NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

FRANCISCO ALCOFORADO DA GAMA DE OLIVEIRA PARRINHA

BEHAVIOUR MODELS FOR A SWARM COMPUTING FRAMEWORK

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
July, 2025

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

BEHAVIOUR MODELS FOR A SWARM COMPUTING FRAMEWORK

FRANCISCO ALCOFORADO DA GAMA DE OLIVEIRA PARRINHA

Adviser: António Ravara
Associate Professor, NOVA University Lisbon

Co-adviser: João Leitão
Associate Professor, NOVA University Lisbon

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
July, 2025

Abstract

Distributed software is becoming increasingly used to meet current demand for digital services. They are often fault-tolerant (due to redundancy) and are usually more efficient than non-distributed systems. However, most distributed protocols typically describe complex, continuously running systems, making them more error prone and difficult to debug. Behaviour models can help by validating correct protocol usage at compile time, allowing developers to detect and fix any problems related to protocols not reaching the end state (potentially leading to memory leaks), incorrect resource usage, and incorrect behaviour.

The solution, however, is not trivial. Behaviour models often require linear usage, whereas distributed software is usually highly dynamic, reactive, and inherently not linear at all, making their combination quite challenging.

This work builds on two tools: **JaTyC**, a Java tool for statically validating code with respect to their typestates, and **Babel**, a performant Java framework for creating and executing distributed protocols. They are used as means of achieving two objectives: (**i**) ensure that class usage protocols specified via JaTyC are properly followed, and (**ii**) verify that exchanged messages between Babel processes adhere to their protocols, logging any violations.

Some contributions to JaTyC have already been achieved, making the tool's current version easier to use. It is also expected to create a new version for JaTyC, capable of handling dynamic, distributed environments. The result is a combined toolkit based on JaTyC and Babel that enhances the behavioural safety and correctness in distributed systems, capable of logging incorrect messages during protocol communication.

Keywords: JaTyC, Babel, Typestates, Behaviour Models

Resumo

A utilização de sistemas distribuídos tem aumentado com a crescente procura por serviços digitais. A distribuição torna frequentemente os sistemas mais tolerantes a falhas (devido à redundância) e, geralmente, mais eficientes do que sistemas não distribuídos. No entanto, a maioria dos seus protocolos descrevem sistemas complexos, concebidos para serem executados por longas durações, o que os torna mais propensos a erros e difíceis de depurar. **Behaviour Models** conseguem validar o correto uso dos protocols em tempo de compilação, permitindo os programadores detectar e corrigir problemas como: protocolos que não atingem o estado final (o que pode levar a *memory leaks*), a má utilização dos mesmos, e a utilização incorreta de recursos.

Contudo, a integração de ambos os paradigmas não é trivial. **Behaviour Models** exigem uso linear, enquanto *software* distribuído é geralmente dinâmico, reativo, e inerentemente não linear.

Este trabalho baseia-se em duas ferramentas: **JaTyC**, uma ferramenta em Java para validação estática de código relacionado com *typestates*, e **Babel**, uma *framework* em Java de alto desempenho para a criação e execução de protocolos distribuídos. Estas ferramentas são utilizadas com o objetivo de alcançar dois propósitos: (**i**) garantir que os protocolos especificados com o JaTyC sejam corretamente seguidos, e (**ii**) verificar se as mensagens trocadas entre processos Babel estão de acordo com seus protocolos, registando qualquer violação.

Algumas contribuições para o JaTyC já foram concretizadas, tornando a versão atual da ferramenta mais fácil de utilizar. Espera-se também criar uma nova versão do JaTyC, capaz de lidar com ambientes dinâmicos e distribuídos. O resultado é um *toolkit*, baseado no JaTyC e no Babel, que melhora a segurança e a correção do comportamento de sistemas distribuídos, capaz de registar mensagens incorretas durante a comunicação entre protocolos.

Palavras-chave: JaTyC, Babel, Typestates, Behaviour Models

Contents

List of Figures

List of Algorithms

Listings

1 Introduction

1.1 Context

Distributed software is becoming increasingly used to meet the current demand for digital services. These systems typically rely on having each node operating independently, without having any central controller or index node. They can usually self-organize and adapt to external (e.g. a node stops responding [6]) or internal state (e.g. a leader election operation [22]) changes. Additionally, when the system contains a high quantity of nodes, it can leverage their combined processing power to achieve a common processing goal (e.g. swarm computing [25]).

As a result, distributed systems are often fault-tolerant and they are usually more efficient and scalable than their counterpart (this is, non-distributed systems). However, it does not come without trouble. Most distributed protocols typically describe complex systems, making them harder to develop and more error-prone. Moreover, if they are stateful, they usually require coordination and consensus before agreeing on a value or operation [14].

Detecting programming errors at compile time becomes then a crucial activity for alleviating the development complexity of these protocols. Thankfully, *behaviour models* provide a strong foundation for such analysis. They describe how systems transition between states, enabling static verification of protocol correctness. As a result, they can be used for early detection of errors like memory leaks, resource misuse, and incorrect behaviour [1,2,3].

This work builds on two tools: JaTyC 2.2, a Java tool for statically validating code with respect to their typestates, which effectively encode behaviour models; and Babel 2.1, a reactive, event-driven, performant Java framework for creating and executing distributed protocols.

[1] https://www.cisa.gov/sites/default/files/2023-12/The-Case-for-Memory-Safe-Roadmaps-508c.pdf

[2] https://dl.acm.org/doi/10.1145/3708553

[3] https://link.springer.com/chapter/10.1007/978-3-030-64437-6_6

1.2 Problem

Combining both tools - JaTyC and Babel - is not trivial. Due to Babel's event-driven nature, Babel protocols require a collection storing each pending event. The collection, or event-queue, will eventually be internally processed, raising their corresponding event-handlers. Each protocol is single-threaded, and the event-queue is also processed sequentially. It also grows in a natural way: events are processed as fast as the process can, and events are received in a natural fashion according to the system's usage (internally and externally).

Typestates effectively model a finite state-machine, representing their underlying protocol's states and transitions. For this reason, JaTyC must enforce linearity [36] on the usage of objects associated with typestates. Typestates can be associated to Babel protocols, allowing JaTyC to verify their behaviour at compile time. A typestate for a Babel protocol would contain its different event-handlers, organized by their correct states. This means a typestate associated to a Babel protocol encodes the protocol's behaviour through its callbacks' usage. An example is displayed in the section B.7.

Since each event, when processed, raises its corresponding callback, it means the event-queue's order corresponds to the protocol's future behaviour. However, the event-queue is a collection whose state only changes during the protocol's runtime, making it impossible to decide, at compile time, its future order, how many elements it will have, and which events it will contain. As a consequence, since the event-queue describes the order of which the typestate's functions are invoked, it becomes impossible to statically verify Babel protocol's behaviours, even if they are associated to typestates.

Moreover, there is no implementation that directly links both tools. As it is explain further below in the section 2.1, it is required to register any created Babel protocol through the usage of a particular function, passing the new protocol's instance as its argument. The method allows Babel to internally manage any created protocol. On the other hand, to guarantee linearity, JaTyC enforces on all methods that it does not know, that their arguments do not contain any typestate. It does so, because it cannot conclude at which state does the typestate need to be when accepted by the method. For this reason, it is impossible to add any Babel protocol associated to a typestate to the Babel's internal engine. JaTyC simply blocks the protocol registration process, marking it as an illegal operation.

These are the two main problems when integrating both tools - the behaviour for all Babel protocols is reflected by their dynamic event-queue's order, making it impossible to verify at compile time; and JaTyC does not allow any Babel protocol with typestate to be registered to the Babel's internal engine. It is shown that JaTyC's linearity and Babel's event-driven nature enter in conflict, rendering JaTyC's usage in validating Babel protocols useless.

1.3 Objectives

There are two main objectives to accomplished:

1. **internally**, it is wanted to use and explore JaTyC as extensively as possible, as means of eliminating memory related errors, wrong behaviour, incorrect resource usage, and protocol non-completion for any distributed protocol created with Babel;

2. **externally**, it is required to analyze and validate the exchanged messages between different distributed processes, ensuring these are sound according to their protocols, logging any information regarding those that are not.

1.4 Expected Contributions

Some contributions to JaTyC have already been achieved, making the tool's current version easier to use. Note the actual changes to the codebase were done by JaTyC's original creator.

1. Firstly, JaTyC did not correctly support *static* fields. This problem was fixed and static fields for primitive and String types was finally supported. The previously presented error can be seen in the listing 1.1;

2. Lastly, it was found that in several instances, JaTyC assumed objects to possibly be null, when it was known they were not. To fix this issue, a new utility function - *nonNull()* - was added. The listing 1.2 illustrates an example for its usage.

Regarding future contributions, it is expected to create a new version for JaTyC capable of handling dynamic, event-driven environments, that can be used with Babel's protocols. The final product tackles the described problems, achieving both **internal** and **external** objectives. The result is a toolkit based on JaTyC and Babel that enhances the behavioural safety and correctness in distributed software, presenting a monitoring system capable of ensuring the exchanged messages between the developed distributed systems are sound to their protocols, logging any information about those that are not.

Listing 1.1: The displayed error message when using a static value. Here, BitSend.ID is a String constant. JaTyC displays it cannot be accessed when used. This problem was fixed.

```
1 src/main/java/abp/messages/BitSend.java:10: error: Cannot access[abp.messages.BitSend.ID]
2         super(ID, bit);
```

Listing 1.2: An example showing how to use the new *nonNull()* utility method. Here it is assumed *getLogger* will never return a null value.

```
1 public class AlternatingBitProtocol extends PublicGenericProtocol {
2     private static final Logger logger =
3         nonNull(LogManager.getLogger(AlternatingBitProtocol.class));
4     ...
```

2

Background

2.1 Babel

Babel [1][15] is a Java framework for developing, implementing and executing distributed protocols. It encourages an event-driven programming style and execution model, simplifying the task of translating distributed protocol definitions into code. The framework aims to remove the complexity of low-level aspects from the programmer, such as the use of communication primitives, scheduling and handling timed actions, and non-trivial concurrency issues that naturally arise when implementing complex systems; thus, streamlining the development of distributed protocols and decreasing prototype production times. These properties are of significant interest to researchers that want to test, and analyze distributed systems, or practitioners that want to compare different solutions. This does not mean it is only useful for prototyping. Quite the contrary, as Babel was built with performance in mind, allowing its users to develop efficient, production ready systems.

Babel was also built to be generic. Its provided abstractions and tools are generic enough for creating a wide variety of distributed protocols. Furthermore, Babel also allows protocols to be developed independently from the networking abstractions that support communication between protocols in different processes.

2.1.1 Architecture

Babel's architecture can be seen as being composed by three layers: the **protocol** layer, the **core** layer, and the **network** layer.

The **protocol** layer contains all protocols created by the developers, encoding the behaviour of the developed system. Each Babel protocol can be modeled as a state-machine, whose state evolves when reacting to external events (these can be from other processes, or from other protocols in the same process). For this reason, each protocol contains an event-queue. Events can be of the following types: *Timers*, *Channel Notifications*, *Network Messages* and *Inter-process events*. It is also important to note that each protocol executes in its own thread, which handles received events by executing their corresponding

[1] Babel's original repository: https://github.com/pfouto/babel-core

callbacks sequentially. As a result, Babel shields developers from concurrency issues, since communication is done via message passing.

The **core** layer is invaluable, responsible for coordinating the execution of all protocols within a Babel process. All communications between protocols are mediated by the core component, since its main responsibility is to deliver events to each protocol's queue. When a protocol wants to send a message to another, the core module interacts with the sender's networking layer, sending the message to the receiver's address. Then, the receiver's network layer delivers the message to its core module, where it will be routed to the correct protocol. Timer events are also handled by the core module. When a timer is triggered, the core module delivers the event to the correct protocol.

The **network** layer is responsible for the connections between processes. In this layer, Babel employs an abstraction called *Channels*. Channels abstract the complexity of dealing with networking, each one provides a different behaviour. There are three main channels:

- *TCPChannel* - provides the ability to establish TCP connections (outgoing and ingoing). It is the most commonly used;
- *ServerChannel* - does not establish connections, only accepts them;
- *ClientChannel* - the ServerChannel counter-part.

Channels can generate events. For example, the *TCPChannel* generates events whenever an outgoing connection is established, fails to be created, or when it becomes disconnected. They are implemented using Netty [2], a popular Java networking framework, although the user does not have to interact with it directly. The framework also allows its users to create their own channels, if they require a specific behaviour or a guarantee at the network level. Lastly, a Babel protocol can use any number of channels, and a channel can be shared by more than one protocol.

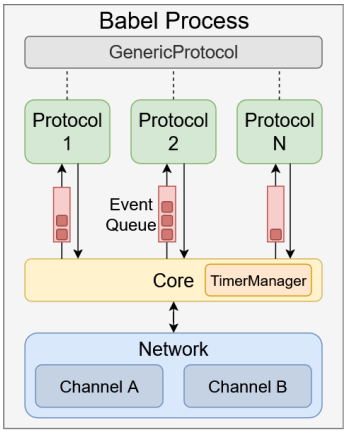


Figure 2.1: Babel process architecture.

[2] Netty's website: https://netty.io/

2.1.1.1 Creating a Babel instance and custom protocols

Babel instances are singleton. Protocols can be registered to the Babel instance. With so, these protocols become managed by its core engine, becoming part of Babel's execution. Every developed protocol must extend a provided generic class: the **GenericProtocol** class. This class contains most generic features that allow registering event handlers, setup timers, message serializers, and other primitives that contribute to the framework's workflow. Note **GenericProtocol** has several constructors. The most commonly used (and simplest one), is a constructor that accepts the name of the protocol (as a String), and the protocol's ID (as a short). Each protocol extending the **GenericProtocol** class must also contain an *init* function, with one parameter whose type is *Properties* (from java.utils), as described in the listing below. The method is used to initiate the protocol's state. The *Properties* instance contains configurations set in a specified configuration file, if there is one. Otherwise, it includes configurations passed via command-line arguments when launching the program.

Listing 2.1: Creating the main class.

```
1  public class Main {
2          private static void main(String[] args) {
3                  // Gets the singleton Babel instance
4                  Babel babel = Babel.getInstance();
5
6                  // Loading configurations to a Properties instance
7                  //              (falls back to args if no file is found)
8                  Properties props = Babel.loadConfig(args, "config.properties");
9
10                 // Registers a new instance of MyProtocol and initiates it
11                 MyProtocol proto = new MyProtocol();
12                 babel.registerProtocol(proto);
13                 proto.init(props);
14         }
15 }
16
17 // Defining a custom Babel protocol
18 public class MyProtocol extends GenericProtocol {
19         private static final String PROTO_NAME = "MyProtocol";
20         private static short PROTO_ID = 123;
21
22         public MyProtocol() {
23                 // Must call super with the protocol's name and ID
24                 super(PROTO_NAME, PROTO_ID);
25         }
26
27         @Override
28         public void init(Properties properties) {
29                 // Initiate the protocol's state here
30         }
31 }
```

2.1.1.2 Timers

Timers allow the creation of periodic tasks, or timed/postponed actions (e.g. checking if a given peer is still alive). All created timers must extend a provided class called **ProtoTimer**. ProtoTimers are simple classes, requiring only the timer's ID in their constructor. This ID does not correspond to an instance's ID, rather, it identifies the timer's class. Each class representing a timer contains its own ID. Babel still creates a unique ID for each timer instance, however, its role is different than the one provided in the timer's constructor.

Once a timer is created, one can set it up using the *setupTimer* function offered by **GenericProtocol**. If the timer is meant to be periodic, then *setupPeriodicTimer* is preferable. To cancel any timer, the programmer can always call *cancelTimer*, passing the correct timer ID (in this case, the instance's one). Protocols can react to timer events by registering their corresponding handlers. The listing below illustrates how to use timers in Babel protocols.

Listing 2.2: Using timers in Babel protocols.

```
1  public class MyProto extends GenericProtocol {
2          ...
3
4          @Override
5          public void init(Properties properties) {
6                  registerTimerHandler(MyTimer.ID, this::uponMyTimer);
7                  registerTimerHandler(MyTimer.ID, this::uponMyTimer);
8
9                  // it will be raised after 2000ms
10                 long id1 = setupTimer(new MyTimer(), 2000);
11
12                 // it starts after 1000ms, rasing events every 50ms
13                 long id2 = setupPeriodicTimer(new MyTimer(), 1000, 50);
14
15                 // The first timer executes, the second does not
16                 cancelTimer(id2);
17         }
18
19         private void uponMyTimer(MyTimer timer, long timerId) {
20                 // React to the timer ...
21         }
22 }
```

Listing 2.3: Creating a custom timer.

```
1  public class MyTimer extends ProtoTimer {
2          private static final short ID = 100;
3
4          public MyTimer() {
5                  super(ID);
6          }
7
8          @Override
9          public ProtoTimer clone() {
10                 return this;
11         }
12 }
```

2.1.1.3 Inter-protocol communication

For inter-protocol commucation, this is, the communication between two protocols within the same process, Babel offers a generic class called **ProtoRequests**. Every request must extend this class. Requests can be sent to other protocols, in return they can reply with classes extending **ProtoReply**. As stated before, Babel is an event-driven framework. Protocols can only react to requests and replies if their handlers are correctly registered (using *registerRequestHandler* and *registerReplyHandler*) . To send any request or reply, the programmer may use the *sendRequest* and *sendReply* methods, respectively.

Listing 2.4: Creating a custom request.

```
public class MyRequest extends ProtoRequest {
        private static final short ID = 100;

        public MyRequest() {
                super(ID);
        }
}
```

Listing 2.5: Creating a custom reply.

```
public class MyReply extends ProtoReply {
        private static final short ID = 101;

        public MyReply() {
                super(ID);
        }
}
```

Listing 2.6: How to send requests in a protocol. The requester must also be able to handle the reply.

```
public class ProtoOne extends GenericProtocol {
        private static final short ID = 200;
        ..

        @Override
        public void init(Properties properties) {
                // Register the reply handler first
                registerReplyHandler(MyReply.ID, this::uponMyReply);

                // Send a request
                sendRequest(new MyRequest(), ProtoTwo.ID);
        }

        private void uponMyReply(MyReply reply, short from) {
                // React to the rquest's reply ...
        }
}
```

Listing 2.7: How to handle requests and send replies in a protocol.

```
public class ProtoTwo extends GenericProtocol {
        private static final short ID = 201;
        ...

        @Override
        public void init(Properties props) {
                // Registering the handler for incoming requests of type MyRequest
                registerRequestHandler(MyRequest.ID, this::uponMyRequest);
        }

        private void uponMyRequest(MyRequest request, short from) {
                ...

                // Reply back
                sendReply(new MyReply(), from);
        }
}
```

Notifications are also an important way of establishing inter-process communications. Contrary to requests and replies, notifications are not sent to individually specified hosts, and they do not expect any reply at all. Notifications work similarly to a publisher/-subscriber pattern. Protocols can subscribe to notifications. Triggered notifications send events to all subscribed protocols. Their event handlers are registered using the *registerNotificationHandler* function, provided by the GenericProtocol class. To send notifications to its subscribers, the function *triggerNotification* must be called.

Listing 2.8: Creating a custom notification.

```
public class MyNotification extends ProtoNotification {
        private static final short ID = 300;

        public MyNotification() {
                super(ID);
        }
}
```

Listing 2.9: Using notifications. In this example, the protocol sends a notification to itself.

```
public class MyProto extends GenericProtocol {
        ...

        @Override
        public void init(Properties props) {

                // Subscribe first
                subscribeNotification(MyNotification.ID, this::uponMyNotification);

                // Trigger a new notification
                triggerNotification(new MyNotification());

                // How to unsubscribe
                unsubscribeNotification(MyNotification.ID);
        }

        private void uponMyNotification(MyNotification notification, short sourceProto) {
                // React to the notification...
        }
}
```

2.1.1.4 Inter-process communication

Channels must be established between two different processes, allowing them to send messages to each other. To do so, one must use the function *createChannel*, specifying the type of channel intended to be created. Protocols commonly use only one channel. A channel may establish and mantain connections between a set of hosts, creating a network. To open and close connections with other hosts, the methods *openConnection* and *closeConnection* are respectively used.

Listing 2.10: How to create a TCPChannel.

```
public class ExampleProto extends GenericProtocol {
        ...
        @Override
        public void init(Properties properties) throws IOException {
                ...

                // Create a properties object and assing values to the following settings
                var channelProps = new Properties();
                channelProps.setProperty(TCPChannel.ADDRESS_KEY, myAddress);
                channelProps.setProperty(TCPChannel.PORT_KEY, port);
                channelProps.setProperty(TCPChannel.HEARTBEAT_INTERVAL_KEY, "1000");
                channelProps.setProperty(TCPChannel.HEARTBEAT_TOLERANCE_KEY, "3000");
                channelProps.setProperty(TCPChannel.CONNECT_TIMEOUT_KEY, "1000");

                // Then, call createChannel, specifying the type and its properties
                createChannel(TCPChannel.NAME, channelProps);
        }
}
```

Listing 2.11: Reacting to channel events and opening connections.

```
1  public class ExampleProto extends GenericProtocol {
2          ...
3          @Override
4          public void init(Properties properties) throws IOException {
5                  ...
6                  // Babel also allows reacting to channel event
7                  registerChannelEventHandler(channelId, OutConnectionFailed.EVENT_ID,
8                                                  this::uponOutConnectionFailed);
9                  registerChannelEventHandler(channelId, OutConnectionUp.EVENT_ID,
10                                                 this::uponOutConnectionUp);
11
12                 // Finally, the user can open connections
13                 Host target =  new Host(InetAddress.getByName("127.0.0.1"), 8080);
14                 openConnection(target);
15
16                 // (an example for closing connections)
17                 closeConnection(target);
18         }
19
20         // Channel event handlers
21         private void uponOutConnectionFailed(OutConnectionFailed<ProtoMessage> event,
22                                                 int channel) {
23                 ...
24         }
25         private void uponOutConnectionUp(OutConnectionUp event, int channel) {
26                 ...
27         }
28 }
```

Intuitively, messages must be sent when inter-process communication is used (e.g. when two protocols in different processes communicate with each other). For this purpose, the framework provides the generic class called **ProtoMessage**. Every message used in protocols must extend this generic class, specifying the message's ID. Once again, this is not an identifier for an instance of a message. It identifies the message's class among all others. Note that a connection with the target host must already be opened before any message can be sent (they are queued if there isn't one).

Babel provides both *sendMessage* and *registerMessageHandler* functions via the GenericProtocol class, allowing protocols to send messages and react to incoming ones, respectively. Since messages travel through the network layer, they are required to be serialized into bytes before being sent; and deserialized into the correct message class before the receiver can react to them. For this purpose, the GenericProtocol class also provides one other protected function: *registerMessageSerializer*. Message serializers implement an interface called **ISerializer**, supporting both serializing and deserializing functions. For every message that a protocol may send, it is required to have its serializer registered, otherwise Babel may reject the operation entirely.

Listing 2.12: Creating a custom message.

```
public class MyMessage extends ProtoMessage {
        private static final short ID = 100;

        public MyMessage() {
                super(ID);
        }

        // The serializer for the message
        public static ISerializer<MyMessage> serializer = new ISerializer() {
                @Override
                public void serialize(MyMessage, ByteBuf out) throws IOException {
                        // Write to "out"
                }

                @Override
                public void deserialize(ByteBuf in) throws IOException {
                        // Read from "in"
                }
        }
}
```

Listing 2.13: How to send a message to another protocol.

```
public class ExampleProto extends GenericProtocol {
        ...
        @Override
        public init(Properties properties) {
                ...
                // Register the handler and the serializer first
                registerMessageHandler(MyMessage.ID, this::uponMyMessage);
                registerMessageSerializer(MyMessage.ID, MyMessage.serializer);

                // Create target host and send message
                Host target = new Host(InetAddress.getByName("127.0.0.1"), 8080);
                sendMessage(new MyMessage(), target);
        }

        private void uponMyMessage(MyMessage message, Host sender,
                                   short sourceProto, int channelId) {
                // React to messages of type MyMessage
        }
}
```

2.1.2 Prototypes

Several prototype projects were developed, exploring many functionalities offered by the framework and creating simple use cases for the tool. Later on, these projects were used to test JaTyC's integration. Three examples were created, ranking by difficulty: the *AlternatingBitProtocol*, the *NumberSetProtocol*, and the *BitVoteProtocol*.

The prototype projects are too big to be displayed in the document: they include several classes and their protocols' implementations are very extensive. For this reason, the examples are presented in the form of pseudo-code in the appendix A. Their implementation, with and without JaTyC, is also provided via a public repository [3] on GitHub.

Note that all protoypes that require a leader in their peer-to-peer network do not have any leader election system. Its implementation is considered to be difficult enough to require its own seperate protocol.

In the following chapters, each protocol is going to be briefly explained.

2.1.2.1 Alternating Bit Protocol

The Alternating Bit Protocol is an acknowledgement protocol, and it is the simplest protocol of all developed prototypes. There are two nodes: a *sender* and a *receiver* node.

The sender periodically sends a message with a bit to the receiver. The receiver then compares its last acknowledged bit with the received one, and then, if they are the same, it replies back with the same bit, updating its last acknowledged one; otherwise, it just sends the last acknowledged bit it has stored. This way, the sender can check if it received the correct acknowledgement for the sent bit. If so, it flips it. The protocol continues indefinitely.

2.1.2.2 Number Set Protocol

The Number Set Protocol is a protocol supporting multiple nodes, extending the Alternating Bit Protocol's difficulty. There is a leader in the peer-to-peer network, known by all nodes. All nodes contain a set of integers and a set of unique numbers. The leader periodically selects an element from the unique number set, adds it to its own set, removes it from the unique number set, and broadcasts an *update* message with the new value.

To support failures, the protocol also contains a session and only the leader can use it. A session is essentially a collection that maps each peer in the network to its pending messages that are yet to be acknowledged. When broadcasting updates, the leader always sends a list containing the messages to be acknowledged combined with the new set update.

Peers that receive a message with their session reply back, sending the received list of messages to the leader. This serves as an acknowledgement that the peer received the

[3] Babel prototype projects' repository: https://github.com/f-parrinha/BehaviourModels-for-a-Swarm-Computing-Framework-BabelExamples

update. The leader then keeps updating the session, removing acknowledged messages from their corresponding peer.

The protocol runs as long as there are unique numbers left and updates to be acknowledged.

2.1.2.3 Bit Vote Protocol

The Bit Vote Protocol is the most intricate protocol of all developed prototypes. It supports multiple nodes, having a leader that is known by all peers. The leader proposes a bit election to all nodes, while the other nodes can only vote to the proposal (which is done randomly with equal probability). The leader uses a majority-quorum to decide the proposal's final value, 0 or 1, sending a final write-back message to all participants.

Each round has its number of retries, supporting the possibility of missed vote responses. The leader keeps retrying, sending a vote request to all peers that have not yet replied (or whose response may have been lost), maximizing the number of votes. A round is considered to be terminated when no more retries are left, or when all peers have already replied. At the end, the round is considered to be successful when a majority-quorum has been reached – that is, when the leader has received:

$$\left\lfloor \frac{n}{2} \right\rfloor + 1$$

replies, considering n is the number of peers in the system. If a vote round does not reach a majority, and yet, no retries are left, the leader assumes the vote round has failed. At this stage, the leader also decides the final value (0 or 1) by choosing the value with the largest representation among the received votes.

When a vote round is finished, the leader then sends a write-back message to all peers, sending its decided value and its status (successful or failed). The protocol continues indefinitely, having the leader proposing several vote rounds.

2.1.3 JaTyC/Babel Integration

Babel prototypes were used to test JaTyC's integration. JaTyC proved useful by detecting potential *null pointer exceptions* and enabling the creation of typestates for utility classes, improving protocol and behavioural safety.

However, their integration had issues. JaTyC lacks proper support for lambda expressions and generic class inheritance. Method references cannot be used, and extending generic classes is impractical unless the subclass remains generic (e.g., `MessageHandler<T> extends Handler<T>` works, but `MessageHandler extends Handler<Message>` does not).

Additionally, protocols associated to a typestate cannot yet be registered, as `Babel.registerProtocol()` is treated by JaTyC as an *anytime* method[28] , preventing static checks at registration.

2.2 JaTyC

JaTyC [4][28] - Java Typestate Checker - is a Java tool inspired by the Mungo toolset [5], used to statically verify Java source code with respect to typestates. Typestates are class usage protocols: they define which methods, belonging to a given class, can be invoked at a given state. JaTyC is also a plugin for the Checker [6] framework: a tool that extends the native Java type system to support several compiler plug-ins that verify the absence of several software errors, including null pointer exceptions, unintended side effects, SQL injections, concurrency errors, mistaken equality tests, and other run-time errors. It is only a transparent checker, meaning it does not change the default Java compilation procedure. Essentially, JaTyC is a tool that allows one to complement the code with protocol specifications and statically validate whether the code respects the defined protocols [3], ensuring protocol compliance and completion.

Well-developed projects using JaTyC are guaranteed to have the following properties: objects follow their typestates when used, they reach the **end** state, and no null-pointer exceptions are raised. To ensure these properties, JaTyC linearizes the usage of objects associated with typestates [3]. It tracks their behaviour using their protocols, detecting not only incorrect usage, but any possible aliasing as well. Values with typestates must be used exactly once along a protocol path. This behaviour can be compared to *move semantics* in Rust [23], where objects are consumed when used, effectively removing aliasing. In comparison, JaTyC can detect non-linear usage at compile time, allowing the developer to fix any problems regarding the detected interference, achieving similar effects.

Although linear behaviour is required, Java - JaTyC's target environment - is not linear at all. References can be referenced by multiple values, and they can be copied and discarded by different entities. When instances with typestates are used in non-linear scenarios, where aliasing is detected, JaTyC assumes the instance is in a reserved state: the **Shared** state. In this state, almost no usage is allowed, since, due to the detected interference, it is impossible to track which state is the instance at (in the code), and at which states can it transition to. It means no method belonging to the typestate can be invoked. It also serves as a safety mechanism: invoking methods when an object is in a Shared state means there are resources that may be required, that still do not exist, as the instance may be in a state different than the one assumed by the programmer when writing the code.

[4]JaTyC's repository: https://github.com/jdmota/java-typestate-checker
[5]Mungo's website: https://www.dcs.gla.ac.uk/research/mungo/index.html
[6]The Checker Framework's website: https://checkerframework.org/

Another vital feature, besides protocol completion and compliance, is **Nullness checking** - null pointer errors are the cause of most runtime exceptions in Java programs [28]. When using JaTyC, all types are guaranteed to be **non-null** by default. It is still allowed to specify nullable types in the code. When creating a variable (or in a method parameter), the programmer can use the **@Nullable** annotation to specify it is a nullable type. This approach helps prevent null pointer exceptions at compile time, making Java programs more robust and reducing the likelihood of runtime failures due to unintended null values.

Subtyping is also supported by JaTyC - a typestate can be a subtype of another typestate. When dealing with inheritence, there are four cases that need to be considered, regarding whether the classes have typestates or not:

	Parent	**Child**
Case 1	No	No
Case 2	Yes	Yes
Case 3	No	Yes
Case 4	Yes	No

Table 2.1: Different/Possible inheritance cases. Yes and No refer to whether the class has a typestate associated to it.

For case number 1, the solution is trivial. Since no class has a typestate associated to them, JaTyC does not have to do anything. Regarding case number 2, where both the parent and child classes have typestates, the subtyping system is used. For the third case, the use of overwritten methods are governed by the inherited protocol (parent class) and newly added methods are treated as *anytime* methods. *Anytime* methods are functions that can be called at any moment. All functions not belonging to a protocol are also considered to be *anytime* methods. Finally, for the fourth case, all methods belonging to the parent class are considered to be *anytime* methods. It is also enforced that these remain like so in its subclass as well, which implies these methods cannot be written in the subclass's protocol [3].

2.2.1 Usage

Typestates must be associated to classes, such that JaTyC can statically evaluate their behaviour according to their assigned protocol. There are two required steps to define a typestate and assign it to a Java class:

- Define the typestate using JaTyC's syntax in a specified file;
- Assign the defined typestate to a class using a suite of annotations provided by JaTyC.

2.2.1.1 Defining typestates

To create typestates, one must start by creating a separate file, not required but usually ending with the extension ".protocol" (e.g. MyProtocol.protocol). The syntax used to define typestates is quite simple and related to Java's syntax. Typestates are defined by using the word **typestate**, followed by the name of the typestate.

Listing 2.14: Typestate definition.

```
// Firstly, "typestate" is written, then the name "MyTypestate" is given.
// The body of the typestate is written between brackets
typestate MyTypestate {
        ...
}
```

States are written in the typestate's body. Within each state, there can be a set of methods. When a object is in a particular state, only the defined methods for that state can be invoked. To define a method, one must provide the return value - same as in Java; the name of the method, the type for each argument that the function can accept, and, finally, followed by colon (**:**), to which state will the object transition to.

Listing 2.15: Defining states and their allowed methods.

```
typestate MyTypestate {
        // This is a state
        InitState = {
                // begin() belongs to InitState
                //      and transitions to OtherState
                void begin(int): OtherState,
                ...
        }

        // This is another state
        OtherState = {
                // value() returns a String and remains in the same state
                String value(): OtherState,
                ...
        }
}
```

Since these typestate definitions represent a Finite-State Machine for the underlying protocol, an end state must also be defined. To do so, one must write a function that, at the end of its execution, it transitions the protocol to the end state, by using a reserved keyword - **end**.

Listing 2.16: Adding the end state.

```
typestate MyTypestate {
        ...

        MyState = {
                // Transitioning to the "end" state after a shutdown call
                void shutdown(): end,
                ...
        }
}
```

Last, but not least, it is also important to notice that it is possible to import other classes to the protocol specification. The import section is defined manually at the top and the syntax is the same as Java's. This allows specifications to have methods returning and consuming instances of custom classes designed by the developer.

Listing 2.17: Importing classes.

```
// Defining the package
package org.myproject;

// Importing MyClass
import org.myproject.MyClass

// Importing MyOtherClass
import org.myproject.MyOtherClass

typestate MyTypestate {
        ...

        MyState = {
                MyClass get(MyOtherClass): MyState,
                ...
        }
}
```

2.2.1.2 Droppable States

Although it was explained earlier that it is required to have a transition to the end state, JaTyC supports a feature that removes the need to write it. **Droppable states** is a feature that allows one to specify states in which an object may be stopped from being used without having to reach the final state. In other words, droppable states do not require a method that explicitely transitions to the end state.

It is crucial to have droppable states, as distributed software is usually designed to run for very long periods of time. Instances may be running in idle states that do not require an explicit shutdown function, and yet they can still stop and transition to a final state without interference.

To define droppable states, one must add a line with "drop:end" in the desired state, as illustrated below:

Listing 2.18: Adding Droppable States.

```
typestate MyTypestate {
        ...

        MyDroppableState = {
                // State's body ...
                ...
                // This marks the state as "droppable"
                drop: end
        }
}
```

2.2.1.3 Assigning typestates

Now that it is possible to create class usage protocols, it is required to assign them to the desired classes such that JaTyC can statically verify their behaviour. It is offered to the programmer a suite of annotations to assign protocols and control (in certain cases) their flow, while programming in Java. All described annotations accept a string argument. There are three important annotations to consider:

1. **@Typestate()** - assigns one typestate to a class. Accepts a string with the protocol's file name;

2. **@Ensures()** - used on methods returning an object with protocol. Its parameter corresponds to the state at which the returned object will be at. It guarantees the object will be at that given state at the end of the method's execution;

3. **@Requires()** - used when a method parameter accepts objects with protocol. Its argument is used for specifying which state is the object required to be at. It dictates the method's argument must be in the given state at the beginning the the function's execution.

When assigning a protocol to a class, one must use the **@Typestate** annotation, passing the corresponding protocol file in its parameter. For the following example, it is considered there is a protocol file called "MyProtocol.protocol". The annotation searches for a file whose name matches the passed string. In the case below, it searches in the current directory for a file whose name is "MyTypestate". Note it is also possible to write relative paths to the annotation's argument (e.g. @Typestate("../../MyTypestate")).

Listing 2.19: Associating typestates to Java classes.

```
1 // The MyProtocol typestate is now associated to MyClass
2 @Typestate("MyProtocol")
3 public class MyClass {
4         ...
5 }
```

For finer control over the protocol's state, one can use two different annotations: **@Ensures** and **@Requires**. Both annotations are used on functions. The first is for returned values, the latter for functions' parameters. Note these annotations require linearity in the object's usage. There must be no aliasing, otherwise JaTyC infers the protocol to be in the Shared state, granting both annotations useless.

Listing 2.20: Controlling the flow with @Ensures and @Requires.

```
1  @Typestate("MyProtocol")
2  public class MyClass {
3          ...
4
5          // "Ensures" guarantees the returned value will be in "State2"
6          @Ensures("State2")
7          public static MyClass fun(@Requires("State1") MyClass arg) {
8                  // "Requires" enforces that "arg" must currently be in "State1"
9                  ...
10         }
11 }
```

Note the given function "fun" is a static method. It is easier to work with instances with typestates, and with both Ensures and Requires annotations, when writing static functions. They belong to the class, and not to a particular instance, thus, removing any implicit aliasing from the instance's natural state. All resources and state information is passed via the static method's parameters, granting some form of linearity. As a result, JaTyC infers and tracks precisely the code's behaviour according to the object's protocol.

2.2.2 Prototypes

Several example projects were developed to try the different JaTyC's features, showing different use cases for the tool. Ranking by complexity, there are three important prototype projects to be considered: HouseAlarm, VirusDetector and Robot.

It is important to notice no developed prototype aims to fully capture the object they represent - the code for HouseAlarm does not actually represent a production-ready house alarm system. Rather, they represent the behaviour of the abtraction they try to capture.

2.2.2.1 House Alarm

The HouseAlarm prototype is the simplest of all developed prototypes. It describes an alarm system with different sensors and with three different alarm states: active, triggered and critical.

Listing 2.21: HouseAlarm's typestate.

```
typestate HouseAlarm {
    Disabled = {
        void loadSensors(): LoadSensors,
        void activate(): BootValidator,
        void shutdown(): end
    }

    BootValidator = {
        boolean isValid(): <true: Active, false: Disabled>
    }

    Active = {
        boolean detectMotion(): <true: Triggered, false: Active>,
        boolean disable(long): <true: Disabled, false: Active>
    }

    Triggered = {
        void createSound(): Triggered,
        boolean disable(long): <true: Disabled, false: Triggered>,
        boolean isToleranceBroken(): <true: Critical, false: Triggered>
    }

    Critical = {
        void createSound(): Critical,
        void alertPolice(): Critical,
        boolean disable(long): <true: Disabled, false: Critical>
    }

    LoadSensors = {
        void configSensitivity(double): LoadSensors,
        void addSensor(Object): LoadSensors,
        void removeSensor(String): LoadSensors,
        void done(): Disabled
    }
}
```

Listing 2.22: HouseAlarm's class in Java, associated to its typestate.

```
@Typestate("HouseAlarmProtocol")
public class HouseAlarm {
    private final HashMap<UUID, Sensor> sensors;

    public HouseAlarm() {
        sensors = new HashMap<>();
    }

    // Sound and alarm
    public void createSound() {
        System.out.println("\n(Sound) WHOO Ooo WHOOo ooo WHOOo ooo\n");
    }
    public void alertPolice() {
        System.out.println("*Calling 112*");
    }
    // ... the class continues, implementing each method belonging to the typestate
```

Listing 2.23: HouseAlarm's client example.

```
public class Client {
    public static final long KEY = 123;

    public static void main(String[] args) {
        long key = 111;
        HouseAlarm alarm = setup();
        alarm.activate();

        if (alarm.isValid()) {
             alarm = exec(alarm, key);
        }

        alarm.shutdown();
        System.out.println("Done!");
    }

    @Ensures("Disabled")
    private static HouseAlarm setup() {
        HouseAlarm alarm = new HouseAlarm();
        alarm.loadSensors();
        alarm.addSensor(new Sensor());
        alarm.configSensitivity(10.2f);
        alarm.done();
        return alarm;
    }

    @Ensures("Disabled")
    private static HouseAlarm exec(@Requires("Active") HouseAlarm alarm, long key) {
        while(!alarm.detectMotion()) {
            if (alarm.disable(123)) {
                return alarm;
            }

    // ... the class continues
```

2.2.2.2 Virus Detector

The virus detector prototype was created to increase the HouseAlarm's difficulty, describing a more complex system with more states than the initial prototype.

Listing 2.24: HouseAlarm's typestate.

```
// VirusDetection represents a virus detection system in different network partitions
//              (instead of a regular anti-virus)
typestate VirusDetection {
    Start = {
        void loadSignature(): Start,
        void loadKeys(): Start,
        boolean boot(): <true: Scanning, false: Start>,
        void turnOff(): Shutdown
    }
    Shutdown = {
        void dumpMemory(): end
    }

    Scanning = {
        boolean scanPartition(): <true: VirusDetected, false: Scanning>,
        void turnOff(): Shutdown
    }
    VirusDetected = {
        void isolate(): Quarantine,
        void analyseVirus(): VirusAnalysis
    }
    VirusAnalysis = {
        void isolate(): Quarantine,
        String generateTrace(): Scanning
    }
    ManualReview = {
        // Receives enum of status (Safe, Unimportant, Critical...)
        void setStatus(SafetyStatus): ManualReview,
        char[] getReport(): ManualReview,
        SafetyStatus getStatus(): <SAFE: ManualReview, UNIMPORTANT: ManualReview,
                                                              CRITICAL: ManualReview>,

        // Evaluates status and goes to the next stage (true -> safe, false -> locked)
        boolean evaluate(): <true: Scanning, false: Locked>
    }

    Quarantine = {
        void removeVirus(): Scanning,
        boolean manualReview(long): <true: ManualReview, false: Locked>,
        void lockSystem(): Locked
    }
    Locked = {
        // Receives admin ID
        boolean resume(long): <true: Scanning, false: Locked>
    }
}
```

Listing 2.25: VirusDetection class, implementing its associated typestate methods.

```
@Typestate("VirusDetectionProtocol")
public class VirusDetection {
    @Nullable
    private Virus foundVirus;

    @Nullable
    public Virus getFoundVirus() {
        return foundVirus;
    }

    public void loadSignature() {}
    public void loadKeys() {}
    public boolean boot() { return true; }

    public void turnOff() {}
    public void dumpMemory() {}

    // ... the class continues, implementing the remaining methods
    //          belonging to its typestate
```

Listing 2.26: A section of the client example for testing the VirusDetection typestate.

```
public class Client {
    public static void main(String[] args) {
        VirusDetection detection = init();
        long adminKey = 1287392;

        if (detection.boot()) {
            exec(detection, adminKey);
            return;
        }

        detection.turnOff();
        end(detection);
        System.out.println("Invalid␣security␣credentials!");
    }

    @Ensures("Start")
    private static VirusDetection init() {
        VirusDetection detection = new VirusDetection();
        detection.loadKeys();
        detection.loadSignature();
        return detection;
    }

    private static void exec(@Requires("Scanning") VirusDetection detection, long key) {
        detection = findVirus(detection);
        detection.analyseVirus();
        detection.generateTrace();

        // ... the method's body continues ...
     }

     // ... the class also continues, adding other important methods to the client
```

2.2.2.3 Robot

The robot example contains three different typestates - Robot, ShooterRobot and StorageRobot. The Robot protocol encodes the basic behaviour for all robot protocols. The ShooterRobot and StorageRobot protocols extend the Robot typestate with new states and functions, exploring JaTyC's new subtyping system [4]. Their Java implementation is displayed in the appendix B. The following listings only illustrate their typestates.

Listing 2.27: The Robot typestate encodes a robot capable of moving to different locations. It also travels to the start position before being turned off.

```
typestate RobotProtocol {
    Idle = {
        void shutdown(): Shutdown,
        void move(int, int): Moving
    }
    Moving = {
        void move(int, int): Moving,
        void advance(): Moving,
        boolean hasArrived(): <true: Idle, false: Moving>,
        void stop(): Idle
    }
    Shutdown = {
        void advance(): Shutdown,
        boolean hasArrived(): <true: end, false: Shutdown>
    }
}
```

Listing 2.28: The ShooterRobot typestate extends RobotProtocol by adding shooting and gun loading functionalities.

```
typestate ShooterRobotProtocol {
    Idle = {
        void shutdown(): Shutdown,
        void move(int, int): Moving,
        boolean scanTarget(): <true: Engage, false: Idle>
    }
    Moving = {
        void move(int, int): Moving,
        void advance(): Moving,
        boolean hasArrived(): <true: Idle, false: Moving>,
        void stop(): Idle
    }
    Shutdown = {
        void advance(): Shutdown,
        boolean hasArrived(): <true: end, false: Shutdown>
    }
    Engage = {
        boolean readyToShoot(): <true: Engage, false: LoadGun>,
        void shoot(): Engage,
        void stop(): Idle
    }
    LoadGun = {
        boolean load(utils.Bullet): <true: Engage, false: LoadGun>
    }
}
```

Listing 2.29: StorageRobot's typestate. a more complicated typestate than ShooterRobot. It extends the original Robot protocol to allow storage robots to move in a grid storage system while picking, returning and dropping different items.

```
typestate StorageRobot {
    Idle = {
        // There could be a queue of orders. This method returns the first
        utils.Order awaitOrders(): <Pick: PickItem, Drop: DropItem, None: Idle>,
        void shutdown(): Shutdown,
        void move(int, int): Moving
    }
    Shutdown = {
        void advance(): Shutdown,
        boolean hasArrived(): <true: end, false: Shutdown>
    }
    Moving = {
        void move(int, int): Moving,
        void advance(): Moving,
        boolean hasArrived(): <true: Idle, false: Moving>,
        void stop(): Idle
    }
    DropItem = {
        // Advance to the next grid position, considering
        //  the closest distance to the goal position
        void advance(): DropItem,
        boolean hasArrived(): <true: Idle, false: DropItem>
    }
    PickItem = {
        // Advance to the next grid position, considering
        //  the closest distance to the item's position
        void advance(): PickItem,
        boolean hasArrived(): <true: ReturnItem, false: PickItem>
    }
    ReturnItem = {
        // Advance to the next grid position, considering
        //  the closest distance to the start position
        void advance(): ReturnItem,
        boolean hasArrived(): <true: Idle, false: ReturnItem>
    }
}
```

3

Related Work

3.1 Finding the related work

The search for related work focused on contributions that address specific aspects of protocol specification and enforcement. Priority was given to research on behavioural models that verify correct usage of class protocols. In addition, attention was directed toward distributed frameworks that facilitate the definition and stacking of distributed protocols, particularly those that delegate concurrency management to internal mechanisms, shielding the developers from many concurrency issues that commonly arise when developing such protocols. Finally, the search also included dynamic monitoring approaches capable of analyzing communication protocols at runtime to detect and log protocol violations.

3.2 Distributed Systems Platforms

3.2.1 Appia

Appia [27] is a protocol kernel that offers a clean and elegant way for an application to express inter-channel constraints. A protocol kernel is a software package that supports the composition and execution of micro-protocols. It is a toolkit for Java, allowing developers to compose stacks of protocols according to the application's needs, thus, making the toolkit flexible and modular, allowing communication stacks to be composed and reconfigured at runtime.

Appia introduces the concepts of channels and sessions, which gives developers more control over the binding of protocols to create different types of services within a single application. Furthermore, the toolkit ensures each protocol that composes the developed distributed system is single-threaded, which incures in non-negligibel performance bottlenecks.

The toolkit also leverages Java's inheritence model, allowing developers to create their own protocols by extending base elements of Appia that control the execution of the developed applications.

3.2.2 Yggdrasil

Yggdrasil [10] is a lightweight and efficient C framework, designed to allow the execution of distributed applications in wireless Ad Hoc scenarios (a decentralized type of network). The framework provides an event-driven and mutli-threaded execution environment that shields the programmer from concurrency issues.

Each protocol is modeled as a state-machine, whose internal state evolves accordingly to the sequential reception and processing of events. Protocols may also generate events. These will either be processed by themselves, or be sent to other protocols.

There four types of events provided by Yggdrasil:

1. **Messages**: the only event that can be transported between different processes. They can be destined to all neighboring nodes (i.e., one-hop broadcast) or to a single neighbor (i.e., point-to-point);

2. **Timers**: these notify the exection of some periodic task, or that a local timeout ocurred;

3. **Requests/Replies**: allow direct *one-to-one* communications between protocols in the same process;

4. **Notifications**: allow direct *one-to-many* interactions between protocols in the same process.

3.2.3 Summary

Among the presented tools, each addresses aspects the other does not fully cover. For example, Appia enforces a single-threaded policy for each protocol and introduces concepts such as channels, giving developers greater control over protocol binding. In contrast, Yggdrasil provides features like timers, messages, and notifications but lacks the channel abstraction Appia offers.

In comparison, Babel combines the strengths of both by enforcing single-threaded protocols and using channels for low-level communication, while also supporting timers, messages, requests/replies, and notifications—thereby delivering a more complete framework.

3.3 Typestates/Session Types

Typestates are a concept, introduced by Strom and Yemini [31], which groups operations that may be safely done in a set of program states. The term comes from the fact that a typestate is associated with an instance of a type, defining sequences of operations that do not make the program go wrong. They are a form of *behavioural types*, similar in expressiveness to other approaches, the most popular inspired in Session Types [20, 35, 33].

Typestate–or protocol–analysis is a form of static program analysis that has proved itself useful from simple program calculi to mainstream programming languages [1, 21, 16, 12]. The key safety property is protocol compliance–client code invoking the operations do so respecting the implicit protocol resulting from the typestate declarations.

Dealing with aliasing is the main challenge, addressed either with ownership permissions in languages like Plaid [32] or Obsidian [9], or with linearity in languages using the Session Types view [29, 26, 28]. Other approaches, like [30] for the language Scala, delegate to classical interference control primitives the management of aliasing.

3.3.1 Typestates in object-oriented languages

Besides the works referred above and in the introduction, namely JaTyC, Mungo, Plaid, and SJ, there are not (as far as we know) other implementations of typestates in mainstream object-oriented programming languages. Only the approaches emanating from [18] incorporate static memory management in the analysis discipline and guarantee memory safety.

Recently, an application to the compositional automatic verification in multi-object systems was presented [19], generically explaining the bottom-up approach (infering typestates from extrated automata-like models of the systems) but not giving details about the language-based analysis. Nevertheless, subtyping or inheritance are not considered.

3.3.2 Typestates subtyping

This aspect is only considered in a handful of works. Plural [5] supports behavioural subtyping via typestate 'refine' annotations to declare substates. Typestates are method and field annotations (like pre-conditions and invariant assertions) and the refinement declarations must be checked. Our approach allows to implicitly derive and automatically check subtyping relations between classes, being as we show herein, language independent.

Building on Plaid and Plural, Obsidian is a language for smart contracts. Since it has safety and security as main concerns, it only features parametric polymorphism.

Mungo associates typestates to classes and to channels used to communicate between distributed objects. A limited form of subtyping is used in both forms of typestates, a form which agrees closely with the seminal subtyping algorithm [17] that we extend in this work.

3.3.3 Typestates in OO languages with inheritance

In addition to comparisons already done in the introduction, as with subtyping, as far as we know, only one other work deal with this key feature. Fugue [11] treats typestates as field invariants and associate the fields of a class in a *frame*. Objects are controlled and protected with collections of frames. Subclassing and casting follow the *safe substitutability*

principle. The formal system is elaborated due to the lack of a single protocol specification for each class.

3.3.4 Typestate tools in modern languages

typestate-rs [1] [13] is a library that adds the support for typestates in Rust, allowing developers to declare statically-checked automata, providing a typestated API supported by Rust's advanced type system.

Typestates are defined using Rust's module structures. The library levarages the use of macros to define the typestate itself, its states and their transitions. There are three main macros to use:

1. **[#typestate]**: specifies the module is a typestate, later to be evaluated by the tool;

2. **[#automaton]**: defines the attached structured is the main one for the defined typestate;

3. **[#state]**: defines the attached structure is as a state belonging to the typestate.

To define transitions between states, **typestate-rs** uses *traits*, a Rust feature for defining the functionality a particular type has and can share with other types. There are three main transition types supported by the tool: *initial*, *final* and *state* transitions. **Initial** transitions are contructors. In other words, they do no take a *self* parameter, and will return a valid state. **Final** transitions are the opposite of the initial ones. They take the *self* parameter and do not return a valid state, thus consuming the typestate according to Rust's borrow rules; **state** transitions are the core for all others. They are functions that take the current state (i.e *self*) and return a new valid state, effectively transitioning between two states.

3.4 Monitors

Session types [34] provide a formal mechanism to ensure correctness in concurrent systems. They enforce linearity in the interaction between two parties that follow a previously agreed session type. Their analysis is usually performed statically, using a type system, before programs are executed. However, full static analysis is not always feasible (e.g. third-party code for external libraries may not be present) [7]. The distributed setting may also impose certain challenges that make such analysis harder. Some processes may be written in languages that do not support static typing of sessions, or may be compromised by a malicious attacker, violating invariants of the session types. For such cases, monitor tools are usually employed. [24]

[1] Typestate-rs's repository: https://github.com/rustype/typestate-rs

3.4.1 STMonitor

STMonitor [2] is a tool that synthesis Scala [3] code of a type-checked monitor, given a session type S. In other words, it creates Scala code, designing a monitor capable of evaluating the behaviour and interactions between a client and a server that follow the protocol S.

STMonitor uses an extended definition for session types - probabilistic session types - allowing runtime monitoring of quantitive behaviour for two interacting components. Each branch in choice of a probabilistic session type can have a probability associated to it. STMonitor also assumes the sequence of interactions (exchanged messages) observed up to the current point of execution is a sample of the larger population. With so, the monitoring tool can use a *frequentist* approach. Firstly, it calculates the confidence intervals (CIs) around each desired probablity in a session type S (i.e., for each branch in a choice belonging to S) , giving an aproximation of the expected probabilistic behaviour based on the *sample size* and *confidence level* ($0 \leq \ell < 1$). Then, it compares whether the observed frequency (i.e., the number of times a branch of a choice c was chosen in all occurrences of c) of messages matches the expected confidence-levels, giving warnings about interactions whose observed frequencies are not contained by the created CIs. [8]

```
1  def sendChoice1(srv: In[Choice1], Client: ConnManager): Unit = {
2    srv ? {
3      case msg @ Success(_) =>
4        if (validateId(msg.id, payloads.Login.uname)) {
5          Client.send(msg)
6          /* Continue according to R */
7        } else {
8          /* log and halt: Sending incorrect values. */
9        }
10     case msg @ Retry() =>
11       Client.send(msg)
12       receiveLogin(msg.cont, Client)
13 } }
```

Figure 3.1: An example of synthesized code created by STMonitor. The image was taken from [7].

[2] STMonitor's repository: https://github.com/chrisbartoloburlo/stmonitor
[3] Scala's website https://www.scala-lang.org/

4

Methodology

4.1 Objectives

There are two main objectives to be accomplished. Firstly, it is desired to use and explore JaTyC as much as possible to specify class usage protocols, guarantee these are properly followed, and prevent any existing memory errors - this can be considered to be an **internal** objective. Secondly, as an **external** objective, verify whether the exchanged messages are sound, and log information regarding those that are not. Messages are considered to be sound when the receiver's protocol can react and respond to them, considering its current state. In other words, an object associated to a protocol can be used and may transition to a different state upon receiving the incoming message.

Combining both tools directly, to statically verify the usage of Babel protocols, is not a trivial task. Firstly, when working with Babel, one may use external libraries whose source code is inaccessible. Secondly, Babel's dymamic, event-driven nature implies its procols' usage is reflected through the order at which its event-handlers are called. Associating a typestate to a Babel protocol becomes irrelevant, as its functions, the protocol's event-handlers, are not directly called in the code. They are callbacks for events stored in an event-queue within Babel's core module, which will only be visible at runtime.

To this end, it is desired to have a monitor system, capable of handling dynamic and reactive environments, common in distributed systems. To accomplish the described objectives, the tool is expected to support many new features, creating a new version of JaTyC, useful for swarm computing frameworks. Firstly, JaTyC's parser should be used as a baseline, using an extended version of the current JaTyC's syntax; Secondly, the tool should compile the new syntax to Java code, creating wrapper classes for the user's Babel protocols - these can be called **Monitors**. Monitors should define the state-machines for their internal Babel protocols. They should represent their associated typestates, ensure they are properly followed and guarantee their objects are eventually consumed (in other words, they can reach the **end** state and get their resources freed) [28]. These features can be considered as means of reaching the **internal** objective. For the **external** objective, the tool should contain a set of logging features, such that erroneous messages can get logged

in different logging levels.

With this foundation, it is expected to have a tool capable of evaluating behaviour models, while being robust in the dynamic environments swarm computing frameworks are suitable for. The monitor system will support new vital features, allowing a runtime verification of typestates associated to Babel protocols, satisfying both previously stated objectives, internal and external.

4.1.1 The Tool

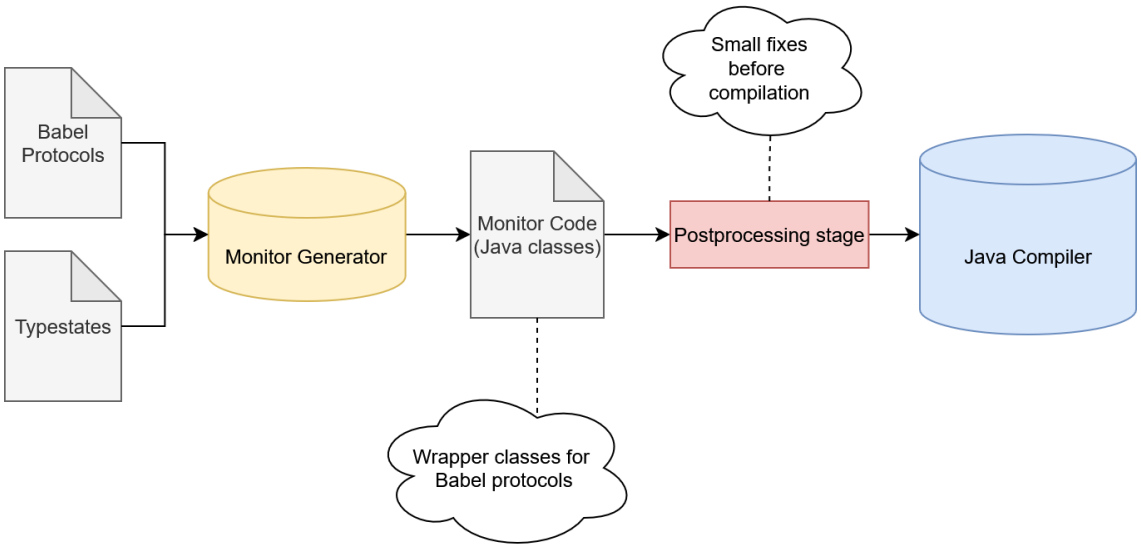


Figure 4.1: The execution flow of the tool.

4.1.1.1 Monitor Generator

The generator should use the current JaTyC's parser as a foundation for the new features it will support. The current parser already creates an AST, followed by a graph representing the state-machine for the parsed typestate - vital features for the monitor generation procedure. The compiler should be capable of parsing typestate definitions and generate Java code. For each Babel protocol associated to a typestate, the compiler should generate a monitor class, wrapping the current Babel protocol. The monitor, using the related typestate, will then contain the same event handlers present in the original protocol, adding state validations before calling the original event handler; and adding state transtions and other state related procedures after invoking the original handler. This means that typestates associated to Babel protocols require the corresponding event handlers written in their states.

The typestate syntax should also be extended. Distributed software usually requires coordination and communication between different peers in a network (e.g. quorum systems [2]). Systems that require coordination, depending on the network size, may require several messages to be processed before deciding a value or action [14]. The

current typestate syntax is not expressive enough to define such conditions. Since the monitor is meant fo swarm computing systems, it is vital to have quantity conditions in the syntax, allowing a finer control over distributed protocols' typestates.

To this purpose – this is, to support quantity conditions – the work done with STMonitor 3.4.1 and its probabilisitic session types are going to be of great influence. It is proposed an extension to the current JaTyC's typestate syntax, to support different probabilities for each function call in a state. Each probability represents its associated method's triggering probability (or ratio within the state). The tool can then, given a confidence level ℓ by the user, build confidence intervals (CIs) around each expected probability. When monitoring, it tracks each event-handler's invocation frequency, checking whether they respect their corresponding CIs, thereby, validating that the runtime behaviour of protocols in a given state aligns with the expected probabilistic distribution.

4.1.1.2 Postprocessing

The compiler for monitor generation, by itself, is not enough to make the monitor system work. It is assumed the programmer may use and create Babel protocols, without considering the new monitor tool. As a safety precaution, the created Babel protocols should be inspected, as they may contain lines of code registering their own event handlers to the Babel's core module. These lines should be removed, if they exist, finally registering the monitor's event handlers in the monitor's classes instead. Once the postprocessing procedure is finished, the code is ready to be compiled by Java's native compiler.

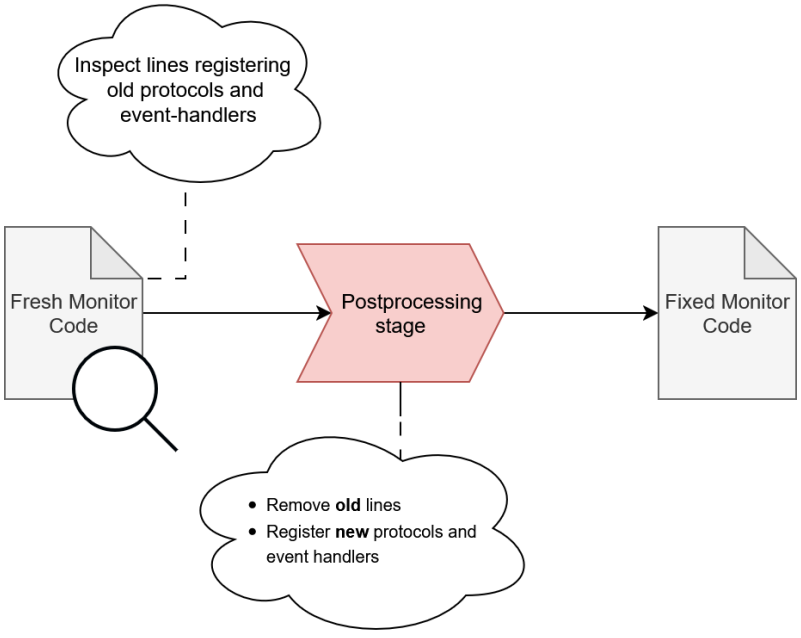


Figure 4.2: Postprocessing scheme.

4.2 Deliverables

1. A compiler tool for protocol monitor generation, creating a monitor system
2. GitHub repository with documentation (with MIT licence)
3. Example projects

4.3 Work Plan

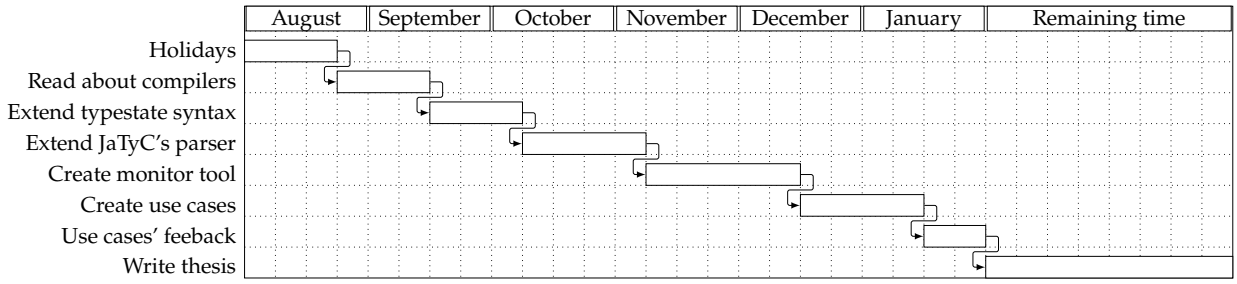
1. Read and learn more about compilers and language theory

   References: "The Definitive ANTLR 4 Reference" [1] and "Modern Compiler Implementation in Java" [2]

   Expected outcome: A deeper understanding of LL grammars, parsers, and parser generators, as well as insights into code generation for multiple languages. Develop a comprehensive understanding of compiler architecture, covering lexical analysis, parsing, semantic checks, optimization, and code generation, all within the context of modern Java implementations.

2. Develop and specify a more expressive language for the current typestate definitions, supporting quantity conditions using expected probabilies for each function call
3. Extend/use JaTyC's parser to support the extended typestate syntax, creating new Java classes (monitors)
4. Create the monitor tool with different logging features
5. Create different use cases
6. Write the thesis

4.3.1 Cronogram



[1] "The Definitive ANTLR 4 Reference" book webpage: https://pragprog.com/titles/tpantlr2/the-definitive-antlr-4-reference/

[2] The "Modern Compiler Implementation in Java" book webpage: https://www.cambridge.org/core/books/modern-compiler-implementation-in-java/34EACED718B1D6D5237705F9BFD7CD4A

Bibliography

[1] D. Ancona et al. "Behavioral Types in Programming Languages". In: *Found. Trends Program. Lang.* 3.2-3 (2016), pp. 95–230. DOI: 10.1561/2500000031 (cit. on p. 29).

[2] H. Attiya, A. Bar-Noy, and D. Dolev. "Sharing Memory Robustly in Message-Passing Systems". In: *J. ACM* 42.1 (1995), pp. 124–142. DOI: 10.1145/200836.200869. URL: https://doi.org/10.1145/200836.200869 (cit. on p. 33).

[3] L. Bacchiani et al. "A Java typestate checker supporting inheritance". In: *Sci. Comput. Program.* 221 (2022), p. 102844. DOI: 10.1016/J.SCICO.2022.102844. URL: https://doi.org/10.1016/j.scico.2022.102844 (cit. on pp. 15, 16).

[4] L. Bacchiani et al. "Behavioural Up/down Casting For Statically Typed Languages". In: *38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria*. Ed. by J. Aldrich and G. Salvaneschi. Vol. 313. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 5:1–5:28. DOI: 10.4230/LIPICS.ECOOP.2024.5. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2024.5 (cit. on p. 25).

[5] K. Bierhoff and J. Aldrich. "Modular typestate checking of aliased objects". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007*. ACM, 2007, pp. 301–320. DOI: 10.1145/1297027.1297050 (cit. on p. 29).

[6] K. P. Birman and T. A. Joseph. "Reliable Communication in the Presence of Failures". In: *ACM Trans. Comput. Syst.* 5.1 (1987), pp. 47–76. DOI: 10.1145/7351.7478. URL: https://doi.org/10.1145/7351.7478 (cit. on p. 1).

[7] C. B. Burlò, A. Francalanza, and A. Scalas. "Towards a Hybrid Verification Methodology for Communication Protocols (Short Paper)". In: *Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*.

Ed. by A. Gotsman and A. Sokolova. Vol. 12136. Lecture Notes in Computer Science. Springer, 2020, pp. 227–235. DOI: 10.1007/978-3-030-50086-3_13. URL: https://doi.org/10.1007/978-3-030-50086-3_13 (cit. on pp. 30, 31).

[8] C. B. Burlò et al. "Towards Probabilistic Session-Type Monitoring". In: *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*. Ed. by F. Damiani and O. Dardha. Vol. 12717. Lecture Notes in Computer Science. Springer, 2021, pp. 106–120. DOI: 10.1007/978-3-030-78142-2_7. URL: https://doi.org/10.1007/978-3-030-78142-2_7 (cit. on p. 31).

[9] M. J. Coblenz et al. "Obsidian: Typestate and Assets for Safer Blockchain Programming". In: *ACM Trans. Program. Lang. Syst.* 42.3 (2020), 14:1–14:82. DOI: 10.1145/3417516 (cit. on p. 29).

[10] P. Á. Costa, A. Rosa, and J. Leitão. "Enabling wireless ad hoc edge systems with yggdrasil". In: *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*. Ed. by C. Hung et al. ACM, 2020, pp. 2129–2136. DOI: 10.1145/3341105.3373908. URL: https://doi.org/10.1145/3341105.3373908 (cit. on p. 28).

[11] R. DeLine and M. Fähndrich. "Typestates for Objects". In: *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Proceedings*. Ed. by M. Odersky. Vol. 3086. Lecture Notes in Computer Science. Springer, 2004, pp. 465–490. DOI: 10.1007/978-3-540-24851-4_21. URL: https://doi.org/10.1007/978-3-540-24851-4_21 (cit. on p. 29).

[12] J. Duarte and A. Ravara. "Taming stateful computations in Rust with typestates". In: *J. Comput. Lang.* 72 (2022), p. 101154. DOI: 10.1016/J.COLA.2022.101154 (cit. on p. 29).

[13] J. Duarte and A. Ravara. "Taming stateful computations in Rust with typestates". In: *J. Comput. Lang.* 72 (2022), p. 101154. DOI: 10.1016/J.COLA.2022.101154. URL: https://doi.org/10.1016/j.cola.2022.101154 (cit. on p. 30).

[14] M. J. Fischer, N. A. Lynch, and M. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *J. ACM* 32.2 (1985), pp. 374–382. DOI: 10.1145/3149.214121. URL: https://doi.org/10.1145/3149.214121 (cit. on pp. 1, 33).

[15] P. Fouto et al. "Babel: A Framework for Developing Performant and Dependable Distributed Protocols". In: *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*. 2022, pp. 146–155. DOI: 10.1109/SRDS55811.2022.00022 (cit. on p. 4).

[16] R. Garcia et al. "Foundations of Typestate-Oriented Programming". In: *ACM Transactions on Programming Languages and Systems* 36.4 (2014), p. 12. DOI: 10.1145/2629609 (cit. on p. 29).

[17] S. J. Gay and M. Hole. "Types and Subtypes for Client-Server Interactions". In: *Proc. of Programming Languages and Systems (ESOP)*. Vol. 1576. Lecture Notes in Computer Science. Springer, 1999, pp. 74–90. DOI: 10.1007/3-540-49099-X_6 (cit. on p. 29).

[18] S. J. Gay et al. "Modular session types for distributed object-oriented programming". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*. ACM, 2010, pp. 299–312. DOI: 10.1145/1706299.1706335 (cit. on p. 29).

[19] R. Grigore, D. Distefano, and N. Tzevelekos. "Automatic Compositional Checking of Multi-object TypeState Properties of Software". In: *Principles of Verification: Cycling the Probabilistic Landscape - Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part I*. Ed. by N. Jansen et al. Vol. 15260. Lecture Notes in Computer Science. Springer, 2024, pp. 3–40. DOI: 10.1007/978-3-031-75783-9_1 (cit. on p. 29).

[20] K. Honda, V. T. Vasconcelos, and M. Kubo. "Language primitives and type discipline for structured communication-based programming". In: *Programming Languages and Systems*. Ed. by C. Hankin. Vol. 1381. Lect. Notes Comput. Sci. Springer, 1998, pp. 122–138. DOI: 10.1007/BFb0053567 (cit. on p. 28).

[21] H. Hüttel et al. "Foundations of Session Types and Behavioural Contracts". In: *ACM Comput. Surv.* 49.1 (2016), 3:1–3:36. DOI: 10.1145/2873052 (cit. on p. 29).

[22] R. Ingram et al. "A leader election algorithm for dynamic networks with causal clocks". In: *Distributed Comput.* 26.2 (2013), pp. 75–97. DOI: 10.1007/S00446-013-0184-1. URL: https://doi.org/10.1007/s00446-013-0184-1 (cit. on p. 1).

[23] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen. "Session types for Rust". In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015*. Ed. by P. Bahr and S. Erdweg. ACM, 2015, pp. 13–22. DOI: 10.1145/2808098.2808100. URL: https://doi.org/10.1145/2808098.2808100 (cit. on p. 15).

[24] L. Jia, H. Gommerstadt, and F. Pfenning. "Monitors and blame assignment for higher-order session types". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by R. Bodík and R. Majumdar. ACM, 2016, pp. 582–594. DOI: 10.1145/2837614.2837662. URL: https://doi.org/10.1145/2837614.2837662 (cit. on p. 30).

[25] K. Kaur and Y. Kumar. "Swarm Intelligence and its applications towards Various Computing: A Systematic Review". In: *2020 International Conference on Intelligent Engineering and Management (ICIEM)*. 2020, pp. 57–62. DOI: 10.1109/ICIEM48762.2020.9160177 (cit. on p. 1).

[26] D. Kouzapas et al. "Typechecking protocols with Mungo and StMungo". In: *Proc. of Principles and Practice of Declarative Programming (PPDP)*. ACM, 2016, pp. 146–159. DOI: 10.1145/2967973.2968595 (cit. on p. 29).

[27] H. Miranda, A. S. Pinto, and L. E. T. Rodrigues. "Appia: A Flexible Protocol Kernel Supporting Multiple Coordinated Channels". In: *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 2001), Phoenix, Arizona, USA, April 16-19, 2001*. IEEE Computer Society, 2001, pp. 707–710. DOI: 10.1109/ICDSC.2001.919005. URL: https://doi.org/10.1109/ICDSC.2001.919005 (cit. on p. 27).

[28] J. Mota, M. Giunti, and A. Ravara. "Java Typestate Checker". In: *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*. Ed. by F. Damiani and O. Dardha. Vol. 12717. Lecture Notes in Computer Science. Springer, 2021, pp. 121–133. DOI: 10.1007/978-3-030-78142-2_8. URL: https://doi.org/10.1007/978-3-030-78142-2_8 (cit. on pp. 14–16, 29, 32).

[29] N. Ng et al. "Safe Parallel Programming with Session Java". In: *Coordination Models and Languages - 13th International Conference, COORDINATION 2011. Proceedings*. Ed. by W. D. Meuter and G. Roman. Vol. 6721. Lecture Notes in Computer Science. Springer, 2011, pp. 110–126. DOI: 10.1007/978-3-642-21464-6_8 (cit. on p. 29).

[30] A. Scalas and N. Yoshida. "Lightweight Session Programming in Scala". In: *30th European Conference on Object-Oriented Programming, ECOOP 2016*. Ed. by S. Krishnamurthi and B. S. Lerner. Vol. 56. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 21:1–21:28. DOI: 10.4230/LIPICS.ECOOP.2016.21 (cit. on p. 29).

[31] R. E. Strom and S. Yemini. "Typestate: A programming language concept for enhancing software reliability". In: *IEEE Transactions on Software Engineering* SE-12.1 (1986), pp. 157–171. ISSN: 0098-5589. DOI: 10.1109/TSE.1986.6312929 (cit. on p. 28).

[32] J. Sunshine et al. "Changing state in the plaid language". In: *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011*. Ed. by C. V. Lopes and K. Fisher. ACM, 2011, pp. 37–38. DOI: 10.1145/2048147.2048166 (cit. on p. 29).

[33] V. T. Vasconcelos. "Fundamentals of session types". In: *Information and Computation* 217 (2012), pp. 52–70. DOI: 10.1016/j.ic.2012.05.002 (cit. on p. 28).

[34] V. T. Vasconcelos. "Fundamentals of session types". In: *Inf. Comput.* 217 (2012), pp. 52–70. DOI: 10.1016/J.IC.2012.05.002. URL: https://doi.org/10.1016/j.ic.2012.05.002 (cit. on p. 30).

[35] V. T. Vasconcelos. "Sessions, from Types to Programming Languages". In: *Bull. EATCS* 103 (2011), pp. 53–73. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/136 (cit. on p. 28).

[36] P. Wadler. "Linear Types can Change the World!" In: *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*. Ed. by M. Broy and C. B. Jones. North-Holland, 1990, p. 561 (cit. on p. 2).

A

Babel Examples in Pseudo-Code

It is assumed all networking state is previously set up. As a result, it is ignored in the following figures. Terms such as *leader*, *myself*, *sender*, *receiver*, or *peers*, may be used throughout the provided pseudo-code.

A.1 Alternating Bit Protocol Pseudo-Code

A summary on the algorithm's behaviour can be found here 2.1.2.1.

Algorithm 1 Alternating Bit Protocol

```
1:  State :
2:      toSendBit
3:      lastAckBit
4:
5:      Upon Init do:
6:          lastAckBit ← 1
7:          toSendBit ← 0
8:           // only the "leader" can do this
9:          Setup Periodic Timer BitTimer(t)
10:
11:     Upon Receiver (BitSend, s, bit) do:
12:         If lastAckBit ≠ bit then
13:             lastAckBit ← bit
14:             Trigger Send (BitAck, s, bit)
15:         Else
16:             Trigger Send (BitAck, s, lastAckBit)
17:
18:     Upon Receiver (BitAck, s, bit) do:
19:         If bit = toSendBit then
20:             If toSendBit = 0 then
21:                 toSendBit ← 1
22:             Else If toSendBit = 1 then
23:                 toSendBit ← 0
24:
25:     Upon Timer BitTimer(t) do:
26:         Trigger Send (BitSend, receiver, toSendBit)
27:
28:
```

A.2 Number Set Protocol Pseudo-Code

A summary on the algorithm's behaviour can be found here 2.1.2.2.

Algorithm 2 Number Set Protocol

```
1:  Interface :
2:      Indications :
3:          getRandomFrom(set)  // Returns a random value from the given set
4:          randomUniqueSet()
5:
6:
7:  State :
8:      peers  // Set of in the network
9:      numbSet  // Set of numbers
10:     session  // Maps peers to a set of messages to be acknowledged
11:     uniqueNumbers  // List of predefined unique numbers
12:     Upon Init do:
13:         session ← {}
14:         numbSet ← {}
15:         uniqueNumbers ← randomUniqueSet()
16:          // only the "leader" can do this
17:         Setup Periodic Timer SendTimer(t)
18:
19:
20:      // "numbers" is the set containing the numbers to be updated
21:      //  (old numbers that have not be acknowledged and the new one)
22:
23:     Upon Receiver (SetUpdate, s, numbers) do:
24:         numbSet ← numbSet ∪ numbers
25:         Trigger Send (SetAck, s, numbers)
26:
27:
28:     Upon Receiver (SetAck, s, numbers) do:
29:         session[s] ← session[s] \ numbers
30:
31:
32:     Upon Timer SendTimer(t) do:
33:         If |session| = 0 ∧ |uniqueNumbers| = 0 then
34:             Cancel Timer SendTimer
35:             Return
36:         update ← ⊥
37:         If  |uniqueNumbers| ≥ 0 then
38:             update ← getRandomFrom(uniqueNumbers)
39:             uniqueNumbers ← uniqueNumbers \ {update}
40:             numbSet ← numbSet ∪ {update}
41:         For each p ∈ peers do:
42:             session[p] ← session[p] ∪ {update}
43:             Trigger Send (SetUpdateMessage, p, session[p])
44:
45:
46:
```

A.3 Bit Vote Protocol Pseudo-Code

A summary on the algorithm's behaviour can be found here 2.1.2.3. W.I.P

Algorithm 3 Bit Vote Protocol (Part 1)

```
1:  Interface :
2:      Indications :
3:          randVote() // Generates a random number between 0 and 1 inclusive.
4:          randId() // Generates a random string id.
5:          decide(ackSet) // Decides the value with the biggets population. "ackSet" contains acked votes (0 or 1)
6:          hasQuorum(ackSet) // Checks whether a maj-quorum has been reached
7:          isAcksFull(ackSet) // Checks whether alls acks have been received
8:
9:
10: State :
11:     peers // Peers in the network
12:     voteRounds // Map of rounds - round: (id, votedBit, retries, acks, status)
13:     voteHistory // Map of records - record: (id, votedBit, status). A history of vote proposals.
14:     currentRountID // String. IDs are generated randomly.
15:
16:     Upon Init do:
17:         // Only the leader should do this
18:         history ← {}
19:         currentRoundId ← "empty-string"
20:         Setup Periodic Timer VoteTimer(t)
21:
22:     Upon Receiver (VoteRequest, s, roundId) do:
23:         voteRecord ← ⊥
24:         If voteHistory[roundId]] = ⊥ then
25:             voteRecord ← (roundId, randVote(), "PENDING")
26:             voteHistory[roundId] ← voteRecord
27:         Else
28:             voteRecord ← voteHistory[roundId]
29:         (id, votedBit, status) ← voteRecord
30:         Trigger Send (VoteAck, s, roundId, votedBit)
31:
32:     Upon Receiver (VoteAck, s, roundId, votedBit) do:
33:         If voteRounds[roundId] ← ⊥ then Return
34:         (id, votedBit, retries, acks, status) ← voteRounds[roundId]
35:         voteRounds[roundId] ← (id, votedBit, retries, acks ← {votedBit})
36:
37:     Upon Receiver (VoteWriteBack, s, roundId, decidedBit, finalStatus) do:
38:         voteRecord ← ⊥
39:         If voteHistory[roundId] ← ⊥ then
40:             voteRecord ← (id, -1, "PENDING")
41:             voteHistory[roundid] ← voteRecord
42:         Else
43:             voteRecord ← voteHistory[roundid]
44:         (id, votedBit, retries, acks, status) ← voteRecord
45:         voteRecord ← (id, votedBit ← decidedBit, retries, acks, status)
46:         If finalStatus = "FAIL" then
47:             voteRecord ← (id, votedBit, retries, acks, status ← "FAIL")
48:         (histId, bit, retries, ack, histStatus) ← voteRounds[roundId]
49:         voteRounds[roundId] ← (histId, decidedBit, retries, ack, status)
50:
51:
```

Algorithm 4 Bit Vote Protocol (Part 2)

```
1:  Interface :
2:      Indications :
3:          randVote() // Generates a random number between 0 and 1 inclusive.
4:          randId() // Generates a random string id.
5:          decide(ackSet) // Decides the value with the biggets population. "ackSet" contains acked votes (0 or 1)
6:          hasQuorum(ackSet) // Checks whether a maj-quorum has been reached
7:          isAcksFull(ackSet) // Checks whether alls acks have been received
8:
9:
10: State :
11:     peers // Peers in the network
12:     voteRounds // Map of rounds - round: (id, votedBit, retries, acks, status)
13:     voteHistory // Map of records - record: (id, votedBit, status). A history of vote proposals.
14:     currentRountID // String. IDs are generated randomly.
15:
16:
17:     // (continuing the protocol...)
18:
19:     Upon Timer VoteTimer(t) do:
20:         voteRound ← voteRounds[currenRoundId]
21:         (id, votedBit, retries, acks, status) ← voteRound
22:         If voteRound = ⊥ ∨ status ≠ "PENDING" then
23:             voteRound ← (randId(), -1, 3, 0, "PENDING")
24:         Call evaluate(voteRound)
25:         Call retry(voteRound)
26:
27:         // refresh state and broadcast vote (current or new)
28:         (id, votedBit, retries, acks, status) ← voteRound
29:         currentRoundId ← id
30:         voteRounds[currentRoundId] ← {voteRound}
31:         If status = "PENDING" then
32:             Trigger Send (VoteRequest, peers, currentRoundId)
33:         Else
34:             Trigger Send (VoteWriteBack, peers, currentRoundId, decide(acks), status)
35:
36:     // Evaluates the state of a round
37:     Procedure evaluate(voteRound)
38:         (id, votedBit, retries, acks, status) ← voteRound
39:         If isAcksFull(acks) ∧ (hasQuorum(acks) ∨ retries <= 0) then
40:             voteRound ← (id, votedBit, retries, acks, "SUCCESS")
41:         Else If retries <= 0 then
42:             voteRound ← (id, votedBit, retries, acks, "FAIL")
43:
44:     // Removes a retry from a round
45:     Procedure retry(voteRound)
46:         (votedBit, retries, acks, status) ← voteRound
47:         voteRound ← (votedBit, retries − 1, acks, status)
48:
49:
```

B

JaTyC Examples

B.1 Robot class

Its typestate can be found here 2.27.

Listing B.1: Generic robot class for the RobotProtocol typestate.

```
@Typestate("RobotProtocol")
public class Robot {
    public void shutdown() {}
    public void move(int x, int y) {}
    public void advance() {}
    public void stop() {}
    public boolean hasArrived() { return true; }
}
```

B.2 StorageRobot

Its typestate can be found here 2.29.

Listing B.2: StorageRobot class for the StorageRobot typestate, extending the Robot class. As it can be seen, although the StorageRobot typestate is more complicated than ShooterRobot, it mostly adds new states, creating a more intricate behaviour. It only extends its original protocol with one new method.

```
@Typestate("StorageRobotProtocol")
public class StorageRobot extends Robot {
    public Order awaitOrders() { return Order.Pick; }
}
```

B.3 ShooterRobot class

Its typestate can be found here 2.28.

Listing B.3: ShooterRobot class for the ShooterRobot typestate, extending the Robot class. As it can be seen, the ShooterRobot protocol extends its orginal not only with different states, but with many new methods as well.

```
@Typestate("ShooterRobotProtocol")
public class ShooterRobot extends Robot {
    @Nullable
    private Bullet bullet;

    public boolean scanTarget() {
        return true;
    }
    public boolean load(Bullet bullet) {
        this.bullet = bullet;
        return true;
    }
    public boolean readyToShoot() {
        return bullet != null;
    }
    public void shoot() {
        bullet = null;
        System.out.println("(SOUND) Boom!!");
    }
}
```

B.4 Robot example

Listing B.4: An example using the Robot typestate in Java.

```
public class RobotExample implements Example {

    @Override
    public void execute() {
        System.out.println("Starting Robot...");
        Robot robot = new Robot();
        Vector2Int goal = new Vector2Int(2,5);

        robot = RobotController.moveTo(robot, goal);
        robot = RobotController.shutdown(robot);
    }
}
```

B.5 ShooterRobot example

Listing B.5: An example using the ShooterRobot typestate in Java.

```
public class ShooterRobotExample implements Example {
    public ShooterRobotExample() {
        super();
    }
    @Override
    public void execute() {
        System.out.println("Starting␣ShooterRobot...");
        ShooterRobot robot = new ShooterRobot();

        Vector2Int goal = Vector2Int.createRandom(10, 10);
        robot = ShooterController.moveTo(robot, goal);
        robot = ShooterController.scanAndEngage(robot);
        robot = ShooterController.shutdown(robot);
    }
}
```

B.6 StorageRobot example

Listing B.6: An example using the StorageRobot typestate in Java.

```
public class StorageRobotExample implements Example {
    @Override
    public void execute() {
        System.out.println("Starting␣StorageRobot...");
        StorageRobot robot = new StorageRobot();

        boolean isRunning = true;
        while (isRunning) {
            robot = StorageController.awaitOrder(robot);
            isRunning = false;
        }
        robot = StorageController.shutdown(robot);
    }
}
```

B.7 Babel protocol typestate example

Listing B.7: An example of a typestate for a Babel protocol. The typestate may contain more states. It primarily illustrates the grouping of event-handlers

```
typestate MyBabelProtocol {
        ...
        Idle = {
                void: uponVoteStart(): Voting,
                drop: end
        }
        Voting = {
                void: uponVoteAck(): Voting,
                void: uponWriteBack(): Idle
    }
    ...
}
```