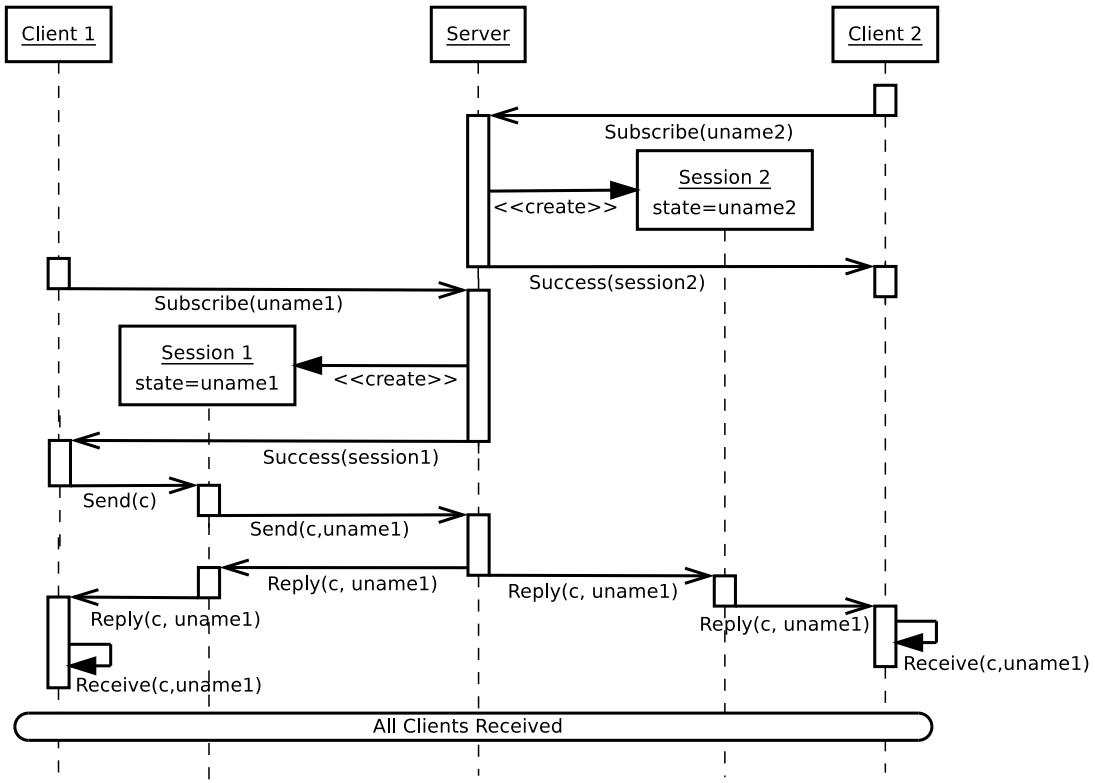

Figure 6.8.: Simulation of chat server model 5b

The communication between subscribed clients and the server is done via sessions.	FUN-8

The complete code of the final machine is included in Appendix B.

6.12. Simulation and Model Checking

For testing and model checking this chat server, we are interested in two liveness properties; one about the subscription process and the other about the sending of messages.

When a client sends a subscription request, it must get an answer from the server. The formal proof shows that the messages are sent correctly. It does not guarantee that the messages are not stalled indefinitely. This can be expressed in LTL as in Equation (6.1). A subscribe request must always be followed by either a success or fail message.

$$\mathbf{G}([Subcribe] \Rightarrow \mathbf{F}([ReceiveSubResponseSuccess] \vee [ReceiveSubResponseFail]) \quad (6.1)$$

We have not included event parameters. This would make the property slightly better, but still would not fully describe the desired behavior. A client could send multiple requests with the same user name and would need to receive the same number of responses. LTL is not able to count, therefore this property cannot be expressed in it.

Another complication are the message identifiers. They are reused while the system is running and are chosen non deterministically at every step. The request and the response can have different message identifiers. We would need to existentially quantify over the identifiers. This is not supported by ProB.

For this reasons, we only use the simpler version without any parameters. It will not find all problems, but can still increase our confidence in the model.

The other property we use is shown in Equation (6.2). It describes the process of sending messages. If a client sends a message at some later point, the `AllClientsReceived` observation must be triggered. Again, we did not include any parameters in this property.

$$\mathbf{G}([ClientSendViaSession] \Rightarrow \mathbf{F}[AllClientsReceived]) \tag{6.2}$$

Using ProB on both of these properties gives us a counterexample. This is to be expected, as we did not include any fairness assumptions. For Equation (6.2) ProB provides us with the following counterexample; we reduced the number of steps and summarized multiple steps.

1 A client successfully subscribes to the server.
2 The same client sends a message to its session.
3 Someone sends a subscribe request.
4 The server rejects it and the answer is received.
5 GOTO 3

The session never gets to forward the message and therefore it will never reach all clients.

The counterexample for Equation (6.1) is similar. Only this time, sending messages is used to block the answer to the subscribe request.

For a finite model these properties are also violated if we assume weak fairness. There is only a finite number of message identifiers. If the server gets flooded with requests, at some point no new identifier is available. This disables all events that want to send a message to the server. They get enabled again, after the server processes one message. Strong fairness would be strong enough, the event is enabled infinitely often, but not always. In the formal model the number of identifiers is infinite. In that case weak and strong fairness are equivalent. ProB on the other hand can only handle a finite model.

Checking the properties under the assumption of strong fairness did not uncover any counterexamples. We had to stop the model checker after it had exhausted all the memory (16GB) of the used machine. So, we have no proof that the property holds. If a counterexample was found, this happened after a few seconds. While the model checker was not able to exhaustively explore the model, it still greatly increases our confidence in the model.

We also used the simulation tool to manually test different corner cases. The bug described in Section 6.11 was found by manual testing.

6.13. Conclusion

In this chapter we developed a chat server using actors. The model starts with a minimal client server system. Each refinement adds more features. The final version supports client authentication and uses sessions to buffer messages between clients and the server.

Figure 6.9 shows the events of all models and how they relate to each other. An arrow from event a to event b means that a is a refinement of b.

For this model no manual proofs where required. The difficulty was in finding the right gluing invariants. These are often not obvious. In some cases, as described in Section 6.10, it is necessary to completely change the meaning of one variable in a refinement step.

This project also shows how important it is to test a model, even if there is a formal proof. Some properties are not covered by the proof and need to be ensured by other means, such as model checking or testing.

By doing this project, we demonstrated that our combination of actor systems and Event-B can be used to model reactive systems resembling real-world servers.

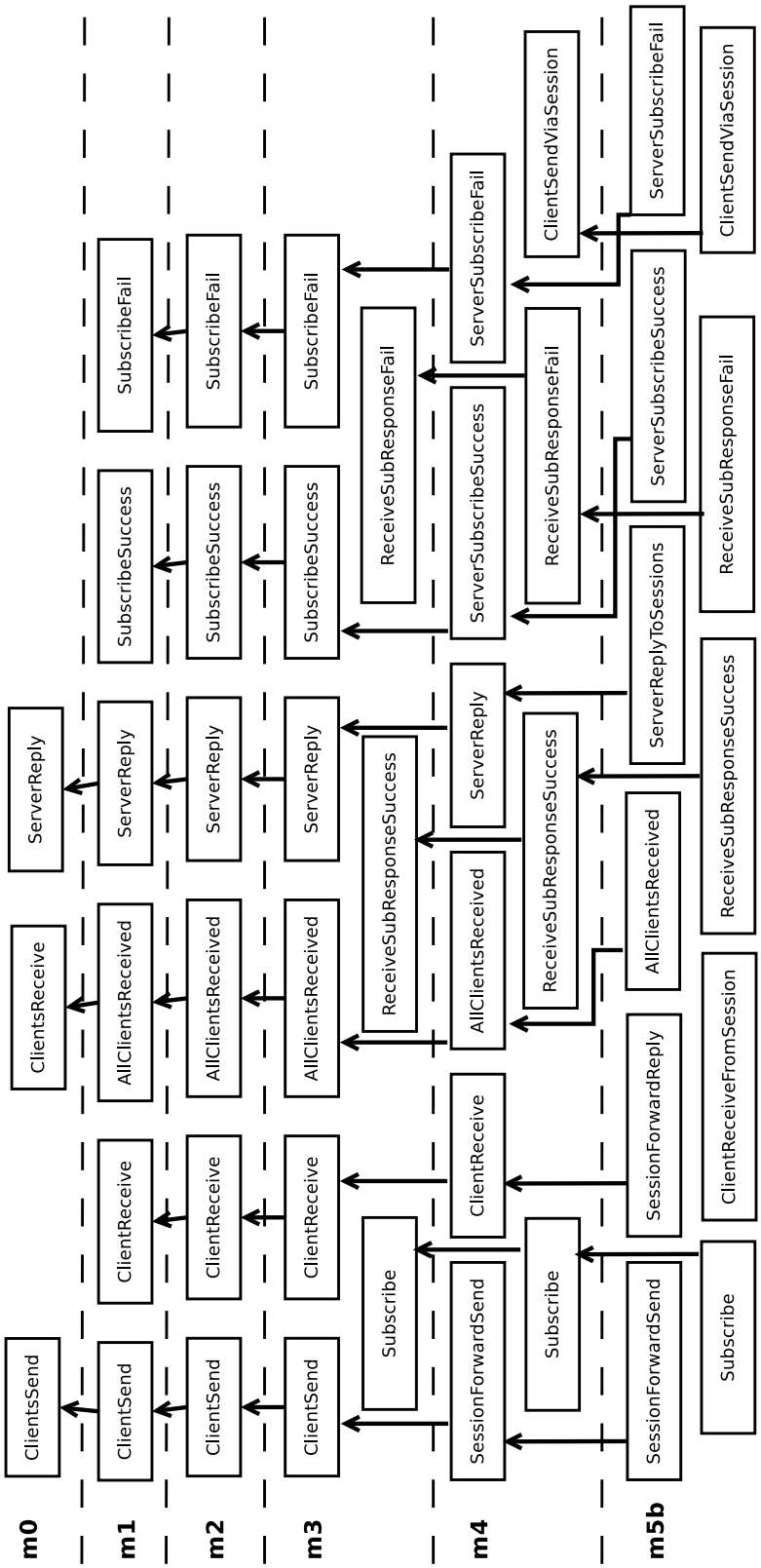

Figure 6.9.: Event refinement for the chat server models

7. Conclusion

7.1. Summary

This thesis presented techniques to model and verify actor systems using Event-B. We studied and compared different encodings of actor systems into Event-B. The used encoding supports the two major distinctive features of actor systems: creating actors at runtime and dynamic network topology. Our techniques allow systems with arbitrary many actors that can be spawned at runtime. An actor can send messages to any other actor if it knows the address. Messages can be used to pass addresses to other actors, this gives us a dynamic topology. The actors and messages are represented as state variables. Sending and receiving messages is done by events where each event is associated with one actor. Based on these encodings we defined comprehensive criteria to decide if an Even-B machine corresponds to an actor system.

The Event-B method is used to develop actor system models via stepwise refinement. We presented two strategies for this, a vertical and a horizontal refinement strategy. Each of them is used in a case study. To demonstrate vertical refinement, we developed an actor system to compute the factorial function. Starting from a mathematical specification, the model is transformed into an actor system by multiple refinements. Two versions of this case study where done, a sequential version that only models a single execution and a concurrent version that can do multiple computations at the same time. A formal termination proof was done for the sequential case.

As a case study for horizontal refinement, we modeled a chat server. In this case the initial model is an actor system that describes the minimal interaction between a server and clients. Each refinement then adds new features to the system. In the final machine there are multiple clients, an asynchronous subscription process, and dynamically created sessions that are used as message buffers.

The required proofs of the refinement relations where done using Rodin's interactive theorem prover and automatic solver plugins. Most proofs can be automated once the correct invariants have been discovered and included in the model. Additionally, we used testing and model checking for liveness properties that where not part of the formal proofs.

7.2. Related Work

Type Systems

The programming language community has developed multiple type systems for message passing systems. These are known as session types [58, 98]. Type systems can be seen as a formal method that is directly embedded into a programming language. They can also state interesting properties about programs similar to the ones that can be expressed in other formal methods. Type systems have been used in conjunction with actors. Charalambides, Dinges, and Agha [36] apply session types to actor systems. This has been extended to also prove liveness properties of actor systems [37]. Type systems for actors are also available as features for modern programming languages. Akka typed [70] is a Scala library that uses the type system to ensure that only messages can be sent if the recipient is equipped to handle them. Type systems are part of a program and provide one specification for the system. Our method on the other hand uses refinement to develop a model in multiple steps. The specification is built gradually and the model is separate from a possible program.

Modeling Languages and Model Checking

Sirjani et al. have developed the modeling language and model checker Rebeca [90, 88, 87] for actor systems. Rebeca (Reactive Objects Language) is as the name suggests based on the reactive object type of actor systems. It consists of a modeling language and a model checker specialized on actor systems. They implemented several optimizations such as compositional verification [89], symmetry and partial order reductions [61]. There is also a version for real time systems [83]. The major difference to our work is that Rebeca is a model checker while we use interactive theorem proving. Our technique can handle dynamic creation of actors and dynamic topology – these are not supported by Rebeca.

Another actor modeling language is ABS [62]. It is an executable and formally specified language based on the active object variant of actor systems. ABS has been used in large industrial case studies [8]. Although the backend includes a model checker, the authors claim that the state space for non-trivial systems becomes too large to handle [62]. This is consistent with our experience using ProB for our Event-B models. Also ABS, does not contain an interactive theorem prover; simulations and static analysis techniques [8] are used instead.

Interactive Theorem Proving

Interactive theorem proving has been used to verify actor systems before. Musser and Varela [76] developed an actor theory in the Athena proof assistant. They give an algebraic perspective of actors, that is abstract enough to apply to multiple actor implementations. Using Athena, they proved properties about actor systems like uniqueness of addresses or fairness. Their theory supports the creation of actors and the exchange of actor identifiers.

Another implementation of actor systems was done in the Coq [38] proof assistant by Yasutake and Watanabe [96]. They also implemented Agha's factorial example [6]. Their system can export Erlang code and they proved uniqueness for their address generation and fairness. Both of these works use correctness properties state as theorems. They do not use stepwise refinement or any other iterative process to develop the final program from the specification.

In contrast to these works, we did not prove uniqueness of identifiers. Instead, we asserted this by using event guards assuming that this is handled correctly by the actor implementation. Our goal was to verify models of actor systems, while these works have a stronger focus on proofing the actor system's infrastructure.

Stepwise Refinement of CSP

As far as we know, actor systems have not been previously studied in Event-B. However, Butler [30, 31, 28] has studied stepwise refinement of Hoare's communicating sequential processes (CSP) [57]. CSPs use synchronized channels between nodes, instead of asynchronous communication and actor addresses. Butler extended action systems [14] to support CSP [30]. He uses a medium to decouple sender and receiver node. This uses a data structure that is similar to our mailbox state variables. A message passing case study is described. It allows one node to send a message to another node in the network. The message needs to be forwarded by multiple nodes in between. The stepwise refinement process introduces the intermediate nodes in the first refinement and later refinements are used to decompose the system. In the last system each node and each channel is represented as its own action system. They communicate via synchronized actions. This works because the number of nodes and channels is finite and known at initialization. No new nodes or channels are created during execution.

Butler later applied similar concepts to B and Event-B. Butler and Leuschel [31] extended ProB to support CSP. This work supports dynamic creation of nodes but does not use refinement. Instead, they use model checking to verify the system. Butler also published lecture notes [28] that contain the theory of his earlier work [30] adapted to Event-B. This work uses a different example but the techniques are mostly the same. There are only two agents, or nodes, that exchange a large block of data by sending multiple messages. Asynchronous communication is again achieved by using a middleware (medium) between the two nodes. The refinement strategy is to decompose the model into three machines: One for each of the two agents and one for the middleware.

Our work differs from these in the choice of the refinement strategies and the focus on actor creation and dynamic network topology. The difference between CSP and actors is less pronounced. Butler uses a medium to buffer messages that are in transit. We directly send messages to the mailbox of the receiver where they are buffered as well. In Event-B both techniques use state variables to hold messages that have been sent but not yet received. The difference is in the number of supported actors, or nodes, and if and how new connections can be established. We use different refinement strategies. Ours focuses on

adding functional requirements to our models. Butler on the other hand uses refinements to decompose the model into one machine for each agent. This decomposition is not possible for our models. It would require an unknown and possible infinite number of machines, where some of them are even dynamically created. Passing actor addresses poses another problem for decomposition as there are no defined channels between two actors.

7.3. Discussion

As far as we know, this work contains the first implementation of actor systems in Event-B. We support creation of actors and a dynamic network topology.

Different encodings for actor systems in Event-B have been explored. Depending on the modeled program, one might be more suitable than another. If an actor is not required to buffer messages in its mailbox, a simpler oneplace mailbox can be used. We presented two choices for handling messages with the same content in a mailbox: via a deterministic counter or by nondeterministically choosing a fresh identifier. Both are viable; the counter is to be preferred if actors need to be linked to a non-actor structure by gluing invariants. Otherwise the abstract identifiers should be favored, as they can be simulated slightly better. We defined a naming schema for all state variables that are part of an actor and rules that shall be obeyed by actor events. These can be used to decide if a machine is an actor system and to map a model to an implementation in a programming language.

Each of the two refinement strategies was used in its own case study.

The vertical refinement case study factorial demonstrates that stepwise refinement can be used to derive an actor program from a mathematical specification. This also shows that models need to be iteratively developed. Not only in the sense of stepwise refinement, but while modeling new insights are gained and they can be used to improve the model in ways that are not compatible with refinement. It is often advantageous to start with a simpler model and later rewrite it in a more complex way when a better understanding of the problem has been reached. This was done with the sequential and concurrent versions of the factorial case study.

It is also important to decide which properties one wants to include in a formal proof and what assumptions are used. We have decided to not include liveness properties. Our models also rely on various assumptions about the environment. Messages will be delivered exactly once, they can neither get lost nor become duplicated. A new actor with a unique identifier can always be created. There are also requirements how events may access the state variables, that if violated would invalidate the proofs.

It is important to be aware of these limitations of the proofs. Where possible other techniques should be used to asses correctness properties that have not been proven. We used simulation and model checking to check liveness properties of our system. When using a model to make claims about a real system, one needs to be aware of all the limitations. For example, our models do not apply to distributed systems with message loss.

In conclusion, the main goals of this thesis have been met.

- We developed encodings of actor systems in Event-B.
- A vertical refinement approach was successfully used to develop a factorial algorithm.
- Using horizontal refinement a chat server with subscribe and sessions was implemented.
- Liveness properties have been excluded from formal proofs, but have been verified by model checking and testing.

7.4. Future Work

The work in this thesis could be extended in various ways.

A weakness of the used technique is that models can become very convoluted. At times it becomes difficult to keep track of all the state variables. One could develop higher level constructs to describe actors and messages. A domain specific language could be used to define messages and actors. This would include type definitions for messages and actor states. A special syntax to send messages similar to the exclamation mark in Erlang could be included. The actual Event-B machine would be generated from this higher level description.

A similar, but orthogonal extension would be a Rodin plugin that checks if a machine is an actor system. In Section 4.1, we defined criteria a machine needs to fulfill to correspond to an actor system. The plugin could check if all state variables and events correspond to this definition. The most important things to check would be that an event only accesses information it is allowed to, removes a message after reading it, and only writes to mailboxes it legitimately knows the address of. The plugin would need some escape hatch because in some cases the border is not clear. Some variables and events are required for the proof, but they do not influence the actual computation. It would probably be impossible to detect these situations automatically.

Many actor systems have temporal correctness properties that we did not include in the Event-B model. Instead, we relied on testing and model checking finite versions of the model. The Event-B models could be extended to include trace variables that store all events that have been executed. Liveness or other temporal properties could then be expressed as invariants over these trace variables. Another way to prove liveness properties of Event-B models was presented by Hoang and Abrial [56]. It introduces new proof rules to reason about liveness properties. One could try to apply these to actor system models.

Distributed systems often operate in uncertain environments. Our model of actor systems could be extended to support message loss and message duplication. This would allow us to model robust actor systems that use strategies to cope with uncertain message delivery.

It would also be interesting to apply the technique from this work to more and larger case studies. A possible target would be to model (a subset) of the MQTT protocol [19]. This is

somewhat similar to the chat server case study from Chapter 6, but would need to handle a lot more ways of interaction.

Bibliography

[1] Jean-Raymond Abrial. "Event Model Decomposition." In: *Technical report ETH, Department of Computer Science* 626 (Apr. 2009).

[2] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. New York, NY, USA: Cambridge University Press, 2010.

[3] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge ; New York: Cambridge University Press, 1996.

[4] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. "Rodin: An Open Toolset for Modelling and Reasoning in Event-B." In: *International Journal on Software Tools for Technology Transfer* 12.6 (2010), pp. 447–466.

[5] Jean-Raymond Abrial and Stefan Hallerstede. "Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B." In: *Fundamenta Informaticae* 77.1-2 (2007), pp. 1–28. URL: `http://content.iospress.com/articles/fundamenta-informaticae/fi77-1-2-02` (visited on Dec. 10, 2019).

[6] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[7] Gul Agha and Carl Hewitt. "Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism." In: *Foundations of Software Technology and Theoretical Computer Science, Fifth Conference, New Delhi, India, December 16-18, 1985, Proceedings*. Ed. by S. N. Maheshwari. Vol. 206. Lecture Notes in Computer Science. Springer, 1985, pp. 19–41.

[8] Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa, and Peter Y. H. Wong. "Formal Modeling and Analysis of Resource Management for Cloud Architectures: An Industrial Case Study Using Real-Time ABS." In: *Service Oriented Computing and Applications* 8.4 (2014), pp. 323–339.

[9] Joe Armstrong. "A History of Erlang." In: *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007* (San Diego, California). Ed. by Barbara G. Ryder and Brent Hailpern. ACM, 2007, pp. 1–26.

[10] Joe Armstrong. "Making Reliable Distributed Systems in the Presence of Software Errors." PhD Thesis. Royal Institute of Technology, 2003.

[11] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Ed. by Susannah Davidson Pfalzer. Second edition. Pragmatic Programmers. Dallas, Texas ; Raleigh, North Carolina: The Pragmatic Bookshelf, 2013.

[12] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1993.

[13] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, 2000.

[14] Ralph-Johan Back and Reino Kurki-Suonio. "Decentralization of Process Nets with Centralized Control." In: *Distributed Computing* 3.2 (1989), pp. 73–87.

[15] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[16] Ralph-Johan Back and Joakim von Wright. "Trace Refinement of Action Systems." In: *CONCUR '94, Concurrency Theory, 5th International Conference, Uppsala, Sweden, August 22-25, 1994, Proceedings*. Ed. by Bengt Jonsson and Joachim Parrow. Vol. 836. Lecture Notes in Computer Science. Springer, 1994, pp. 367–384.

[17] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT press, 2008.

[18] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. "SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft." In: *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings*. Ed. by Eerke A. Boiten, John Derrick, and Graeme Smith. Vol. 2999. Lecture Notes in Computer Science. Springer, 2004, pp. 1–20.

[19] Andrew Banks and Rahul Gupta. "MQTT Version 3.1. 1." In: *OASIS standard* 29 (2014), p. 89.

[20] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4." In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177.

[21] Clark W. Barrett and Cesare Tinelli. "CVC3." In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 298–302.

[22] Clark Barrett, Aaron Stump, and Cesare Tinelli. *The SMT-LIB Standard–Version 2.0*. 2010.

[23] Jens Bendisposto, Fabian Fritz, Michael Jastram, Michael Leuschel, and Ingo Weigelt. "Developing Camille, a Text Editor for Rodin." In: *Software: Practice and Experience* 41.2 (2011), pp. 189–198.

[24] Johan Bevemyr. "How Cisco Is Using Erlang for Intent-Based Networking." Code Beam (Stockholm). June 1, 2018. URL: `https://youtu.be/077-XJv6PLQ` (visited on Nov. 29, 2019).

[25] Hans-Juergen Boehm and Sarita V. Adve. “Foundations of the C++ Concurrency Memory Model.” In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008* (Tucson, AZ, USA). Ed. by Rajiv Gupta and Saman P. Amarasinghe. ACM, 2008, pp. 68–78.

[26] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003. URL: `http://www.springer.com/computer/swe/book/978-3-540-00702-9` (visited on Dec. 10, 2019).

[27] Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. “veriT: An Open, Trustable and Efficient SMT-Solver.” In: *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*. Ed. by Renate A. Schmidt. Vol. 5663. Lecture Notes in Computer Science. Springer, 2009, pp. 151–156.

[28] Michael Butler. “Incremental Design of Distributed Systems with Event-B.” In: *Engineering Methods and Tools for Software Safety and Security* 22.131 (2009).

[29] Michael J. Butler. “Decomposition Structures for Event-B.” In: *Integrated Formal Methods, 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings*. Ed. by Michael Leuschel and Heike Wehrheim. Vol. 5423. Lecture Notes in Computer Science. Springer, 2009, pp. 20–38.

[30] Michael J. Butler. “Stepwise Refinement of Communicating Systems.” In: *Science of Computer Programming* 27.2 (1996), pp. 139–173.

[31] Michael J. Butler and Michael Leuschel. “Combining CSP and B for Specification and Property Verification.” In: *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*. Ed. by John S. Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki. Vol. 3582. Lecture Notes in Computer Science. Springer, 2005, pp. 221–236.

[32] Michael J. Butler and Issam Maamria. “Practical Theory Extension in Event-B.” In: *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*. Ed. by Zhiming Liu, Jim Woodcock, and Huibiao Zhu. Vol. 8051. Lecture Notes in Computer Science. Springer, 2013, pp. 67–81.

[33] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. “Orleans: Cloud Computing for Everyone.” In: *ACM Symposium on Cloud Computing in Conjunction with SOSP 2011, SOCC ’11, Cascais, Portugal, October 26-28, 2011*. Ed. by Jeffrey S. Chase and Amr El Abbadi. ACM, 2011, p. 16.

[34] Denis Caromel, Ludovic Henrio, and Bernard P. Serpette. “Asynchronous Sequential Processes.” In: *Information and Computation* 207.4 (2009), pp. 459–495.

[35] Francesco Cesarini. *Which Companies Are Using Erlang, and Why?* Sept. 11, 2019. URL: `https://www.erlang-solutions.com/blog/which-companies-are-using-erlang-and-why-mytopdogstatus.html` (visited on Sept. 14, 2019).

[36] Minas Charalambides, Peter Dinges, and Gul Agha. "Parameterized Concurrent Multi-Party Session Types." In: *Proceedings 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, FOCLASA 2012, Newcastle, U.K., September 8, 2012*. Ed. by Natallia Kokash and António Ravara. Vol. 91. EPTCS. 2012, pp. 16–30.

[37] Minas Charalambides, Karl Palmskog, and Gul Agha. "Types for Progress in Actor Programs." In: *Models, Languages, and Tools for Concurrent and Distributed Programming - Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday*. Ed. by Michele Boreale, Flavio Corradini, Michele Loreti, and Rosario Pugliese. Vol. 11665. Lecture Notes in Computer Science. Springer, 2019, pp. 315–339.

[38] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. Cambridge, MA: The MIT Press, 2013. URL: `http://adam.chlipala.net/cpdt/`.

[39] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model Checking*. MIT press, 2018.

[40] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. "Symmetry Reductions in Model Checking." In: *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*. Ed. by Alan J. Hu and Moshe Y. Vardi. Vol. 1427. Lecture Notes in Computer Science. Springer, 1998, pp. 147–158.

[41] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer, 2018.

[42] Clearsy. *Atelier B*. 2016. URL: `https://www.atelierb.eu/en/atelier-b-tools/`.

[43] Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. "Orca: GC and Type System Co-Design for Actor Languages." In: *Proceedings of the ACM on Programming Languages* 1 (OOPSLA 2017), 72:1–72:28.

[44] 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, Amsterdam, The Netherlands, October 30, 2016* (Amsterdam, Netherlands). Ed. by Sylvan Clebsch, Travis Desell, Philipp Haller, and Alessandro Ricci. ACM, 2016, pp. 31–40.

[45] Leonardo Mendonça De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.

[46] Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. "A Survey of Active Object Languages." In: *ACM Computing Surveys* 50.5 (2017), 76:1–76:39.

[47] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. "SMT Solvers for Rodin." In: *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings.* Ed. by John Derrick, John S. Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene. Vol. 7316. Lecture Notes in Computer Science. Springer, 2012, pp. 194–207.

[48] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs." In: *Communications of the ACM* 18.8 (1975), pp. 453–457.

[49] Ivaylo Dobrikov, Michael Leuschel, and Daniel Plagge. "LTL Model Checking under Fairness in ProB." In: *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings.* Ed. by Rocco De Nicola and eva Kühn. Vol. 9763. Lecture Notes in Computer Science. Springer, 2016, pp. 204–211.

[50] Ericsson. *Erlang Celebrates 20 Years as Open Source.* May 31, 2018. URL: `https://www.ericsson.com/en/news/2018/5/erlang-celebrates-20-years-as-open-source` (visited on Nov. 29, 2019).

[51] Martin Fowler and Cris Kobryn. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley, 2004.

[52] Carl Hewitt. "Actor Model of Computation: Scalable Robust Information Systems." In: (Aug. 9, 2010). arXiv: `1008.1459 [cs]`. URL: `http://arxiv.org/abs/1008.1459` (visited on Aug. 31, 2019).

[53] Carl Hewitt. "Viewing Control Structures as Patterns of Passing Messages." In: *Artificial Intelligence* 8.3 (1977), pp. 323–364.

[54] Carl Hewitt and Henry G. Baker. "Laws for Communicating Parallel Processes." In: *Information Processing, Proceedings of the 7th IFIP Congress 1977, Toronto, Canada, August 8-12, 1977.* Ed. by Bruce Gilchrist. North-Holland, 1977, pp. 987–992.

[55] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence." In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973.* Ed. by Nils J. Nilsson. William Kaufmann, 1973, pp. 235–245. URL: `http://ijcai.org/Proceedings/73/Papers/027B.pdf` (visited on Dec. 10, 2019).

[56] Thai Son Hoang and Jean-Raymond Abrial. "Reasoning about Liveness Properties in Event-B." In: *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings.* Ed. by Shengchao Qin and Zongyan Qiu. Vol. 6991. Lecture Notes in Computer Science. Springer, 2011, pp. 456–471.

[57] C. A. R. Hoare. "Communicating Sequential Processes." In: *Communications of the ACM* 21.8 (1978), pp. 666–677.

[58] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty Asynchronous Session Types." In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. Ed. by George C. Necula and Philip Wadler. ACM, 2008, pp. 273–284.

[59] Daniel Jackson. "Alloy: A Lightweight Object Modelling Notation." In: *ACM Transactions on Software Engineering and Methodology* 11.2 (2002), pp. 256–290.

[60] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT press, 2012.

[61] Mohammad Mahdi Jaghoori, Marjan Sirjani, Mohammad Reza Mousavi, Ehsan Khamespanah, and Ali Movaghar. "Symmetry and Partial Order Reduction Techniques in Model Checking Rebeca." In: *Acta Informatica* 47.1 (2010), pp. 33–66.

[62] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. "ABS: A Core Language for Abstract Behavioral Specification." In: *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*. Ed. by Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue. Vol. 6957. Lecture Notes in Computer Science. Springer, 2010, pp. 142–164.

[63] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. "Creol: A Type-Safe Object-Oriented Model for Distributed Concurrent Systems." In: *Theoretical Computer Science* 365.1-2 (2006), pp. 23–66.

[64] Cliff B. Jones. *Systematic Software Development Using VDM*. Vol. 2. Prentice Hall, 1990.

[65] Severin Kann. "Refinement of Test Models in Event-B Towards Model-Based Testing." Bachelor thesis. Graz University of Technology, 2017.

[66] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston: Addison-Wesley, 2003.

[67] Michael Leuschel and Michael J. Butler. "ProB: A Model Checker for B." In: *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*. Ed. by Keijiro Araki, Stefania Gnesi, and Dino Mandrioli. Vol. 2805. Lecture Notes in Computer Science. Springer, 2003, pp. 855–874.

[68] Michael Leuschel and Michael J. Butler. "ProB: An Automated Analysis Toolset for the B Method." In: *International Journal on Software Tools for Technology Transfer* 10.2 (2008), pp. 185–203.

[69] Lightbend. *Akka Documentation*. 2019. URL: `https://doc.akka.io/docs/akka/2.5/index.html` (visited on June 9, 2019).

[70] Lightbend. *Akka Typed Documentation*. 2019. URL: `https://doc.akka.io/docs/akka/2.5/typed/actors.html` (visited on June 9, 2019).

[71] John McCarthy. “Towards a Mathematical Science of Computation.” In: *Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962*. North-Holland, 1962, pp. 21–28.

[72] Kenneth L. McMillan. “Verification of Infinite State Systems by Compositional Model Checking.” In: *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME ’99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*. Ed. by Laurence Pierre and Thomas Kropf. Vol. 1703. Lecture Notes in Computer Science. Springer, 1999, pp. 219–234.

[73] Cade Metz. “Why WhatsApp Only Needs 50 Engineers for Its 900M Users.” In: *WIRED* (Sept. 15, 2015). URL: `https://www.wired.com/2015/09/whatsapp-serves-900-million-users-50-engineers/` (visited on Nov. 29, 2019).

[74] Mark S. Miller, Eric Dean Tribble, and Jonathan S. Shapiro. “Concurrency Among Strangers.” In: *Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers*. Ed. by Rocco De Nicola and Davide Sangiorgi. Vol. 3705. Lecture Notes in Computer Science. Springer, 2005, pp. 195–229.

[75] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML: Revised*. MIT press, 1997.

[76] David R. Musser and Carlos A. Varela. “Structured Reasoning about Actor Systems.” In: *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!@SPLASH 2013, Indianapolis, IN, USA, October 27-28, 2013*. Ed. by Nadeem Jamali, Alessandro Ricci, Gera Weiss, and Akinori Yonezawa. ACM, 2013, pp. 37–48.

[77] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. “How Amazon Web Services Uses Formal Methods.” In: *Communications of the ACM* 58.4 (2015), pp. 66–73.

[78] Tobias Nipkow and Gerwin Klein. *Concrete Semantics*. New York, NY: Springer, 2014. URL: `http://www.concrete-semantics.org`.

[79] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Vol. 2283. Springer Science & Business Media, 2002.

[80] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 2008.

[81] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[82] Amir Pnueli. “The Temporal Logic of Programs.” In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57.

[83] Arni Hermann Reynisson, Marjan Sirjani, Luca Aceto, Matteo Cimini, Ali Jafari, Anna Ingólfsdóttir, and Steinar Hugi Sigurdarson. "Modelling and Simulation of Asynchronous Real-Time Systems Using Timed Rebeca." In: *Science of Computer Programming* 89 (2014), pp. 41–68.

[84] Ken Robinson. *A Concise Summary of the Event-B Mathematical Toolkit*. 2014. URL: `http://wiki.event-b.org/images/EventB-Summary.pdf` (visited on Nov. 7, 2019).

[85] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

[86] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs." In: *ACM Transactions on Computer Systems* 15.4 (1997), pp. 391–411.

[87] Marjan Sirjani. "Power Is Overrated, Go for Friendliness! Expressiveness, Faithfulness, and Usability in Modeling: The Actor Experience." In: *Principles of Modeling - Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*. Ed. by Marten Lohstroh, Patricia Derler, and Marjan Sirjani. Vol. 10760. Lecture Notes in Computer Science. Springer, 2018, pp. 423–448.

[88] Marjan Sirjani and Mohammad Mahdi Jaghoori. "Ten Years of Analyzing Actors: Rebeca Experience." In: *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*. Ed. by Gul Agha, Olivier Danvy, and José Meseguer. Vol. 7000. Lecture Notes in Computer Science. Springer, 2011, pp. 20–56.

[89] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. "Model Checking, Automated Abstraction, and Compositional Verification of Rebeca Models." In: *Journal of Universal Computer Science* 11.6 (2005), pp. 1054–1082.

[90] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. "Modeling and Verification of Reactive Systems Using Rebeca." In: *Fundamenta Informaticae* 63.4 (2004), pp. 385–410. URL: `http://content.iospress.com/articles/fundamenta-informaticae/fi63-4-05` (visited on Dec. 10, 2019).

[91] Dave Thomas. *Programming Elixir: Functional, Concurrent, Pragmatic, Fun*. Pragmatic Bookshelf, 2018.

[92] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. "AmbientTalk: Object-Oriented Event-Driven Programming in Mobile Ad Hoc Networks." In: *XXVI International Conference of the Chilean Computer Science Society (SCCC 2007), 8-9 November 2007, Iquique, Chile*. IEEE Computer Society, 2007, pp. 3–12.

[93] Carlos A. Varela and Gul Agha. "Programming Dynamically Reconfigurable Open Systems with SALSA." In: *ACM SIGPLAN Notices* 36.12 (2001), pp. 20–34.

[94] WhatsApp. *1 Million Is so 2011*. Jan. 6, 2012. URL: `https://blog.whatsapp.com/196/1-million-is-so-2011` (visited on Nov. 29, 2019).

[95] Divakar Yadav and Michael J. Butler. “Verification of Liveness Properties in Distributed Systems.” In: *Contemporary Computing - Second International Conference, IC3 2009, Noida, India, August 17-19, 2009. Proceedings*. Ed. by Sanjay Ranka, Srinivas Aluru, Rajkumar Buyya, Yeh-Ching Chung, Sumeet Dua, Ananth Grama, Sandeep K. S. Gupta, Rajeev Kumar, and Vir V. Phoha. Vol. 40. Communications in Computer and Information Science. Springer, 2009, pp. 625–636.

[96] Shohei Yasutake and Takuo Watanabe. *Actario: A Framework for Reasoning about Actor Systems*. 2015.

[97] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. “Object-Oriented Concurrent Programming in ABCL/1.” In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’86), Portland, Oregon, USA, Proceedings*. Ed. by Norman K. Meyrowitz. ACM, 1986, pp. 258–268.

[98] Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. “Parameterised Multiparty Session Types.” In: *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by C.-H. Luke Ong. Vol. 6014. Lecture Notes in Computer Science. Springer, 2010, pp. 128–145.

Appendix

The syntax highlighting colors in the appendix are as follows: keywords are dark blue, labels are light blue, comments are green, and code that has not changed in the last step is brown.

Appendix A.

Factorial

A.1. Sequential Model

Contexts

CONTEXT ctx0
CONSTANTS
fact
input
AXIOMS
axm0_0: $fact \in \mathbb{N} \rightarrow \mathbb{N}_1$
axm0_1: $fact(0) = 1$
axm0_2: $\forall n \cdot n \in \mathbb{N} \Rightarrow fact(n+1) = (n+1) * fact(n)$
axm0_3: $input \in \mathbb{N}$
END

CONTEXT ctx3 EXTENDS ctx0
CONSTANTS
invalid_id
final_id
ACTOR_ID
boolToNat
AXIOMS
axm3_0: $ACTOR\_ID = \mathbb{N} \cup \{-1, invalid\_id\}$
axm3_1: $final\_id = -1$
axm3_2: $invalid\_id \notin \mathbb{N}$
axm3_3: $final\_id \neq invalid\_id$
axm3_4: $boolToNat \in BOOL \rightarrow \mathbb{N}$
axm3_5: $boolToNat(TRUE) = 0$
axm3_6: $boolToNat(FALSE) = 1$
END

CONTEXT ctx5 EXTENDS ctx3
CONSTANTS
CONT_ID
AXIOMS
axm5_0: $CONT\_ID = \mathbb{N}$
END

Machines

MACHINE m0
SEES ctx0
VARIABLES
result
INVARIANTS
inv0_0: $result \in \mathbb{N}$
EVENTS
Initialisation
begin
act0_0: $result := 0$
end
Step ⟨anticipated⟩ $\widehat{=}$
begin
skip
end
Finish ⟨ordinary⟩ $\widehat{=}$
begin
act0_0: $result := fact(input)$
end
END

MACHINE m1
REFINES m0
SEES ctx0
VARIABLES
result
tmp_result
val
INVARIANTS
inv1_0: $val \in \mathbb{N}$
inv1_1: $tmp\_result \in \mathbb{N}_1$
inv1_2: $val \leq input$
inv1_3: $tmp\_result = fact(input - val)$
thm_DLF: ⟨theorem⟩ $(val > 0) \lor (val = 0)$
VARIANT

val

EVENTS
Initialisation

begin
act1_0: $result := 0$
act1_1: $val := input$
act1_2: $tmp\_result := 1$
end

ComputeStep ⟨convergent⟩ $\widehat{=}$
refines Step

when
grd1_0: $val > 0$
then
act1_0: $val := val - 1$
act1_1: $tmp\_result := tmp\_result * (input - val + 1)$
end

Finish ⟨ordinary⟩ $\widehat{=}$
refines Finish

when
grd1_0: $val = 0$
then
act1_0: $result := tmp\_result$
end

END

MACHINE m2
REFINES m1
SEES ctx0
VARIABLES

result
tmp_result
counter
stack
stack_pointer

INVARIANTS

inv2_1: $stack \in \mathbb{N} \pfun \mathbb{N}_1$
inv2_2: $stack\_pointer \in \mathbb{N}$
inv2_3: $0 \upto (stack\_pointer - 1) \subseteq dom(stack)$
inv2_4: $counter \in \mathbb{N}$
inv2_5: $\forall n \cdot n \in dom(stack) \Rightarrow stack(n) = input - n$
inv2_6: $stack\_pointer + counter = val$
inv2_7: $counter = 0 \Rightarrow val = stack\_pointer$
inv2_8: $counter \neq 0 \Rightarrow val = input$
thm_DLF: ⟨theorem⟩ $(counter > 0 \land tmp\_result = 1) \lor (counter = 0 \land stack\_pointer > 0) \lor (counter = 0 \land stack\_pointer = 0)$

VARIANT

$counter$

EVENTS
Initialisation

begin
act2_0: $result := 0$
act2_1: $counter := input$
act2_2: $stack := \varnothing$
act2_3: $stack\_pointer := 0$
act2_4: $tmp\_result := 1$
end

Call $\langle$convergent$\rangle \mathrel{\widehat{=}}$
when
grd2_0: $counter > 0$
grd2_1: $tmp\_result = 1$
then
act2_0: $counter := counter - 1$
act2_1: $stack\_pointer := stack\_pointer + 1$
act2_2: $stack(stack\_pointer) := counter$
end

Return $\langle$convergent$\rangle \mathrel{\widehat{=}}$
refines ComputeStep
when
grd2_0: $counter = 0$
grd2_1: $stack\_pointer > 0$
then
act2_0: $tmp\_result := tmp\_result * stack(stack\_pointer - 1)$
act2_1: $stack\_pointer := stack\_pointer - 1$
end

Finish $\langle$ordinary$\rangle \mathrel{\widehat{=}}$
refines Finish
when
grd2_0: $counter = 0$
grd2_1: $stack\_pointer = 0$
then
act2_0: $result := tmp\_result$
end

END

MACHINE m3
REFINES m2
SEES ctx3
VARIABLES
result
counter
active_actor
msg_exists
msg_recipient
msg_content
num_actors
actor_id
cont_actors_target
cont_actors_value
INVARIANTS

inv3_0: $active\_actor \in actor\_id \cup \{final\_id\}$
inv3_1: $msg\_exists \in BOOL$
inv3_2: $msg\_recipient \in actor\_id \cup \{final\_id, invalid\_id\}$
inv3_3: $msg\_content \in \mathbb{N}_1$
inv3_4: $num\_actors \in \mathbb{N}$
inv3_5: $actor\_id = 0 \mathinner{..} (num\_actors - 1)$
inv3_6: $cont\_actors\_target \in actor\_id \rightarrow (actor\_id \cup \{final\_id\})$
inv3_7: $cont\_actors\_value \in actor\_id \rightarrow \mathbb{N}_1$
inv3_8: $msg\_exists = TRUE \Rightarrow counter = 0$
inv3_9: $msg\_exists = FALSE \Leftrightarrow msg\_recipient = invalid\_id$
inv3_10: ⟨theorem⟩ $invalid\_id \notin actor\_id$
inv3_11: $msg\_content = tmp\_result$
inv3_12: $num\_actors = stack\_pointer$
inv3_13: $active\_actor = num\_actors - 1$
inv3_14: $\forall x \cdot x \in dom(cont\_actors\_value) \Rightarrow stack(x) = cont\_actors\_value(x)$
inv3_15: $\forall x \cdot x \in dom(cont\_actors\_target) \Rightarrow cont\_actors\_target(x) = x - 1$
inv3_16: $msg\_exists = TRUE \Rightarrow msg\_recipient = active\_actor$
DLF invariant
inv3_17: $(counter = 0 \land msg\_exists = FALSE) \Rightarrow active\_actor = input - 1$
DLF invariant
thm_DLF: ⟨theorem⟩
$(counter > 0) \lor$
$(counter = 0 \land msg\_exists = FALSE \land active\_actor = input - 1) \lor$
$(msg\_exists = TRUE \land msg\_recipient \neq final\_id \land msg\_recipient = active\_actor) \lor$
$(msg\_exists = TRUE \land msg\_recipient = final\_id \land msg\_recipient = active\_actor)$

VARIANT

$boolToNat(msg\_exists)$

EVENTS
Initialisation

begin
act3_0: $result := 0$
act3_1: $counter := input$
act3_2: $active\_actor := final\_id$
act3_3: $msg\_exists := FALSE$
act3_4: $msg\_recipient := invalid\_id$
act3_5: $msg\_content := 1$
act3_6: $num\_actors := 0$
act3_7: $actor\_id := \varnothing$
act3_8: $cont\_actors\_target := \varnothing$
act3_9: $cont\_actors\_value := \varnothing$
end

Create ⟨convergent⟩ $\widehat{=}$
refines Call

when
grd3_0: $counter > 0$
then
act3_0: $counter := counter - 1$
act3_1: $actor\_id := 0 \mathinner{..} num\_actors$
act3_2: $cont\_actors\_target(active\_actor + 1) := active\_actor$
act3_3: $cont\_actors\_value(active\_actor + 1) := counter$
act3_4: $active\_actor := active\_actor + 1$
act3_5: $num\_actors := num\_actors + 1$
end

Created ⟨convergent⟩ $\widehat{=}$
when
grd3_0: $counter = 0$
grd3_1: $msg\_exists = FALSE$
grd3_2: $active\_actor = input - 1$
then
act3_0: $msg\_exists := TRUE$
act3_1: $msg\_recipient := active\_actor$
act3_2: $msg\_content := 1$
end
Compute ⟨convergent⟩ $\widehat{=}$
refines Return
when
grd3_1: $msg\_exists = TRUE$
grd3_2: $msg\_recipient \neq final\_id$
grd3_3: $msg\_recipient = active\_actor$
then
act3_0: $msg\_recipient := cont\_actors\_target(msg\_recipient)$
act3_1: $msg\_content := msg\_content * cont\_actors\_value(msg\_recipient)$
act3_2: $num\_actors := num\_actors - 1$
act3_3: $actor\_id := 0 \mathinner{..} num\_actors - 2$
act3_4: $cont\_actors\_target := \{msg\_recipient\} \lhd cont\_actors\_target$
act3_5: $cont\_actors\_value := \{msg\_recipient\} \lhd cont\_actors\_value$
act3_6: $active\_actor := cont\_actors\_target(msg\_recipient)$
end
Finish ⟨ordinary⟩ $\widehat{=}$
refines Finish
when
grd3_1: $msg\_exists = TRUE$
grd3_2: $msg\_recipient = final\_id$
grd3_3: $msg\_recipient = active\_actor$
then
act3_0: $result := msg\_content$
end
END

MACHINE m4
REFINES m3
SEES ctx3
VARIABLES
result
active_actor
msg_exists
msg_recipient
msg_content
num_actors
actor_id
cont_actors_target
cont_actors_value
msgC_exists

msgC_content

INVARIANTS

inv4_0: $msgC\_exists \in BOOL$
inv4_1: $msgC\_content \in \mathbb{N}$
inv4_2: $msgC\_content = counter$
inv4_3: $msgC\_exists = FALSE \Rightarrow msg\_exists = TRUE$
DLF invariant
thm_DLF: ⟨theorem⟩
$(msgC\_exists = TRUE \land msgC\_content > 0) \lor$
$(msgC\_exists = TRUE \land msgC\_content = 0 \land msg\_exists = FALSE \land active\_actor = input - 1) \lor$
$(msg\_exists = TRUE \land msg\_recipient \neq final\_id \land msg\_recipient = active\_actor) \lor$
$(msg\_exists = TRUE \land msg\_recipient = final\_id \land msg\_recipient = active\_actor)$

EVENTS
Initialisation

begin
act4_0: $result := 0$
act4_1: $active\_actor := final\_id$
act4_2: $msg\_exists := FALSE$
act4_3: $msg\_recipient := invalid\_id$
act4_4: $msg\_content := 1$
act4_5: $num\_actors := 0$
act4_6: $actor\_id := \varnothing$
act4_7: $cont\_actors\_target := \varnothing$
act4_8: $cont\_actors\_value := \varnothing$
act4_9: $msgC\_exists := TRUE$
act4_10: $msgC\_content := input$
end

Create ⟨convergent⟩ $\widehat{=}$
refines Create

when
grd3_0: $msgC\_exists = TRUE$
grd3_1: $msgC\_content > 0$
then
act3_0: $msgC\_content := msgC\_content - 1$
act3_1: $actor\_id := 0 \mathinner{..} num\_actors$
act3_2: $cont\_actors\_target(active\_actor + 1) := active\_actor$
act3_3: $cont\_actors\_value(active\_actor + 1) := msgC\_content$
act3_4: $active\_actor := active\_actor + 1$
act3_5: $num\_actors := num\_actors + 1$
end

Created ⟨convergent⟩ $\widehat{=}$
refines Created

when
grd3_0: $msgC\_exists = TRUE$
grd3_1: $msgC\_content = 0$
grd3_2: $msg\_exists = FALSE$
grd3_3: $active\_actor = input - 1$
then
act3_0: $msg\_exists := TRUE$
act3_1: $msg\_recipient := active\_actor$
act3_2: $msg\_content := 1$

act3_3: $msgC\_exists := FALSE$

end

Compute ⟨ordinary⟩ $\widehat{=}$
extends Compute

when
- grd3_1: $msg\_exists = TRUE$
- grd3_2: $msg\_recipient \neq final\_id$
- grd3_3: $msg\_recipient = active\_actor$

then
- act3_0: $msg\_recipient := cont\_actors\_target(msg\_recipient)$
- act3_1: $msg\_content := msg\_content * cont\_actors\_value(msg\_recipient)$
- act3_2: $num\_actors := num\_actors - 1$
- act3_3: $actor\_id := 0 \mathinner{..} num\_actors - 2$
- act3_4: $cont\_actors\_target := \{msg\_recipient\} \lhd cont\_actors\_target$
- act3_5: $cont\_actors\_value := \{msg\_recipient\} \lhd cont\_actors\_value$
- act3_6: $active\_actor := cont\_actors\_target(msg\_recipient)$

end

Finish ⟨ordinary⟩ $\widehat{=}$
extends Finish

when
- grd3_1: $msg\_exists = TRUE$
- grd3_2: $msg\_recipient = final\_id$
- grd3_3: $msg\_recipient = active\_actor$

then
- act3_0: $result := msg\_content$

end

END

MACHINE m5
REFINES m4
SEES ctx5
VARIABLES

result
num_actors
actor_id
cont_actors_target
cont_actors_value
fact_mail_msgC_content
fact_index_msgC
cont_mail_msg_content
cont_index_msg

INVARIANTS

- inv5_0: $fact\_mail\_msgC\_content \in \mathbb{N} \pfun \mathbb{N}$
- inv5_1: $msgC\_exists = TRUE \Leftrightarrow ran(fact\_mail\_msgC\_content) = \{msgC\_content\}$
- inv5_2: $msgC\_exists = FALSE \Leftrightarrow fact\_mail\_msgC\_content = \varnothing$
- inv5_3: $fact\_index\_msgC \in \mathbb{N}$
- inv5_4: $cont\_mail\_msg\_content \in (ACTOR\_ID \times \mathbb{N}) \pfun \mathbb{N}$
- inv5_6: $cont\_index\_msg \in \mathbb{N}$
- inv5_7: $msg\_exists = TRUE \Leftrightarrow ran(cont\_mail\_msg\_content) = \{msg\_content\}$
- inv5_8: $msg\_exists = FALSE \Leftrightarrow cont\_mail\_msg\_content = \varnothing$

inv5_9: $\exists n \cdot msg\_exists = TRUE \Rightarrow dom(cont\_mail\_msg\_content) = \{active\_actor \mapsto n\}$

EVENTS
Initialisation

begin
- act5_0: $result := 0$
- act5_5: $num\_actors := 0$
- act5_6: $actor\_id := \varnothing$
- act5_7: $cont\_actors\_target := \varnothing$
- act5_8: $cont\_actors\_value := \varnothing$
- act5_9: $fact\_mail\_msgC\_content := \{0 \mapsto input\}$
- act5_10: $fact\_index\_msgC := 1$
- act5_11: $cont\_mail\_msg\_content := \varnothing$
- act5_13: $cont\_index\_msg := 0$

end

Create ⟨convergent⟩ $\widehat{=}$
refines Create

any
- content
- index

where
- grd5_0: $index \in dom(fact\_mail\_msgC\_content)$
- grd5_1: $fact\_mail\_msgC\_content(index) = content$
- grd5_2: $content > 0$

then
- act5_0: $fact\_mail\_msgC\_content := \{fact\_index\_msgC \mapsto content - 1\}$
- act5_1: $fact\_index\_msgC := fact\_index\_msgC + 1$
- act5_2: $actor\_id := 0 \mathinner{..} num\_actors$
- act5_3: $cont\_actors\_target(num\_actors) := num\_actors - 1$
- act5_4: $cont\_actors\_value(num\_actors) := content$
- act5_6: $num\_actors := num\_actors + 1$

end

Created ⟨convergent⟩ $\widehat{=}$
refines Created

any
- index

where
- grd5_0: $\{index\} = dom(fact\_mail\_msgC\_content)$
  we need to guarante that there is only one msg, because of the previous machines
- grd5_1: $fact\_mail\_msgC\_content(index) = 0$
- grd5_2: $cont\_mail\_msg\_content = \varnothing$
- grd5_3: $num\_actors = input$

then
- act5_0: $cont\_mail\_msg\_content := \{(num\_actors - 1 \mapsto cont\_index\_msg) \mapsto 1\}$
- act5_3: $fact\_mail\_msgC\_content := \{index\} \lhd fact\_mail\_msgC\_content$

end

ContCompute ⟨ordinary⟩ $\widehat{=}$
refines Compute

any
- actor
- index

where
- grd5_0: $\{actor \mapsto index\} = dom(cont\_mail\_msg\_content)$
- grd5_1: $actor \neq final\_id$

then
act5_1: $cont\_mail\_msg\_content := \{(cont\_actors\_target(actor) \mapsto cont\_index\_msg) \mapsto (cont\_mail\_msg\_content(actor \mapsto index) * cont\_actors\_value(actor))\}$
act5_2: $num\_actors := num\_actors - 1$
act5_3: $actor\_id := 0 \mathinner{..} num\_actors - 2$
act5_4: $cont\_actors\_target := \{actor\} \lhd cont\_actors\_target$
act5_5: $cont\_actors\_value := \{actor\} \lhd cont\_actors\_value$
end

Finish $\langle$ordinary$\rangle$ $\widehat{=}$
refines Finish

any
actor
index
where
grd5_0: $\{actor \mapsto index\} = dom(cont\_mail\_msg\_content)$
grd3_1: $actor = final\_id$
then
act3_0: $result := cont\_mail\_msg\_content(actor \mapsto index)$
end

END

A.2. Concurent Model

Final Machine

MACHINE m5
REFINES m4
SEES ctx3
VARIABLES

tasks
results
num_actors
actor_ids
cont_actors_target
cont_actors_value
fact_mail_msgC_content
cont_mail_msg_content

INVARIANTS

inv0_0: $tasks \in REQUEST\_ID \pfun \mathbb{N}$
inv0_1: $results \in REQUEST\_ID \pfun \mathbb{N}_1$
inv3_00: $num\_actors \in dom(tasks) \tfun \mathbb{N}$
inv3_01: $actor\_ids \in dom(tasks) \tfun \mathbb{P}(\mathbb{N})$
inv3_08: $cont\_actors\_target \in dom(tasks) \tfun (\mathbb{N} \pfun ACTOR\_ID)$
inv3_10: $cont\_actors\_value \in dom(tasks) \tfun (\mathbb{N} \pfun \mathbb{N}_1)$
inv5_00: $fact\_mail\_msgC\_content \in dom(tasks) \pfun \mathbb{N}$
inv5_0: $fact\_mail\_msgC\_content = msgsC\_content$
inv5_1: $cont\_mail\_msg\_content \in (ACTOR\_ID \times REQUEST\_ID) \pfun \mathbb{N}_1$
inv5_2: $\forall a, t \cdot a \mapsto t \in dom(cont\_mail\_msg\_content) \Leftrightarrow t \mapsto a \in msgs\_recipient$
inv5_3: $\forall i \cdot i \in dom(msgs\_recipient) \Rightarrow msgs\_content(i) = cont\_mail\_msg\_content(msgs\_recipient(i) \mapsto i)$
inv5_4: $\forall i \cdot i \in dom(msgs\_recipient) \Rightarrow active\_actors(i) \mapsto i \in dom(cont\_mail\_msg\_content)$
inv5_5: $\forall i \cdot (i \in dom(fact\_mail\_msgC\_content) \land fact\_mail\_msgC\_content(i) = 0) \Rightarrow active\_actors(i) = tasks(i) - 1$

EVENTS
Initialisation
begin
act5_0: $tasks := \varnothing$
act5_1: $results := \varnothing$
act5_5: $num\_actors := \varnothing$
act5_6: $actor\_ids := \varnothing$
act5_7: $cont\_actors\_target := \varnothing$
act5_8: $cont\_actors\_value := \varnothing$
act5_10: $fact\_mail\_msgC\_content := \varnothing$
act5_11: $cont\_mail\_msg\_content := \varnothing$
end

Start ⟨ordinary⟩ $\widehat{=}$
refines Start
any
input
request
where
grd4_0: $input \in \mathbb{N}$

grd4_1: $request \notin dom(tasks)$

then

act4_0: $tasks(request) := input$
act4_2: $num\_actors(request) := 0$
act4_3: $actor\_ids(request) := \varnothing$
act4_4: $cont\_actors\_target(request) := \varnothing$
act4_5: $cont\_actors\_value(request) := \varnothing$
act5_7: $fact\_mail\_msgC\_content(request) := input$

end

Create ⟨ordinary⟩ $\widehat{=}$
refines Create

any

task

where

grd5_1: $task \in dom(fact\_mail\_msgC\_content)$
grd5_2: $fact\_mail\_msgC\_content(task) > 0$

then

act5_0: $fact\_mail\_msgC\_content(task) := fact\_mail\_msgC\_content(task) - 1$
act3_1: $actor\_ids(task) := 0 \mathinner{..} num\_actors(task)$
act3_2: $cont\_actors\_target(task) := cont\_actors\_target(task) \cup \{num\_actors(task) \mapsto num\_actors(task) - 1\}$
act3_3: $cont\_actors\_value(task) := cont\_actors\_value(task) \cup \{num\_actors(task) \mapsto fact\_mail\_msgC\_content(task)\}$
act3_5: $num\_actors(task) := num\_actors(task) + 1$

end

Created ⟨ordinary⟩ $\widehat{=}$
refines Created

any

task

where

grd5_0: $task \in dom(fact\_mail\_msgC\_content)$
grd5_1: $fact\_mail\_msgC\_content(task) = 0$

then

act5_0: $fact\_mail\_msgC\_content := \{task\} \lhd fact\_mail\_msgC\_content$
act5_1: $cont\_mail\_msg\_content(num\_actors(task) - 1 \mapsto task) := 1$

end

Compute ⟨ordinary⟩ $\widehat{=}$
refines Compute

any

task
actor
content

where

grd3_0: $task \in dom(tasks)$
grd3_1: $task \notin dom(results)$
grd5_0: $actor \mapsto task \mapsto content \in cont\_mail\_msg\_content$
grd5_1: $actor \neq final\_id$

then

act3_2: $num\_actors(task) := num\_actors(task) - 1$
act3_3: $actor\_ids(task) := 0 \mathinner{..} num\_actors(task) - 2$
act3_4: $cont\_actors\_target(task) := \{actor\} \lhd cont\_actors\_target(task)$
act3_5: $cont\_actors\_value(task) := \{actor\} \lhd cont\_actors\_value(task)$

act5_1: $cont\_mail\_msg\_content := (\{actor \mapsto task\} \lhd cont\_mail\_msg\_content) \lhd\!\!+ \{cont\_actors\_target(task)(actor) \mapsto task \mapsto content * cont\_actors\_value(task)(actor)\}$

end

Finish ⟨ordinary⟩ $\widehat{=}$
refines Finish

any
task
actor
content

where
grd3_0: $task \in dom(tasks)$
grd3_1: $task \notin dom(results)$
grd5_0: $actor \mapsto task \mapsto content \in cont\_mail\_msg\_content$
grd5_1: $actor = final\_id$

then
act3_0: $results(task) := content$

end

END

Appendix B.

Chat Server

Final Machine

MACHINE m5b
REFINES m4
SEES ctx4
VARIABLES

server_mail_msgB_content
in_flight_msgA_content
server_mail_msgB_sender
in_flight_msgA_sender
server_mail_msgSubscribe_alias
server_mail_msgSubscribe_client
client_mail_msgSubResponse_answer
session_state_client
client_mail_msgSubResponse_session
session_mail_msgBS_content
client_state_session
session_state_uname
server_state_sessions
server_state_unames
client_mail_msgAS_content
client_mail_msgAS_sender
session_mail_msgAS_content
session_mail_msgAS_sender

INVARIANTS

inv5_0: $session\_mail\_msgAS\_content \in (server\_state\_sessions \times MSG\_ID) \pfun MSG\_CONTENT$
inv5_1: $session\_mail\_msgAS\_sender \in (server\_state\_sessions \times MSG\_ID) \pfun UNAME$
inv5_2: $dom(session\_mail\_msgAS\_content) = dom(session\_mail\_msgAS\_sender)$
inv5_3: $client\_mail\_msgAS\_content \in (ran(session\_state\_client) \times MSG\_ID) \pfun MSG\_CONTENT$
inv5_4: $client\_mail\_msgAS\_sender \in (ran(session\_state\_client) \times MSG\_ID) \pfun UNAME$
inv5_5: $dom(client\_mail\_msgAS\_sender) = dom(client\_mail\_msgAS\_content)$
inv5_6: $\forall s, i \cdot s \mapsto i \in dom(session\_mail\_msgAS\_content) \Rightarrow session\_state\_client(s) \mapsto i \in dom(client\_mail\_msgA\_content)$
inv5_7: $\forall c, i \cdot c \mapsto i \in dom(client\_mail\_msgA\_content) \Rightarrow session\_state\_client^{-1}(c) \mapsto i \in dom(session\_mail\_msgAS\_content)$
inv5_8: $\forall s, i \cdot s \mapsto i \in dom(session\_mail\_msgAS\_content) \Rightarrow session\_mail\_msgAS\_content(s \mapsto i) = client\_mail\_msgA\_content(session\_state\_client(s) \mapsto i)$

inv5_9: $\forall s, i \cdot s \mapsto i \in dom(session\_mail\_msgAS\_sender) \Rightarrow session\_mail\_msgAS\_sender(s \mapsto i) = client\_mail\_msgA\_sender(session\_state\_client(s) \mapsto i)$

EVENTS
Initialisation

begin
- act1_2: $server\_mail\_msgB\_content := \varnothing$
- act1_6: $in\_flight\_msgA\_content := \varnothing$
- act2_1: $server\_mail\_msgB\_sender := \varnothing$
- act2_3: $in\_flight\_msgA\_sender := \varnothing$
- act3_0: $server\_mail\_msgSubscribe\_alias := \varnothing$
- act3_1: $server\_mail\_msgSubscribe\_client := \varnothing$
- act3_3: $client\_mail\_msgSubResponse\_answer := \varnothing$
- act4_3: $session\_mail\_msgBS\_content := \varnothing$
- act4_6: $client\_state\_session := \varnothing$
- act4_7: $session\_state\_uname := \varnothing$
- act4_8: $client\_mail\_msgSubResponse\_session := \varnothing$
- act4_9: $server\_state\_sessions := \varnothing$
- act4_10: $session\_state\_client := \varnothing$
- act4_11: $server\_state\_unames := \varnothing$
- act5_0: $session\_mail\_msgAS\_content := \varnothing$
- act5_1: $session\_mail\_msgAS\_sender := \varnothing$
- act5_2: $client\_mail\_msgAS\_content := \varnothing$
- act5_3: $client\_mail\_msgAS\_sender := \varnothing$

end

ClientSendViaSession ⟨ordinary⟩ $\widehat{=}$
extends ClientSendViaSession

any
- *client*
- *content*
- *fresh_id*

where
- grd4_0: $client \in dom(client\_state\_session)$
- grd4_1: $content \in MSG\_CONTENT$
- grd4_2: $fresh\_id \notin \{i \cdot client\_state\_session(client) \mapsto fresh\_id \in dom(session\_mail\_msgBS\_content) | i\}$

then
- act4_0: $session\_mail\_msgBS\_content(client\_state\_session(client) \mapsto fresh\_id) := content$

end

SessionForwardSend ⟨ordinary⟩ $\widehat{=}$
extends SessionForwardSend

any
- *session*
- *content*
- *msg_id*
- *fresh_id*

where
- grd4_0: $(session \mapsto msg\_id) \in dom(session\_mail\_msgBS\_content)$
- grd4_1: $session\_mail\_msgBS\_content(session \mapsto msg\_id) = content$
- grd4_2: $session \in dom(session\_state\_uname)$
- grd4_3: $session \in ran(client\_state\_session)$
- grd4_4: $fresh\_id \notin dom(server\_mail\_msgB\_content)$

then
- act4_0: $server\_mail\_msgB\_content(fresh\_id) := content$

act4_1: $server\_mail\_msgB\_sender(fresh\_id) := session\_state\_uname(session)$
act4_2: $session\_mail\_msgBS\_content := \{session \mapsto msg\_id\} \lhd session\_mail\_msgBS\_content$

end

SessionForwardReply ⟨ordinary⟩ $\widehat{=}$
refines ClientReceive

any
session
content
msg_id
sender

where
grd1_0: $content \in MSG\_CONTENT$
grd1_2: $(session \mapsto msg\_id) \in dom(session\_mail\_msgAS\_content)$
grd1_3: $session\_mail\_msgAS\_content(session \mapsto msg\_id) = content$
grd2_0: $sender \in UNAME$
grd2_1: $(session \mapsto msg\_id) \in dom(session\_mail\_msgAS\_sender)$
grd2_2: $session\_mail\_msgAS\_sender(session \mapsto msg\_id) = sender$

with
client: $client = session\_state\_client(session)$

then
act1_0: $session\_mail\_msgAS\_content := \{session \mapsto msg\_id\} \lhd session\_mail\_msgAS\_content$
act2_0: $session\_mail\_msgAS\_sender := \{session \mapsto msg\_id\} \lhd session\_mail\_msgAS\_sender$
act5_0: $client\_mail\_msgAS\_content(session\_state\_client(session) \mapsto msg\_id) := content$
act5_1: $client\_mail\_msgAS\_sender(session\_state\_client(session) \mapsto msg\_id) := sender$

end

ClientReceiveFromSession ⟨ordinary⟩ $\widehat{=}$

any
client
content
msg_id
sender

where
grd5_1: $(client \mapsto msg\_id) \in dom(client\_mail\_msgAS\_content)$
grd5_2: $client\_mail\_msgAS\_content(client \mapsto msg\_id) = content$
grd5_3: $(client \mapsto msg\_id) \in dom(client\_mail\_msgAS\_sender)$
grd5_4: $client\_mail\_msgAS\_sender(client \mapsto msg\_id) = sender$

then
act5_0: $client\_mail\_msgAS\_content := \{client \mapsto msg\_id\} \lhd client\_mail\_msgAS\_content$
act5_1: $client\_mail\_msgAS\_sender := \{client \mapsto msg\_id\} \lhd client\_mail\_msgAS\_sender$

end

AllClientsReceived ⟨ordinary⟩ $\widehat{=}$
refines AllClientsReceived

any
content
msg_id
sender

where
grd1_0: $content \in MSG\_CONTENT$
grd1_3: $\{c \cdot c \in ran(session\_state\_client) | c \mapsto msg\_id\} \lhd client\_mail\_msgAS\_content = \varnothing$
grd1_4: $msg\_id \mapsto content \in in\_flight\_msgA\_content$
grd2_0: $msg\_id \mapsto sender \in in\_flight\_msgA\_sender$
grd5_0: $\{s \cdot s \in server\_state\_sessions | s \mapsto msg\_id\} \lhd session\_mail\_msgAS\_content = \varnothing$

then

act1_0: $in\_flight\_msgA\_content := \{msg\_id\} \lhd in\_flight\_msgA\_content$
act2_0: $in\_flight\_msgA\_sender := \{msg\_id\} \lhd in\_flight\_msgA\_sender$

end

ServerReplyToSessions ⟨ordinary⟩ $\widehat{=}$
refines ServerReply

any
content
msg_id
fresh_id
sender

where
grd1_0: $content \in MSG\_CONTENT$
grd1_1: $msg\_id \in dom(server\_mail\_msgB\_content)$
grd1_2: $server\_mail\_msgB\_content(msg\_id) = content$
grd1_3: $fresh\_id \notin \{s, i \cdot s \mapsto i \in dom(session\_mail\_msgAS\_content) | i\}$
grd1_4: $fresh\_id \notin dom(in\_flight\_msgA\_content)$
grd2_0: $msg\_id \mapsto sender \in server\_mail\_msgB\_sender$

then
act1_0: $session\_mail\_msgAS\_content := session\_mail\_msgAS\_content \lhd\!\!\!- \{s \cdot s \in server\_state\_sessions | (s \mapsto fresh\_id) \mapsto content\}$
act1_2: $server\_mail\_msgB\_content := \{msg\_id\} \lhd server\_mail\_msgB\_content$
act1_4: $in\_flight\_msgA\_content(fresh\_id) := content$
act2_0: $session\_mail\_msgAS\_sender := \{s \cdot s \in server\_state\_sessions | (s \mapsto fresh\_id) \mapsto sender\} \lhd\!\!\!- session\_mail\_msgAS\_sender$
act2_1: $server\_mail\_msgB\_sender := \{msg\_id\} \lhd server\_mail\_msgB\_sender$
act2_2: $in\_flight\_msgA\_sender(fresh\_id) := sender$

end

Subscribe ⟨ordinary⟩ $\widehat{=}$
extends Subscribe

any
client
uname
fresh_id

where
grd4_0: $client \in CLIENT\_ID$
grd4_1: $uname \in UNAME$
grd4_2: $fresh\_id \notin dom(server\_mail\_msgSubscribe\_alias)$

then
act4_0: $server\_mail\_msgSubscribe\_alias(fresh\_id) := uname$
act4_1: $server\_mail\_msgSubscribe\_client(fresh\_id) := client$

end

ServerSubscribeSuccess ⟨ordinary⟩ $\widehat{=}$
extends ServerSubscribeSuccess

any
client
uname
msg_id
fresh_id
fresh_session_id

where
grd1_0: $client \in CLIENT\_ID$
grd1_1: $client \notin ran(session\_state\_client)$
grd2_0: $uname \in UNAME$

grd2_1: $uname \notin server\_state\_unames$
grd3_0: $msg\_id \mapsto uname \in server\_mail\_msgSubscribe\_alias$
grd3_1: $msg\_id \mapsto client \in server\_mail\_msgSubscribe\_client$
grd3_2: $fresh\_id \notin \{i \cdot client \mapsto i \in dom(client\_mail\_msgSubResponse\_answer) | i\}$
grd4_0: $fresh\_session\_id \notin dom(session\_state\_client)$

then

act3_0: $server\_mail\_msgSubscribe\_alias := \{msg\_id\} \lhd server\_mail\_msgSubscribe\_alias$
act3_1: $server\_mail\_msgSubscribe\_client := \{msg\_id\} \lhd server\_mail\_msgSubscribe\_client$
act3_2: $client\_mail\_msgSubResponse\_answer(client \mapsto fresh\_id) := TRUE$
act4_0: $session\_state\_uname(fresh\_session\_id) := uname$
act4_1: $session\_state\_client(fresh\_session\_id) := client$
act4_3: $server\_state\_sessions := server\_state\_sessions \cup \{fresh\_session\_id\}$
act4_4: $client\_mail\_msgSubResponse\_session(client \mapsto fresh\_id) := fresh\_session\_id$
act4_5: $server\_state\_unames := server\_state\_unames \cup \{uname\}$

end

ServerSubscribeFail ⟨ordinary⟩ $\widehat{=}$
extends ServerSubscribeFail

any

$client$
$uname$
msg_id
$fresh\_id$

where

grd1_0: $client \in CLIENT\_ID$
grd2_0: $uname \in UNAME$
grd3_0: $msg\_id \mapsto uname \in server\_mail\_msgSubscribe\_alias$
grd3_1: $msg\_id \mapsto client \in server\_mail\_msgSubscribe\_client$
grd3_2: $fresh\_id \notin \{i \cdot client \mapsto i \in dom(client\_mail\_msgSubResponse\_answer) | i\}$

then

act3_0: $server\_mail\_msgSubscribe\_alias := \{msg\_id\} \lhd server\_mail\_msgSubscribe\_alias$
act3_1: $server\_mail\_msgSubscribe\_client := \{msg\_id\} \lhd server\_mail\_msgSubscribe\_client$
act3_2: $client\_mail\_msgSubResponse\_answer(client \mapsto fresh\_id) := FALSE$

end

ClientReceiveSubResponseSuccess ⟨ordinary⟩ $\widehat{=}$
extends ClientReceiveSubResponseSuccess

any

$client$
msg_id

where

grd3_0: $client \in CLIENT\_ID$
grd3_3: $((client \mapsto msg\_id) \mapsto TRUE) \in client\_mail\_msgSubResponse\_answer$
grd4_0: $client\_mail\_msgSubResponse\_session(client \mapsto msg\_id) \mapsto client \in session\_state\_client$

then

act4_0: $client\_mail\_msgSubResponse\_answer := \{client \mapsto msg\_id\} \lhd client\_mail\_msgSubResponse\_answer$
act4_1: $client\_mail\_msgSubResponse\_session := \{client \mapsto msg\_id\} \lhd client\_mail\_msgSubResponse\_session$
act4_2: $client\_state\_session(client) := client\_mail\_msgSubResponse\_session(client \mapsto msg\_id)$

end

ClientReceiveSubResponseFail ⟨ordinary⟩ $\widehat{=}$
extends ClientReceiveSubResponseFail

any

$client$

msg_id
where
grd3_0: $client \in CLIENT\_ID$
grd3_3: $((client \mapsto msg\_id) \mapsto FALSE) \in client\_mail\_msgSubResponse\_answer$
then
act3_0: $client\_mail\_msgSubResponse\_answer := \{client \mapsto msg\_id\} \lhd client\_mail\_msgSubResponse\_answer$
end
END