AkkaRef: Actor model refactoring into typed actors

Mantas Zdanavičius

AkkaRef: Actor model refactoring into typed actors

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Mantas Zdanavičius
born in Kaunas, Lithuania

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

AkkaRef: Actor model refactoring into typed actors

Author: Mantas Zdanavičius
Student id: 5377684

Abstract

This thesis addresses the challenge of refactoring untyped actor systems into typed ones, particularly within the Scala ecosystem using Akka framework [48]. The actor model, with its message-passing architecture, offers a solution to concurrency and scalability issues in distributed systems, but transitioning from untyped Akka Classic to typed Akka actors is complex and error-prone task. This work proposes a solution through the development of a communication flow graph that captures actor interactions and message exchanges, which is then used to automate the refactoring process while preserving system behavior and communication integrity.

Key contributions include the implementation of the communication flow graph and an automated refactoring tool that translates untyped actor systems to typed ones. The evaluation demonstrates that while the refactored systems maintain communication patterns and functionality, they may introduce a slight reduction in maintainability. Additionally, the resulting effectiveness of the refactored benchmarks varies depending on the complexity of the underlying system.

Lastly, this research sets the foundation for improving refactoring tooling aimed at actor systems, extending the usability of the communication flow graph for the automated documentation of actor systems, and highlights the need to revisit traditional software quality metrics to suit actor-based systems.

Thesis Committee:

Chair:	Dr. B. K. Özkan, Faculty EEMCS, TU Delft
University supervisor:	Dr. B. K. Özkan, Faculty EEMCS, TU Delft
Committee Member:	Dr. R. V. Prasad, Faculty EEMCS, TU Delft

Preface

This thesis wasn't the subject I had originally imagined to do when starting my MSc studies, yet it turned out to be the most impactful challenge in my academic journey. My initial interests lay elsewhere, but the unique concurrency model of this topic drew me in. At first I didn't realize the scope of this topic, seeing only a small part of it, but it soon became clear how deeply it was related with topics static code analysis, concurrent communication patterns, and the secrets of code intermediate representations.

At the start of this journey, my experience with the actor model and the Scala programming language was limited to vague familiarity, and my exposure to functional programming was minimal. This thesis became not only the most challenging step in my journey toward my MSc but also the one that sparked the greatest growth. This research demanded perseverance, technical skills, and experiences that was initially lacking, testing both my technical limits and my capacity to solve problems. Through this, I grew not only as a researcher but as an individual, learning more about myself, my surroundings, and issues I hadn't known existed.

I owe much of this growth to the people who supported me along the way. My supervisor, Burcu K. Özkan, deserves heartfelt thanks for her invaluable guidance, insight, and ideas. To my family and friends, your encouragement and understanding helped me push through when the path seemed most difficult. And to my colleagues for the suggestions and for providing laughter along the way. Without each of you, I'm not sure I'd have completed this journey.

Mantas Zdanavičius
Delft, the Netherlands
October 28, 2024

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
- 1.1 Problem Description . . . 2
  - 1.1.1 Research Question(s) . . . 3

2 Overview 5
- 2.1 Overview of Refactoring . . . 5
- 2.2 Information extraction . . . 6
  - 2.2.1 Communication Flow Graph . . . 7

3 Implementation 9
- 3.1 Definitions . . . 9
  - 3.1.1 Data structures . . . 11
- 3.2 Technical assumptions . . . 15
- 3.3 Execution Overview . . . 16
- 3.4 Static Analysis . . . 17
  - 3.4.1 Points-to lists . . . 18
  - 3.4.2 Class Dependency Graph . . . 19
  - 3.4.3 Call Graph . . . 22
  - 3.4.4 Actor class analysis . . . 27
  - 3.4.5 Identifying Message Types . . . 33
- 3.5 Refactoring . . . 34
  - 3.5.1 Actor Type Resolution . . . 35
  - 3.5.2 Message Type Definition . . . 39
  - 3.5.3 Actor Class Refactoring . . . 41
- 3.6 Other attempts . . . 43
- 3.7 Constraints and considerations . . . 44
  - 3.7.1 Limitations . . . 45
- 3.8 Implementation Setup . . . 46

4 Evaluation & Results 47
- 4.1 Evaluation Questions . . . 47
- 4.2 Evaluation setup . . . 48
  - 4.2.1 Benchmark applications . . . 48

4.2.2 Real-world application . . . 52
4.2.3 Technical setup . . . 52
4.3 Experimental results . . . 53
4.3.1 Results on effectiveness . . . 53
4.3.2 Results on efficiency . . . 56
4.3.3 Results on Applicability . . . 57
4.4 Discussion on evaluation . . . 59

5 Related Work **61**
5.1 Actor system analysis . . . 61
5.2 Automated software refactoring and refactoring of concurrent and distributed systems . . . 63

6 Conclusions and Future Work **67**
6.1 Contributions and Discussion . . . 67
6.2 Future work . . . 68
6.2.1 Research improvements . . . 68
6.2.2 Follow up research . . . 68
6.3 Conclusions . . . 69

Bibliography **71**

A Helper Algorithms **77**

List of Figures

2.1 Generated MFG from the example source code presented on Listing 2.1 8

3.1 Execution flow demonstrated as separate stages with dash-line representing optional step that exports Communication Flow Graph (CFG) as a dot-graph file. 16

Chapter 1

Introduction

In modern software development, particularly in the domain of distributed systems, achieving concurrency, scalability, and fault tolerance is of utmost importance. The actor model, first introduced by Carl Hewitt in 1973, offers a robust approach to handling these challenges [21]. Actors are independent computational entities that communicate exclusively through asynchronous message-passing, without a shared state, enabling them to avoid many concurrency issues such as race conditions and deadlocks. This makes them ideal for building scalable and resilient systems [44]. One such library for implementing the actor model on the JVM is Akka, which has gained significant traction for its ability to simplify the development of concurrent and distributed applications [48].

In modern software development, particularly in distributed systems, achieving concurrency, scalability, and fault tolerance is both difficult and important. Traditionally, concurrency was handled using the thread model, where multiple threads share the same memory space and would require specific constructs to ensure data isolation. However, this approach often led to complex issues like race conditions, deadlocks, and the need for intricate locking mechanisms to ensure data consistency [25]. The actor model emerged as an alternative to address these challenges more simply and effectively [21]. Unlike the thread-based model, actors are self-contained, independent computational entities that communicate exclusively through asynchronous message passing. By avoiding shared state, the actor model eliminates the need for locks, significantly reducing the risks of concurrency bugs like race conditions and deadlocks. This makes actors a powerful tool for building scalable and resilient systems in which components operate independently and failures are isolated to individual actors, enhancing fault tolerance [5]. One such library implementing the actor model on the JVM is Akka, which has gained significant traction for simplifying the development of concurrent and distributed actor-system applications by offering high-level abstractions that ease the complexities of managing concurrency [48].

Initially, Akka was implemented based on an untyped actor system, which does not enforce any restrictions on the types of messages exchanged between actors. While flexible, this approach can lead to runtime errors when messages of the wrong type are sent, as type checking only occurs at runtime [20]. This increases the difficulty of code maintenance, as developers must rely on manual management of message contracts and interactions. As a result, in large systems, untyped actors can introduce complications in maintaining and reasoning about the flow of messages across the system. To overcome these issues, Akka Typed was introduced to provide strong type safety into the actor model, enforcing type checks at compile time. This change significantly reduces the risk of runtime errors and improves code maintainability by clearly defining which messages an actor can handle, along with other type safety benefits. Typed actors specifically provide better guarantees about the correctness of message exchanges and make the system more understandable for developers, supporting long-term maintenance [31]. However, despite these benefits, there has been a noticeable adoption lag in migrating from Akka Classic (untyped) to Akka Typed [19].

The primary reason for this adoption lag is the complexity involved in refactoring an untyped actor system into a typed one. Refactoring and restructuring existing code without changing its external behavior has inherent challenges, but refactoring a distributed system poses additional difficulties due

to the system's complexity, non-deterministic behavior, and reliance on asynchronous message passing [16, 42, 51]. Ensuring that communication patterns and behavior remain intact while transitioning from untyped to typed actors is a demanding and error-prone task, which becomes even harder when a system spans multiple components or services across a network. In addition, there is a significant gap in tooling support for such transition as there are few tools or automated refactoring solutions to help developers transform untyped actor systems into typed [8]. Without automated support, developers must heavily invest productive time to ensure that all communication patterns are correctly transformed [36].

Such lack of tooling support can be explained by the currently limited ability to extract and interpret underlying actor system communication patterns, message protocols, individual actor interactions, and possible states [26]. Without this underlying capability, it becomes very difficult for an automated solution to comprehend and refactor actors while maintaining the correctness of the overall actor system. As a result, this thesis seeks to address the challenges of untyped actor system refactoring, by both providing a way to extract underlying communication patterns within the system, as well as implementing an automated refactoring solution for systems running on Scala. In addition, these extracted patterns could provide valuable insights to help with the migration from untyped to typed actor systems and lay the foundation for automated tasks like documentation generation, quality assurance, and test case generation [26]. By focusing on the refactoring of actor systems, and the unique issues they present, this work contributes to the broader goal of making typed actor systems more accessible to Scala developers.

1.1 Problem Description

The core issue tackled in this thesis is the lack of tooling and capability to effectively identify and extract communication patterns within actor systems, particularly those built in Scala using the Akka framework. This lack of capability to understand the underlying communication between actors leads to the broader problem of insufficient tooling support for refactoring efforts when transitioning from untyped actor systems to typed ones. Developers are often discouraged from refactoring due to the associated complexity and high effort cost resulting in adoption lag. Presently, no automated tool can analyze and interpret the communication protocols, actor interactions, and message exchanges necessary to perform accurate refactoring. This research aims to bridge that gap by proposing a method to extract the communication protocols, which can then be used to automate the refactoring of untyped actors in the Scala ecosystem.

The problem consists of several complex issues, each contributing to the overall challenge of extracting and refactoring actor communication patterns. First, identifying and resolving actor references is essential to determine inter-actor communications. This involves identifying each variable and constructor parameters and determining their values to know an actor's initial state. By knowing the initial state, we can start mapping out the behavior of each actor which includes determining what messages an actor can process, where the processing takes place, and how the actor's state or behavior may change over time. This also contributes to comprehending each message exchange between actors since it is also necessary to understand the data types and parameters carried by messages. Lastly, any and all interactions between actors or the external environment must be identified and mapped out between the variables representing actors and targets of each interaction.

To successfully automate the refactoring of an actor system, additional challenges must be addressed. Other than the identification of code elements that require modification, the resulting transformation of expressions and underlying data types must be determined. On top of that, some cases cannot be directly refactored with equivalent code expression due to underlying differences between untyped and typed actor systems, which may require design space exploration [51]. A robust yet flexible enough solution is essential to address wide range of scenarios, and only by addressing all these aspects can the refactoring process be successful and fully automated.

1.1.1 Research Question(s)

The primary goal of this thesis is to propose a way to extract communication patterns and implement an automated refactoring tool for the Akka library assisting the Scala ecosystem with the transition from untyped actors to typed ones. To achieve this, the following research questions will be answered:

- **RQ1**: *How to refactor an actor-based system to a typed actor-based system?*

  The first research question focuses on the foundational problem of transforming an untyped actor system into a typed actor system. The difficulty of this task comes from the requirement of preserving the communication patterns and message protocols that actors use to interact with each other in the system.

  - **RQ1.1**: *How can a message flow be extracted from an actor system?*

    The next key challenge is understanding and extracting the message flow between actors. Accurate extraction of message flow allows for the correct mapping of untyped messages to their typed equivalents, ensuring system correctness after refactoring.

- **RQ2**: *How effective is the implemented automated actor refactoring?*

  This research question aims to evaluate the efficiency of the refactored program. The goal here is to determine whether automated tools can reliably and accurately refactor actor systems without introducing bugs or degrading performance. This leads to three sub-questions to evaluate efficiency, maintainability, and applicability of the refactored code:

  - **RQ2.1**: *What effect does refactoring have on the performance of the refactored actor-based systems?*

    The first sub-question addresses the impact of refactoring on actor system performance. It investigates whether the refactored actor-based system performance is comparable to original after transitioning to a typed actor system. The evaluation focuses on ensuring that the system remains efficient without introducing significant delays or resource consumption that could hinder its operation.

  - **RQ2.2**: *How does refactoring influence the maintainability of actor-based systems?*

    The second sub-question explores how refactoring affects the system's maintainability. This involves assessing whether the refactored system remains easy to understand, modify, and extend. If refactoring leads to higher complexity or decreases maintainability, it may negate the benefits of automated refactoring due to requiring large manual effort to maintain the code which could have been used to manually refactor the system in the first place.

  - **RQ2.3**: *Are the implemented refactoring strategies applicable to real-world applications?*

    The last sub-question addresses the practical applicability of the proposed refactoring strategy. It seeks to validate whether the approaches developed in this thesis can be successfully applied to real-world actor-based applications and whether the technique generalizes beyond small use cases.

Chapter 2

Overview

This chapter provides an overview of the transformative process of refactoring from classical actor systems into typed actor systems. We begin by demonstrating a code example before and after refactoring and highlighting key modifications made to the actor system which includes Actor object definitions, Actor behavior and apply functions, and typed Message objects. The subsequent sub-chapter provides and explains the essential information for accurate refactoring by the software. Lastly, a concept of a Communication Flow Graph is presented that not only aids in understanding the intended actor communication protocol [26] but also ensures the software can accurately and effectively perform refactoring.

2.1 Overview of Refactoring

In order to provide a general understanding of the proposed refactoring approach, this section will present a simple example of Classic Akka code before and after refactoring into Typed Akka. Specifically, what is changed, why these changes are made, and the assumptions behind these changes.

The code example in Listing 2.1 depicts a simple actor system where the actor *Counter* can increment or decrement internal value. By receiving *Increment*, *Decrement*, or *Print* messages from the external environment, the *Counter* actor will increase the value, decrease the value, or print the current value respectively. The *Counter* actor also inherits the trait *traitWithActor* that itself extends *Actor* trait which allows for the class to define and override receive method. *Counter* class also requires a constructor parameter that indicates starting value. Finally, the actor system is created which then creates the *Counter* actor and processes incoming messages.

New constructs after refactoring. Because of the changes introduced in the Typed Akka framework [51], a few additional constructs are generated during the refactoring process. Primarily, the *Messages* object that may or may not contain message definitions along with associated trait, and the *userGuardian* actor [54]. Both of these constructs will be explained in detail in later chapters but in essence, (1) *Messages* object contains message types that are used by several actors and (2) the user guardian is defined as a setup *Behaviour* that is spawned as a top-level actor by the actor system in order to enforce best practices of actor hierarchy [48]. It is possible, however unlikely, that the *Messages* object may not exist because individual actors may send a certain message only to itself, meaning that the actor itself only needs to be aware of the message type, and as such that specific message type will be defined within the actor object definition. In all the other cases, at least two separate actor object definitions will need to be aware of message types because both the sending and receiving actors must know the message type and its definition. Then the *userGuardian* will, in general, contain every code entry and definition that creates or sends messages to an actor within the actor system. Given that, as a result, the refactored code in Listing 2.2 has an additional object *CounterUserGuardianMessages* that contains a trait and case objects that act as messages of type *Message*. Then, the *CounterUserguardianMessages* object is imported to the inherited trait *traitWithActor*, *Counter* actor object and a function *startActorSystem()* that defines user guardian actor behavior.

Actor object. In classical Akka, an actor class must extend or inherit an *Actor* trait in order to function as an actor and a function that returns *Receive* entity, but in Typed Akka, all it needs is a function that returns *Behaviour* type entity [50]. Because in Scala both object and class can contain a function, in Typed Akka it is best practice to define actors as objects and implement *apply()* function that would return a *Behaviour* entity [53]. As such, the actor classes are refactored into objects that inherit the same traits, but because the objects cannot consume constructor parameters, the *apply()* function that returns a *Behaviour* entity accepts the constructor parameters in order to process the parameter in the same way as a classical actor would.

Actor behavior and apply function. In a similar manner, any function that is named *receive(...)* of a type *Receive*, is assumed to provide the initial *Receive* entity definition, and as such it is refactored in Typed Akka as the starting *apply(...)* function with identical function parameters. Because both actor class constructor parameters and initial receive function parameters are moved to the *apply(...)* function as parameters, a second overloaded *apply(...)* function will be created if any of the constructor parameters are used within the function's definition. The first one will have an identical definition to the initial *receive(...)* function with some predefined default values being used to replace the constructor parameters, while the second *apply(...)* function will include initial *receive(...)* function parameters as well as class constructor parameters while keeping identical definition. This allows us to preserve three things; (1) inherited and overridden functions are preserved without the need to understand the deeper purpose of the overridden function, (2) ensure that the refactored typed actor *Behaviour* is aware of the parameters that are used within its definition, and (3) simplify the actor creation refactoring by preserving the actor parameter structure. This refactoring approach of the overloaded *apply(...)* function can be seen in the Listing 2.2 lines 20 and 21.

Message type object. Classical Akka also allows for actor class inheritance so that the inherited actor methods and values could be reused by other actors, but this presents two problems. The first problem occurs when you want to maintain the actor class hierarchy in Typed Akka because if we strictly refactor actor classes into objects then inheritance breaks due to Scala disallowing object inheritance, i.e. a class or an object inheriting another object. For that purpose, if an actor class is detected to be inherited by another class, then during refactoring both actor object and actor class will be created having identical internal structure to each other. In essence, this creates a companion object to the actor class that supports both standard Typed Akka actor creation approaches [53] as well as allows for actor objects to inherit the actor class definitions, functions, and values.

The second problem emerges when trying to determine the message type during *Behaviour* construction. Because the actor objects can inherit classes and traits that may or may not contain functions that return *Behaviour* entities, it is possible for them to be overridden. This means that the message type must be known in advance for the whole inheritance chain, and as a result, every function that returns a *Behaviour* entity and is overridden down the inheritance chain, must have the same message type. Such type inheritance can be seen in refactored Typed Akka code in the Listing 2.2 line 11 and line 20 where the *apply()* function is overridden. The overloaded function *apply(i: Int)* at line 21 is also defined with the same *Behaviour* type because otherwise it would cause a type mismatch with *countReceive(i: Int)* function, and the latter must match the type with overridden *apply()* function.

2.2 Information extraction

In order to refactor an actor system, it is necessary to correctly extract, comprehend, and eventually transform the intended communication protocol for the typed actor system. For that, the first step of the process is to correctly read and interpret the source code of a classic actor system by constructing an actor class inheritance graph. As mentioned in the last chapter, the classical akka actor system [50] allows for actor trait inheritance, and as such we must consider classes and traits that inherit it. Only by having an accurate inheritance graph it is possible to determine what *receive* functions each actor class can call since that determines different messages that an actor can process. That is important when refactoring *receive* functions due to typed *behavior*s and cases when a function is overridden in typed

```scala
object BasicAgentClassicExample extends App {
  object Counter {
    case object Increment
    case object Decrement
    case object Print
  }

  trait traitWithActor extends Actor {
    def receive: Receive = {
      case x => println("[Trait] I got a message: " + x.
          toString)
    }}

  class Counter() extends traitWithActor {
    import Counter._
    override def receive: Receive = countReceive(0)
    def countReceive(i: Int): Receive = {
      case Increment => context.become(countReceive(i + 1)
          )
      case Decrement => context.become(countReceive(i - 1)
          )
      case Print =>
        println(s"[Counter] Current count: $i")
        sender() ! Print
    }}

  import Counter._
  val system = ActorSystem("CounterSystem")
  val counter = system.actorOf(Props( new Counter(0)), "
      Counter")
  counter ! Increment
   .  .  .

  Thread.sleep(1000)
  system.terminate()
}
```

Listing 2.1: Classical Akka code example in Scala

```scala
object BasicAgentTypedExample extends App {
  object CounterUserGuardianMessages {
    trait Message
    case object Increment extends Message
    case object Decrement extends Message
    case object Print(replyTo: ActorRef[UserGuardian])
        extends Message
  }

  trait traitWithActor {
    import CounterUserGuardianMessages._
    def apply(): Behavior[Message] = Behaviors.receive { (
        context, message) =>
      message match {
        case x => println(s"[Trait] I got a message: " + x
            .toString)
          Behaviors.same
      }}}

  object Counter extends traitWithActor {
    import CounterUserGuardianMessages._

    override def apply(): Behavior[Message] = countReceive
        (0)
    def countReceive(i: Int): Behavior[Message] =
        Behaviors.receive { (context, message) =>
      message match {
        case Increment =>
          countReceive(i + 1)
        case Decrement =>
          countReceive(i - 1)
        case Print(replyTo) =>
          println(s"[Counter] Current count: $i")
          replyTo ! Print(context.self)
          Behaviors.same
      }}}

  def startActorSystem(): Unit = {
    import CounterUserGuardianMessages._
    val userGuardian: Behavior[Unit] = Behaviors.setup {
        context =>
      val counter = context.spawn(Counter(), "system-
          Counter1")
      counter ! Increment
       .  .  .
      Behaviors.empty
    }

    val system = ActorSystem(userGuardian, "Counting")
    Thread.sleep(1000)
    system.terminate()
  }
  startActorSystem()
}
```

Listing 2.2: Typed Akka code example in Scala

actors. In addition to constructing an inheritance graph, the names of every class and trait that extends or inherits actor trait are recorded as that will be needed when refactoring, referencing, and representing actors in the communication flow graph.

2.2.1 Communication Flow Graph

Once the inheritance graph is constructed, the next step is to construct a communication flow graph [26]. In essence, the Communication Flow Graph (CFG) represents an intended actor system communication pattern between actors, child actors, and potential message exchanges between them. As described in section 2.3 previous works have described and defined a similar representation, and largely our CFG will be very similar to [26] with few key differences. Since it is important for us to differentiate between different *receive/behavior* functions that process messages as well as what message triggers which response, the Communication Flow Graph will also include additional information for each incoming and outgoing edge. The information, represented as labels on an edge, will indicate the state of an actor, or more specifically the *receive/behavior* function, that must be active in order to process an incoming or

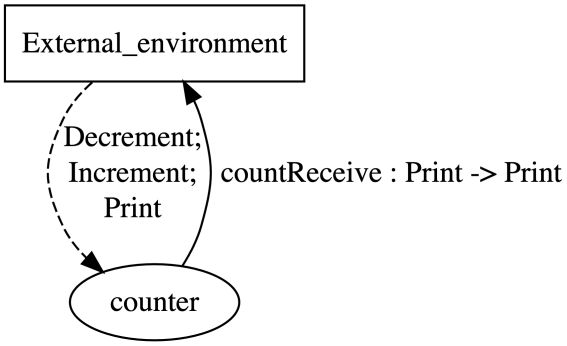

Figure 2.1: Generated MFG from the example source code presented on Listing 2.1

evoke an outgoing message. Additionally, an outgoing edge will also contain information on a message that was received by an actor and caused the same actor to respond with a message(s) to some target actor. Such representation requires to recognize, extract and record each message type that a given actor can process and, each message type that is sent to some actor. As a result, it is imperative to have a complete view of every state an actor can transition to because only by having a complete view it is possible to see what messages an actor can process, what messages the actor is capable of sending, and what, if any, child actors are spawned. Only then CFG can contain an actor representation that would depict every interaction that an actor makes within the actor system. Such depiction of an actor will be referred as an "**abstract actor**" as it will only represent total interactions that an actor class can make without considering the possibility of multiple actor instances of the same class existing in different states, and as such interacting differently.

To accurately collect the total states an actor can possess, each function that returns *Actor.Receive* type is analyzed. That is because any function that returns *Actor.Receive* type can be assigned as an actor state, and as result, it must be considered even if the state is rarely accessed. Then, it is necessary to collect the messages and their types that each receive function can process as it will be used when resolving behavior types and message objects during the refactoring process. In addition, it will be required when constructing MFG as it would provide a complete view of the possible actor interactions.

Finally, it is important to recognize the intended recipients of each message that an actor is sending in order to correctly represent an abstract actor's communication protocol. Generally, there are four ways of providing an actor with a target actor reference [53, 49]: (1) by passing an actor reference object as a constructor parameter, (2) by an actor using *context.sender()* function to reference the sender actor, (3) an actor itself finds or produces an actor reference by searching for an actor reference in the actor hierarchy tree or by spawning a child actor, and (4) by passing an actor reference object within a message. However, even considering only these four methods it may still present an exploration space too large to consider every possible expression as there are no hard restrictions imposed by the Classical Akka framework nor a developer is restricted to substitute standard expressions and functions with his own [52]. As such, in order to limit exploration space, we will limit ourselves by only considering the first two methods along with a few standardized expressions belonging to the third. The fourth method is omitted from our consideration both because it introduces far too much variability in possible expressions due to the nature of untyped actors and messages, and because it is considered bad practice to message anything other than immutable objects since that can introduce race conditions [31]. The detailed self-imposed restrictions and algorithms by which the target actors are resolved will be explained in section 3.

An example CFG can be seen in Figure 3.1 that was generated based on extracted information from the source code example in Listing 2.1. In it, you can see the messages that the external environment is sending indicated by the dashed line, and an abstract actor responding with *Print* message to the sender. The outgoing label from the actor *counter* contains additional information: the *receive/behavior* function that produces the outgoing message, the message that was processed and evoked the outgoing message, and the outgoing message.

Chapter 3

Implementation

This chapter delves into the implementation details of our refactoring tool designed for actor-based systems, outlining the underlying technical framework and the methodologies employed. The chapter begins by defining the foundational concepts and technical assumptions that guide the approach, setting a clear framework for the subsequent analysis. The core of our implementation revolves around static analysis techniques, including syntax tree analysis, flow graph construction, and inheritance graph construction, which are essential for accurate actor extraction and interaction mapping. Then we will describe the data structures and object models that support these processes and discuss the specific algorithms developed to facilitate the refactoring of the analyzed source code. Furthermore, the chapter will cover our setup for implementing these functionalities, highlighting integrations of existing libraries and tools. Despite exploring various existing solutions, limitations were encountered that required the development of customized tools tailored to meet the unique demands of actor-based systems. This detailed exposition aims to provide a clear and thorough understanding of the technical details underpinning the refactoring implementation.

3.1 Definitions

To provide clarity and establish a consistent understanding throughout this document, this section outlines the key definitions and terminology employed in describing our refactoring implementation.

One of the first steps in the process is the construction of the class dependency graph. Below are the terms and definitions used to describe the class dependency graph.

- **Class Dependency Graph** - Class dependency graph captures classes that either directly inherit an '*Actor*' trait or inherit a class inherited an '*Actor*' trait. The trait '*Actor*' is implemented by the Akka Classic library and it is required to define, create, and handle an actor in the actor system [52]. Any class or trait that does not inherit an '*Actor*' trait is omitted, even if a child class does inherit an '*Actor*' trait from another source. It is assumed that the class dependency graph is a directed acyclic graph that can also be a disconnected graph. The graph contains a set of nodes and edges that represent the inheritance of traits and classes defined in the extracted syntax tree.

- **Class Dependency Graph Node** - A graph node represents a single trait or a class. Additionally, each node contains two lists of parent and child node names, respectively. Each list contains nodes that are directly adjacent to a given node, meaning that there is only a single edge distance between the given node, parent nodes, and child nodes. Lastly, each node stores the syntax subtree that represents the body, name, and a few boolean variables that indicate whether it is a class or a trait, if a node extends the "Actor" trait, and if a class contains an '*abstract*' keyword in its definition.

- **Class Dependency Graph Edge** - A graph edge is a directed edge 'UV' from node 'U' to node a 'V' where 'U' comes before 'V'. The edges themselves do not exist as separate objects, but rather they are derived from the parent and child lists that each node contains.

- **Class Dependency Graph Start Node** - a node that directly inherits the '*Actor*' trait defined by the Akka Classic library. Note that the start node can contain any number of parent nodes but none of the parent nodes can inherit an '*Actor*' trait.

- **Class Dependency Graph Terminal Node** - a node with no outgoing directional edges, i.e., an empty child list.

The other large component is a Call Graph that guides the static analysis and refactoring. Below are the associated terms and definitions.

- **Call Graph** - A call graph represents calling relationships between different functions within the extracted Class Dependency Graph. Each node in the call graph corresponds to a function, and edges between nodes indicate that one function calls another. The graph is a cyclic-directed graph that can also be a disconnected graph.

- **Call Graph Node** - A node in a call graph represents a function that exists in the analyzed actor classes, and only in actor classes. So if an actor class calls a function that is implemented in, for example, a singleton object that is not a companion object, then such functions will be overlooked and ignored impacting the accuracy of the call graph. In addition, Each node also contains the name of the function, the fully qualified name of the node, which includes the namespace, any enclosing functions, traits, objects, or classes [17]. The node also indicates if the function, or any of its parent functions, returns a Akka Classic '`Receive`' type, and a list of calls made from the given function. The calls are represented as a list of custom objects that represent a function definition and a set of parameters that inform about the function's context.

- **Call Graph Edge** - Just as the Class Dependency Graph Edge, the Call Graph Edge is also a directional edge that is represented by the list of calls in any given node. By knowing the fully qualified name of a function in each node, we can know both the names and the namespaces, which can then be used to look up the next called function in the list of call graph nodes.

Another set of definitions relates to a points-to lists that identifies every instance of Akka Classic '`Props`' objects, actor instances, and actor reference objects '`ActorRef`'.

- **'`PropsInst`' points-to list** - This points-to list is produced by a Context-Insensitive, Flow-Insensitive source code analysis that captures every Akka Classic '`Props`' instance by recording the pointing variables to such instance, used parameters, and actor class that was used to instantiate the '`Props`' object.

- **'`ActorInst`' points-to list** - It is produced by a Context-sensitive, Flow-Insensitive analysis by identifying keywords that instantiate an actor object. Each entry records variable names that point to the actor object, the actor class that is identified through the '`Props`' instance (that we resolve through '`PropsInst`' points-to list), the parameters that are used during instantiation, and indication if this actor object was instantiated within a class or a trait that directly inherits Akka Classic trait '*Actor*'.

- **'`ActorRefs`' points-to list** - as opposed to the previous points-to analysis, this one is performed in a limited scope in order to resolve actor reference objects '`ActorRef`' in an actor class constructor parameters. As before, this analysis is a Context-Insensitive, Flow-Insensitive points-to analysis as it only maps an actor reference object to a potential list of target actor classes it could represent.

The last group of definitions is used to represent a Communication Flow Graph that represents an abstracted view of the actor system's communication message flows.

- **Communication Flow Graph** - The graph itself is a cyclic-directed graph that can also be a disconnected graph, where each node is an actor that can be created and each edge is a message flow path that can be sent or received by a given actor node. The graph does not guarantee that each actor node has been instantiated at any given time during runt-time, it only depicts every actor instance type that can be created and the interactions between them.

- **Communication Flow Graph Node** - each node represents an abstracted view of an actor, meaning that there is only a single node per actor class, even when there can be multiple actor instances of the given class with different communication patterns [53]. That is to say that each node is an abstracted view of all the observed actor class configurations and communication patterns through out all of its instances. Additionally, each node has a label that is equal to the given actor class name.

- **Communication Flow Graph Edge** - an edge is a directed edge 'UV' from node 'U' to node a 'V' where 'U' comes before 'V'. Since the graph can contain cycles, the source and the target of an edge can be the same node. Each edge has a label that indicates the message type that is sent, representing an interaction between actors caused by sending and receiving a message.

3.1.1 Data structures

This subsection delves into the data structures utilized within our source code refactoring framework corresponding to the definitions outlined previously. These structures serve to represent various objects, capturing essential information about their attributes, representation of source code elements, and their relationships within the software system.

The '`MetaClass`' data object represents a node in a Class Dependency Graph. The MetaClass encapsulates essential information about a given class that was recognized in the source code and identifies inheritance hierarchies, and their relationships within a software system. The attributes of the '`MetaClass`' object include:

- `name` - The name of the class/trait.
- `parentC` - A list of class/trait names that this class inherits from.
- `childC` - A list of class/trait names that inherit from this class.
- `isActor` - Indicates whether this class/trait inherits the 'Actor' trait class.
- `isAbstract` - Indicates whether this class/trait contains the 'abstract' keyword.
- `isClass` - Indicates whether this object represents a class (false indicates a trait).
- `tree` - Contains the syntax tree that implements the entire class.

```
1 case class MetaClass(
2           name: String,
3           parentC: List[String],
4           childC: List[String],
5           isActor: Boolean,
6           isAbstract: Boolean,
7           isClass: Boolean,
8           tree: Tree)
```

Listing 3.1: MetaClass data object definition

The '`FunctionNode`' data object represents a node in the function call graph, capturing essential information about individual functions and their calls within a code. The '`FunctionNode`' object contains the following parameters:

- `name` - The name of the function.
- `namePath` - The fully qualified name of the function, encompassing the namespace and any enclosing functions, traits, objects, or classes. This comprehensive identifier ensures unique identification

of the function within the entire software system. Because of the nature of the identifier, it is used to identify and handle inheritance and override possibilities.

- `isReceive` - A boolean indicator that specifies whether the function returns a Receive type or if any of its parent functions return a Receive type. This attribute is particularly relevant when identifying what functions may process the received messages.
- `calls` - A list of functions that this function calls. Each function in the list is of type '`FunctionTemp`', representing another data object that contains a number of parameters describing the target functions and their properties.

```
case class FunctionNode(
          name: String,
          namePath: String,
          isReceive: Boolean,
          calls: List[FunctionTemp])
```

Listing 3.2: FunctionNode data object definition

The '`FunctionTemp`' is a separate data object that gets initially constructed to represent all the functions observed in the source code, as well as collect a number of parameters describing the function and its fully qualified namespace. Because the fully qualified namespace of a function can contain various scopes in which a function is accessed, it was necessary to include classes, traits, and objects for the recognition logic to correctly determine the namespace. As a result, each '`FunctionTemp`' object can represent a trait, class, object, or a function with varying degree of parameters. The '`FunctionTemp`' contains the following parameters;

- `name` - The name of the function.
- `namePath` - The fully qualified name of the function, encompassing the namespace and any enclosing functions, traits, objects, or classes. This comprehensive identifier ensures unique identification of the function within the entire software system.
- `objectType` - An integer value indicating the type of node. This value helps to determine if the node is a function definition or a potential function call within a class, trait, or object. The possible values are: 0, 1, 2, and 3, representing function definition node, object node, class node, and trait node, respectivelly.
- `access` - An integer value indicating the access level of the function. The possible values are: 0, 1, 2, 3, representing public, protected, override, and private keywords, respectivelly.
- `params` - A list of lists of parameters of the given node, if it has any.
- `imports` - A list of import statements used in the body of the current node.
- `calls` - A list of function names that this node makes a call to.
- `dclType` - The return type of the current function. If the node is not a function, this value is equal to None.
- `body` - The body of the current node.

```
case class FunctionTemp(
          name: String,
          namePath: String,
          objectType: Int,
          access: Int,
          params: List[List[Term.Param]],
          imports: List[Import],
          calls: List[String],
          dclType: Option[Type] = None,
          body: Term = Term.Name("EMPTY"))
```

Listing 3.3: FunctionTemp data object definition

Both '`PropsInst`' and '`ActorInst`' have almost identical data structure except that the'`ActorInst`' also indicates whether an actor instance is created within an actor class. As a result, only the '`ActorInst`'

data object's parameters will be explained:

- `varNames` - A list of variables representing the actor instance. This attribute captures different variable names used to reference the same actor instance.
- `aClass` - The actor class associated with this actor instance. Identifies the type of actor to be instantiated.
- `params` - A list of parameters or variables received by the actor instance during instantiation. This may include variables that reference an actor props class.
- `inAClass` - A boolean flag that identifies whether this actor instance is created inside an actor class that directly extends the actor trait. This attribute distinguishes between actor instances created within actor classes and those created elsewhere in the program.

```
case class ActorInst(
          varNames: List[String],
          aClass: String,
          params: List[String],
          inAClass: Boolean)
```

Listing 3.4: ActorInst data object definition

Like previous data structures that represent graphs, the Communication Flow Graph is also expressed as list of a single data objects '`IntermediateActor`'. However, each CFG node contains additional custom data objects to represent the discovered actor's possible states, interactions, and outgoing message calls. Each of these data objects will be explained separately, starting with a '`IntermediateActor`' that represents an abstract actor in the Communication Flow Graph:

- `name` - The name of the intermediate actor, which is the same as the actor class name.
- `actorRefParam` - A list of constructor parameters for the actor class that is of type ActorRef. This provides information about the actor references used within the actor's constructor.
- `states` - A list of behavioral states that the actor can possess and transition to. Each of the states contains details about that particular state that an actor can be in.

```
case class IntermediateActor(
   name: String,
   actorRefParam: List[ActorRefs],
   states: List[State])
```

Listing 3.5: IntermediateActor data object definition

The '`State`' data object represents a receive function within an actor, containing the name of the function, a flag indicating if it is overridden, and a list of interactions that occur within each state. Each state can be viewed as a possible behavioral state that an actor could take. So a list of '`State`' within an '`IntermediateActor`' object is a list of behaviors that an actor could transition to and from. The '`State`' data object parameters are as follows:

- `recM` - The name of the state, which is the same as the receive function name. This attribute identifies the specific state of the actor.
- `isOverride` - A boolean flag indicating whether this state overrides a previous definition of the same state, such as a state inherited from a superclass or trait.
- `interactions` - A list of interactions that occur within this state. Each interaction details the communication and behavior associated with this state.

```
case class State(
   recM: String,
   isOverride: Boolean,
   interactions: List[Interaction])
```

Listing 3.6: State data object definition

Each state's object '`Interaction`' models a case statement and the code block it triggers within an actor's receive function. Crucially, it holds a list of outgoing messages with the message recipients, along with the triggering message for this interaction, and an indication of whether the triggering message's

type should be included in the refactoring later on. This is because some 'case' statement expressions may not indicate the message type directly. The '`Interaction`' data object is characterized by the following attributes:

- `caseName` - The name of the variable or message that triggered this case statement. This is the part that appears directly after the 'case' keyword in the code.
- `msType` - The message type, if a type was specified. This provides information on the type of message handled in this case statement.
- `inMessageObj` - An optional boolean indicating whether this interaction's **caseName** should be considered as a message that might be included in a new and refactored message definition object. This distinction is important because some case statement expressions may contain a message with an implicit type, while others might capture any input and treat it as an unknown type variable.
- `messageCall` - A list of message calls originating from this actor as a result of triggering this interaction. This details the subsequent communications initiated by the actor upon receiving this type of message within its current state.

```
case class Interaction(
    caseName: String,
    msType: Type,
    inMessageObj: Option[Boolean],
    messageCall: List[MessageCall])
```

Listing 3.7: Interaction data object definition

Within an '`Interaction`' object, the '`MessageCall`' data object represents a single message sent to some target actor, detailing the target, message type, and message contents. Each '`MessageCall`' instance contains the following parameters:

- `target` - The name of the target receiving the message. Since the target may not directly point to an actor reference, it is treated as an unknown variable that eventually points to some actor.
- `message` - The message type being sent. This attribute captures the specific message being communicated.
- `args` - The contents within the message that is being sent.

```
case class MessageCall(
    target: String,
    message: String = "",
    args: List[Term])
```

Listing 3.8: MessageCall data object definition

Lastly, the data object '`MessageType`' is used to discover and represent the actual message definitions within the whole actor system and later verify the observed messages during refactoring. The data object '`MessageType`' records these parameters:

- `name` - The name of the message type. This identifies the specific message being defined.
- `params` - A list of parameters specified in the message definition. These parameters define the data type and contents of the message.
- `msTrait` - An optional trait that the message definition inherits, if any. This indicates any traits or interfaces the message type may implement.
- `template` - This captures the complete message definition body and structure. It becomes important when refactoring message definitions.
- `isCLass` - A boolean flag indicating whether the message definition is expressed as a class or an object.

```
case class MessageType(
    name: String,
    params: List[List[Term.Param]],
    msTrait: Option[Type],
```

```
5     template: Template,
6     isCLass: Boolean)
```

Listing 3.9: MessageCall data object definition

Note: the data types of 'Tree', 'Type', 'Term', and 'Template' are defined by a library that we use to produce and modify the syntax tree of the actor system application source code that is under analysis.

3.2 Technical assumptions

This section outlines the initial technical assumptions that the analysis and refactoring of actor-based systems is based on. These assumptions define the actor expressions and the specific rules that must be satisfied by the source code of the actor system application. That includes the expected structure of actor definitions, message-passing expressions, and their definitions, handling of actor and message variables, and interaction patterns between the actor system and external environment (i.e. remaining application code that interacts with the actor system). These assumptions ensure the accuracy and effectiveness of our static analysis and subsequent refactoring processes.

For the most part, these assumptions are based on the official Akka Library best practices and style guide guidelines that aim at preserving the advantages that an actor system brings while preventing common pitfalls [61, 52]. However, these assumptions also serve to limit possible code expressions and structures by setting boundaries and reducing the complexity associated with diverse coding styles. As a result, this simplifies the process of static analysis and refactoring, making it possible to automate these tasks effectively. Below you can find the list of assumptions:

- **Single Actor System Analysis** - The analysis and refactoring are conducted on a single actor system. By extension, the application on which analysis and refactoring are conducted will host only a single actor system.
- **Actor Responsibility** - Adheres to the principle of single responsibility for actors, meaning an actor should handle only one type of task and receive and process a specific set of messages. In addition, a single message should represent a single intent of interaction.
- **Actor Definition** - Actor definitions must extend an `Actor` trait or inherit it from another class or trait. As a result, a class, trait, or object that does not extend or inherit an 'Actor' trait should not have functions that return an actor behavior ('Receive' type) and hence will not process messages or otherwise influence the actor's state. This ensures that actors communicate exclusively through asynchronous message passing, with no direct sharing of state between actors. Moreover, classes, traits, or objects implementing actors must be defined in the specified source code files and inherited functions should not be ambiguous, i.e. inherited function should exist only in a single parent.
- **Stateless Actors** - Actors should not be stateful, i.e. contain or save stateful variables other than constructor parameters. Instead, the changes to the state should be passed as function call parameters, aligning more to a functional programming style.
- **Type Inference** - Sufficient type information can be inferred or provided for messages and actor behaviors to transition them to a typed system. As a result, no actors should exclusively process messages of type 'Any'.
- **Message-handling** - Each actor must contain message-handling logic that processes incoming messages by using pattern matching to differentiate between messages. Such logic can be changed, indicating a change in the actor's behavior.
- **Actor Behavior Switching** - The actors should support different lifecycle phases that can be represented by switching between different behaviors ('Receive' functions), like a finite state machine, aligning more with a functional programming style.

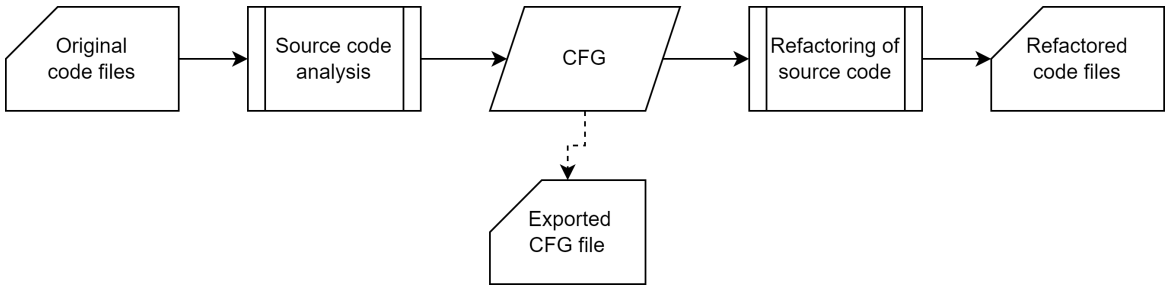

Figure 3.1: Execution flow demonstrated as separate stages with dash-line representing optional step that exports Communication Flow Graph (CFG) as a dot-graph file.

- **Message Parameters** - Message parameters must be immutable, thereby maintaining encapsulation of an actor's state and preventing access by other actors. Consequently, messages will not contain another message as a constructor parameter as that would introduce a risk of variable outcome, and violate the **Actor Responsibility** assumption.

3.3 Execution Overview

The execution flow is designed to systematically transform untyped actors within an actor system by conducting a static analysis followed by a refactoring process. Such two-phase approach ensures that the actor's interactions and message handling are observed, examined, and then optimized or modified as necessary. The overall objective is to accurately represent actor behaviors and message interactions within the system while transforming untyped into a typed actor system.

The execution begins with the **Static Analysis** 3.3 phase where the system parses and analyzes the actors and their associated message types. This involves identifying all messages used by each actor and determining how these messages are processed in various states. The analysis phase results in the generation of several data structures: a class dependency graph, communication flow graph, call graph, points-to lists, and message type list. These structures provide a core understanding of the communication patterns and dependencies within the system, serving as a required input for the refactoring.

Following the static analysis, the implementation proceeds to the **Refactoring** 3.3 phase which utilizes the extracted data from the static analysis to transform the actor and their message-handling function definitions. This involves modifying actor class and trait, updating constructor parameters, and creating new message objects where necessary. The refactoring process ensures that all actor references and message interactions are considered and aligned with the original system design. By incorporating the static analysis results, the refactoring phase can accurately target areas that require modification and apply changes precisely and efficiently.

The static analysis and refactoring phases are inherently interconnected. The static analysis provides a comprehensive understanding of the system's current state, which is essential for informed decision-making during refactoring for targeted modifications that preserve the system's performance and maintainability. Together, these phases form an execution flow that ensures the system is both thoroughly analyzed and effectively transformed to benefit from a typed actor system. The overview of this process is visualized in the figure 3.1.

Static Analysis Overview

The execution of static analysis begins with the formation of points-to lists 3.4.1. Initially, a points-to list for **Props** instances is created by parsing the source code syntax tree. This list identifies which actors are instantiated by each **Props** object. Subsequently, an actor instance points-to list is formed by using the previously created **Props** points-to list. This second list maps actor instances to the specific **Props** objects that create them, providing a clear mapping between actors, **Props** instance, and their variables.

Next to the creation of points-to lists, a class dependency graph is constructed, also using the source code syntax tree 3.4.2. This graph represents the dependencies between different classes and traits in the system which is then topologically sorted to facilitate the construction of a call graph, and subsequently a communication flow graph.

The call graph 3.4.3 maps out the relationships between various function calls within the system, indicating which functions invoke others, and as a result, it provides important details into the flow of execution within the actor classes and helps identify potential points of interaction and dependency when processing received messages .

Following the creation of the call graph, a communication flow graph is generated 3.4.4. This graph is built using the previously created data structures: the topologically sorted class dependency list, the actor instance points-to list, and the call graph. The communication flow graph is represented by a list of **Intermediate Actors**, each encapsulating the behavior and possible states of individual actor class. The process of creating the communication flow graph involves a chain of sub-functions that construct various data structures as part of the **Intermediate Actor** creation. These sub-functions extract states representing receive functions, interactions representing case statements within the receive functions, and call statements made within the case statements.

The final step is the identification of message types used by the actor system 3.4.5. This is achieved by analyzing the source code syntax tree in conjunction with the communication flow graph represented by a list of **Intermediate Actors**. This step ensures that all message types utilized within the system are identified.

With the set of these data structures laying the groundwork for the subsequent refactoring phase, the refactoring phase can begin by calling a custom syntax tree transformer that acts as an entry point.

Refactoring Overview

When the transformer is called, the execution flow begins by creating a map from the list of **Intermediate Actors** that will represent a set of messages each intermediate actor can understand and process 3.5.1. The map is then used to identify the largest possible supersets in order to determine potential actor types and consolidate the message-type objects based on the underlying type. Such a process is particularly useful when inheritance is heavily used across the actor system producing numerous but similar actor classes.

Once the largest possible supersets are identified, the message type objects are generated based on these underlying types described by each superset, the list of original message types, and the source code syntax tree message type definitions 3.5.2. Subsequently, each actor class is refactored using **Intermediate Actors** representation, the respective superset, and associated actor type, the original syntax tree of the actor class, the call graph, and a set of parameters based on observed actor definitions 3.5.3. This refactoring process is similarly composed of a set of functions that evaluate and process specific parts of the actor class's definition that eventually produce two lists of refactored syntax trees: one for the refactored actor classes and another for the new message type definitions containing the identified actor types. These lists are then returned by the transformer and the main function proceeds to reconstruct the syntax tree to restore the source code into proper format.

The refactoring phase ensures that each discovered actor class is taken into account and transformed to match the expected syntax structure for a typed actor system without falling back to untyped actor definitions. While every actor class is transformed, as we will see some of the possible actor system compositions could result in more complex structures than it was beforehand degrading constancy and maintainability 4.

3.4 Static Analysis

This section looks into the static analysis process, the first step in the source code refactoring as demonstrated here 3.1, detailing the execution flow, algorithms, and outcomes. The algorithms employed will

be looked at through pseudocode and explained with the rationale behind them highlighting how they help reconstruct the underlying actor system communication protocol. Such analysis sets the foundation for refactoring the untyped actor system into a typed one.

The first step in the process is the construction of the supporting data structures explained previously, specifically in this order: points-to lists for 'Props' and 'ActorRef' objects, class dependency graph with topological sorting, collection of all functions and construction of call graph, discovery and construction of defined message types, and construction of Communication Flow Graph with the help of supporting data structures.

Algorithm 1: Identify Props Instances

```
   Input: tree: Source
   Output: List of PropsInst
1  Function identifyProps(tree):
2      props ← empty list of PropsInst;
3      foreach value definition in tree do
4          if definition creates "Props" && assigns to a variable then
5              names ← list of variables that this value is assigned to;
6              if "Props" contain parameters then
7                  foreach p in parameters do
                       /* Traverse syntax subtree p and look for certain
                          expressions                                      */
8                      if p is term "classOf" && p contains any type parameter t then
9                          params ← all other parameters that are not p
10                         aClass ← lowercase string of the type parameter t
11                         props ← props ∪ {PropsInst(names, aClass, params)};
12                     else if p is term "new" then
                           /* We know the class type parameter t from the term "new"
                              */
13                         aClass ← lowercase string of type parameter t
14                         if class type parameter t contains other parameters then
15                             foreach param in t do
16                                 params ← params ∪ {param} ;
17                         else
18                             params ← empty list of strings
19                         props ← props ∪ {PropsInst(names, aClass, params)};
               /* Assumes that "Props" does not contain constructor parameters
                  and expects only a type parameter                        */
20             else if "Props" contains one and only one type parameter t then
21                 aClass ← lowercase string of type parameter t
22                 props ← props ∪ {PropsInst(names, aClass, empty list of strings)}
23     return props;
```

3.4.1 Points-to lists

Both 'Props' and 'ActorRef' points-to lists are constructed in a similar fashion, however, the recognition logic for these object types is different. In both cases, the entirety of the source code syntax tree is traversed in order to identify syntax patterns that indicate the initialization of a 'Props' and 'ActorRef' objects, but the patterns that identify the objects are different. For that purpose, we will examine only

the algorithm used to produce the 'Props' points-to list while noting a few key differences between an algorithm that produces 'ActorRef' instance points-to list.

The Algorithm1 is a relatively simple one as it only traverses the syntax tree while looking for expressions that define a 'Props' object. Once an expression is recognized that creates a 'Props' object and is assigned to some variable, then a list of variables is collected that refer to this object. Afterward, the 'Props' object itself is examined to differentiate between several different expressions that can be used to pass an actor class to the 'Props' constructor. If 'Props' contains parameters, then it must be a class runtime representation of an actor class, which we assume is represented either with the '*new*' keyword or '*classOf*' function. If there are no constructor parameters but there is one and only one type parameter, then it is assumed that the specified type parameter is an actor class. This holds true because 'Props' expects a class with inherited *Actor* trait [56]. Once an actor class is identified, the remaining parameters are collected and a 'PropsInst' instance is created and added to the 'PropsInst' list. Lastly, after traversing the complete syntax tree, the 'PropsInst' list is returned.

There are only two key differences compared to the process when producing the **ActorInst** list. Firstly, instead of matching definitions that create a **Props** instance, a specific call to a function 'actorOf' is looked for. This function call signifies an actor instantiation which, by definition [52], means that the parameter must be a **Props** instance. As a result, the found parameter is evaluated, whether it is a direct instantiation of a **Props** object, or whether it is some variable. If it indeed is a variable, then it is matched with the variables identified in the **PropsInst** list, and if found, the associated actor class is known.

Clearly, such implementation can only handle one degree of indirectness without awareness of the surrounding context. Even if two separate variables representing a different instance of **Props** object existing in different class scopes share the same name, then they will both be incorrectly recognized as a reference to an actor class. Having this in mind, the points-to lists are used in a limited scope to resolve actor class constructor parameters of type **ActorRef** [57], as explained later.

3.4.2 Class Dependency Graph

A Class Dependency Graph (CDG) will be essential for refactoring untyped actor classes, as it provides a structured representation of the relationships and dependencies between various classes within an actor system. By mapping out these dependencies, the CDG allows us to understand how different classes interact and rely on each other, and what traits, functions, and definitions are accessible for a given actor class. That is crucial for identifying potential points of failure or areas that need modification during the refactoring process. Additionally, having a topologically sorted CDG can help with the communication flow graph's construction as that would simplify the process by presenting the dependencies in a linear order. As a result, the source syntax tree is analyzed and CDG is constructed once points-to lists are processed.

The algorithm begins by initializing two empty lists and starts iterating through each node in the syntax tree, checking if the node is a trait or class definition. In both cases, the function 'scan' 3 is called that parses and extracts necessary information for the class dependency graph from a trait or a class node, but the difference is the value of a 'isClass' flag. The results of these scans, if not empty, are added to a temporary meta class (*tmc*) list 'tmcList'. Once all trait and class definitions are processed, the algorithm refines each *tmc* to establish child-parent relationships to create a Meta Class list 'mcList'. It is necessary because initially we only know the traits and classes that a given definition inherits when observing it for the first time, not what classes or traits extend it. In an ideal world, if every trait and class is defined in sequential order of their inheritance then we could identify the child classes the first time a class or a trait is observed, however, the real world is messy so we can not make such an assumption.

Algorithm 2: Construct Class Dependency Graph

```
   Input: tree: Source
   Output: List of MetaClass
 1 Function constructClassDependencies(tree):
 2     tmcList ← empty list of TempMetaClass
 3     mcList ← empty list of MetaClass
 4     foreach node in syntax tree do
 5         if node is a Trait definition then
 6             scan(false, node);
 7             If result was not empty, add it to tmcList;
 8         if node is a Class definition then
 9             scan(true, node);
10             If result was not empty, add it to tmcList;
11     foreach tmc in tmcList do
12         childC ← empty list of String
13         foreach otherTmc in tmcList do
14             if otherTmc != tmc && otherTmc is child of tmc then
15                 Add otherTmc to childC;
           /* Initially, every inherited class or trait is treated as a parent.
              They must be filtered.                                          */
16         newParentC ← Filter tmc.parentC unobserved classes;
17         mcList ← mcList ∪ {MetaClass(tmc.name, newParentC, childC, tmc.isActor,
            tmc.isAbstract, tmc.isClass, tmc.tree)};
18     startNodes ← Filter mcList for start nodes;
       /* Ensures the class dependency graph adheres to Actor Definition
          assumption.                                                         */
19     mcListFiltered ← filterActorDependencies(startNodes, mcList - startNodes);
20     return mcListFiltered;
```

Having that in mind, each *tmc* checks other temporary meta classes to determine if they inherit the current *tmc*, which is added to 'childC' list if true. Afterward, the parent class of the *tmc* is filtered to remove unobserved parent classes and then constructs a **MetaClass** object, adding it to 'mcList'. The filtering of the parent's list is important because initially every inherited class and trait is treated as a parent, but not every inherited class or trait may implement an actor and its relevant functions. Since we established the assumption **Actor Definition** that only traits and classes that inherit the actor trait can define the actor and its states, we should filter out the parent list only to contain the relevant classes or traits.

Once all meta classes are processed, the algorithm identifies the starting nodes in order to filter out all and any unrelated traits and classes initially identified. This is the consequence of observing and capturing every class and trait definition, even if a given definition does not inherit an actor trait. Even though such an approach is not ideal, it is necessary since initially we do not know what traits and classes may inherit an actor trait. For that purpose, the function 'filterActorDependencies' 4 will remove any unrelated classes and traits from the CDG. As a result, the last step of the algorithm filters the 'mcList' to produce only the start nodes and calls the 'filterActorDependencies' function.

Next, we will look into the 'scan' function algorithm that extracts all the relevant information from a given class or a trait and then returns a temporary meta class object option. The algorithm takes two inputs: a Boolean 'isClass' indicating whether the node is a class, and the 'node' itself, representing a class or trait definition with its body. It begins by initializing several variables with their default values

and then checks if the node extends any classes or traits. If true, then two checks are conducted: whether the given node contains an abstract keyword, and whether inherited classes or traits contain the *Actor* trait. There is an additional sanity check on the syntax structure since due to limitations, not all possible syntax expressions can be recognized. Once the list of inherited classes and traits is processed, the option of **TempMetaClass** is created with the name of the class/trait, parent list, three boolean flags indicating the presence of actor trait, presence of abstract keyword, and a class flag, and the node itself. Since classes or traits can only inherit the actor trait, the '`scan`' function will return **None** for any class or a trait that does not inherit anything.

Algorithm 3: Scan class/trait for Class Dependency Graph

```
Input: isClass: Boolean, node: Tree
Output: Option of TempMetaClass
1  Function scan(isClass, node):
2      isActor ← false
3      isAbstract ← false
4      mc ← None
5      parentC ← empty list of String
6      if node extends then
7          if node definition contains abstract keyword then
8              isAbstract ← true;
9          foreach parent in inherited classes/traits do
10             if parent meets expected syntax structure then
11                 Add parent to parentC;
12                 if parent is "Actor" trait then
13                     isActor ← true;
14         mc ← Option(TempMetaClass(node.name, parentC, isActor, isAbstract, isClass, node));
15     return mc;
```

The value **None** in Scala represents an absence of value, the opposite of it being a value of option of some type. As a result, the algorithm returns an option of **TempMetaClass** representing a value's possible presence or absence in a type-safe way [63].

Algorithm 4: Filter Class Dependency Graph

```
Input: startNodes: List[MetaClass], remainingMcList: List[MetaClass]
Output: List of MetaClass
1  Function filterActorDependencies(startNodes, remainingMcList):
2      newStartNodes ← filtered list of remainingMcList that are children of startNodes
3      if newStartNodes is not empty then
4          return startNodes + filterActorDependencies(newStartNodes, remainingMcList -
             newStartNodes);
5      else
6          return startNodes;
```

Lastly, the algorithm for the function '`filterActorDependencies`' filters out the Meta Class list to remove any classes or traits that do not inherit or extend the **Actor** trait to meet the **Actor Definition** assumption. The function takes two inputs: a list of **MetaClass** objects representing the starting nodes of the graph, and the list of remaining **MetaClass** objects to be processed. The algorithm begins by filtering the '`remainingMcList`' to contain only those objects that are children of the given start nodes list '`startNodes`'. If the filtered list '`newStartNodes`' is not empty, the algorithm recursively calls itself, passing in the '`newStartNodes`' as the start nodes, and the filtered list '`remainingMcList`' without the '`newStartNodes`' as the new list of remaining **MetaClass** objects to be processed. This process

continues until the new start node list is empty. When no more new start nodes are found, the algorithm returns the accumulated 'startNodes', representing the filtered class dependency graph.

In essence, this function serves two purposes: first to ensure the class dependency graph contains only the classes that are relevant to the actors, consequently ensuring the **Actor Definition** assumption. Secondly, it resolves a few edge-case scenarios where some of the classes used to remain in the list even when validating if a class inherits **Actor** trait.

Class Dependency Graph Topological sorting

Because of the possibility of an actor inheriting receive functions or functions that process messages outside of the receive function while overriding others, we must consider every previous class in the CDG when processing all the states an actor can be in. As a result, we must process every class an actor class inherits sequentially to have a complete representation of an actor class. Having that in mind, and because the CDG is an acyclic-directed graph, topological sorting becomes a convenient way of processing the class in the correct order when constructing intermediate actor data objects.

The topological sorting is implemented via Depth-First search algorithm described here [14]. In principle, it is a recursive algorithm that visits each node of the graph in an arbitrary order and preprends a given node to the output list when all of nodes it depends on have been already visited and prepended to the output list.

3.4.3 Call Graph

The construction of the call graph utilizes a similar two-stage pattern used in constructing CDG where initially all functions are found and recorded, and then each of them is processed to build the call graph. For that, the function 'collectAllFunctionTemp' 5 visits every node of CDG and scans every function defined while constructing fully qualified namespaces for each trait, class, object, and function definition. As a result, in addition to functions, the initial list of 'FunctionTemp' objects will contain nodes that are traits, classes, and objects that contain calls made to any other function.

It starts with an empty list of intermediate temporary functions and an empty map to track inheritance names, where the map keys are class names, and the values are lists of sequences representing trait, class, object, or function names in inheritance paths.

The algorithm iterates over each dependency class in the input list which at that point is topologically sorted. For each class without a parent (a root node), it adds the list of traits, classes, objects, or functions found in a given trait/class to the list 'functionsTemp'. Then, it initializes the namespace for the root class in the inheritance names map 'inheritanceNames' with an empty list. The extraction of **FunctionTemp** from a given trait/class is achieved by 'produceNames' and 'findFunctionsTemp' helper functions that format the namespace string identifier and scan the node, respectively. The actual extraction is done by the function 'constructFunctionTemp' that is called by the 'findFunctionsTemp'. In short, the 'constructFunctionTemp' traverses the syntax tree looking for a specific pattern, and once found, identifies certain parameters, constructs and then returns the **FunctionTemp** instance.

For traits/classes with parents, the algorithm first checks if the parent is a root node in the inheritance namespace map. If it is, it adds the found FunctionTemp in the trait/class to the list and updates the inheritance path by adding the current class name to the list of sequences associated with the parent class. The process of identification and extraction of the **FunctionTemp** objects are identical to the process when a node is a root node except with different parameters passed down when calling the 'produceNames' and 'findFunctionsTemp' functions.

For remaining traits/classes, it iterates over the entries in the inheritance names map to find a matching parent. If a matching parent is found and the current node name is not already in the sequence, it further processes the node. It filters the sequences in the inheritance names map to find those that contain the parent class name and end with it. If such sequences are found, the **FunctionTemp** from the node is constructed and added to the 'functionsTemp' list. The inheritance path is updated by appending the node name to the appropriate sequence's end. If no sequences are found that end with the parent name,

it creates a new sequence by splitting the existing sequence at the parent class name, adding the current node name at the end of that sequence, and updating the inheritance name map with a new namespace path and the list of 'functionsTemp' accordingly. Like previously, the identification and extraction of **FunctionTemp** are done by calling 'produceNames' and 'findFunctionsTemp' functions. Finally, the algorithm returns the list of **FunctionTemp**.

In essence, the inheritance names map 'inheritanceNames' acts as a reference map that is used to keep track of and resolve the namespace of each trait, class, object, or function that is discovered during analysis, whereas 'functionsTemp' is a list of discovered nodes that may contain calls to some function with already identified namespace. Since each **FunctionTemp** instance already knows its own namespace, there is no need to return the inheritance names map. However, the algorithm relies on a few assumptions that will be covered in a later section.

Once all possible function calls and functions are discovered within the class dependency graph, the 'constructCallGraph' will process the 'functionsTemp' 6 list to produce a call graph.

Algorithm 5: Collect All Function Temp

```
   Input: topoSClassDepL: List[MetaClass]
   Output: List of FunctionTemp
1  Function collectAllFunctionTemp(topoSClassDepL):
2      functionsTemp ← empty list of FunctionTemp
3      inheritanceNames ← empty map of Strings mapping to List of Sequences of type String
4      foreach class in topoSClassDepL do
           /* Considers root nodes                                              */
5          if class.parentC is empty then
6              functionsTemp ← functionsTemp ∪ {findFunctionsTemp(class,
                produceNames("", empty Seq of type String))};
               /* Initialize root nodes in the namespace reference             */
7              inheritanceNames[class.name] ← empty List of Seq of type String;
8          else
9              foreach parent in class.parentC do
                   /* Considers nodes directly after root node                 */
10                 if inheritanceNames contains key parent.name then
11                     functionsTemp ← functionsTemp ∪ {findFunctionsTemp(class,
                        produceNames("." + parent.name, empty Seq of type String))};
12                     inheritanceNames[parent.name] ← inheritanceNames[parent.name] ∪
                        Seq(class.name);
                   /* Considers all remaining nodes                            */
13                 else
14                     foreach (root, names) in inheritanceNames do
15                         if names contains parent.name and not class.name then
                               /* Selects all Seq that contain the parent      */
16                             targetBranch ← inheritanceNames[root].filter(seq → seq contains
                                parent.name);
17                             parentBranch ← targetBranch.filter(seq → seq last element equals
                                parent.name);
18                             if parentBranch is not empty then
                                   /* Scan class and update namespaces         */
19                                 foreach seq in parentBranch do
20                                     functionsTemp ← functionsTemp ∪
                                        findFunctionsTemp(class, produceNames("." +
                                        root.name, seq));
21                                     index ← inheritanceNames[root].indexOf(seq);
22                                     inheritanceNames[root][index] ← seq :+ class.name;
23                             else
                                   /* Add a new Seq to the list root           */
24                                 indexSeq ← targetBranch.head.indexOf(parent.name) + 1;
25                                 (firstHalf, _) ← targetBranch.head.splitAt(indexSeq);
26                                 newParentBranch ← firstHalf :+ class.name;
27                                 functionsTemp ← functionsTemp ←
                                    findFunctionsTemp(class, produceNames("." + root.name,
                                    firstHalf));
28                                 inheritanceNames[root] ← inheritanceNames[root] +
                                    newParentBranch;
29     return functionsTemp;
```

Algorithm 6: Construct Call Graph

```
   Input: functionDepL: List[FunctionTemp]
   Output: List of FunctionNode
 1 Function constructCallGraph(functionDepL) :
 2     funcList ← filter functionDepL for function nodes & sort it by namespace position in
         descending order;
 3     funcNodesTemp, funcReceive ← empty sequence of FunctionNode;
 4     funcNonPrivate ← filter functionDepL for non-private nodes;
 5     foreach function in funcList do
 6         funcMatch ← foreach call in function.calls do
 7             callName ← function call name;
 8             allDuplicates ← filter funcNonPrivate for nodes that are named callName;
 9             finalCalls ← empty list of FunctionTemp;
10             if allDuplicates.size > 1 then
                   /* Selects the function definition in the highest position in
                      the namespace                                              */
11                 highestHierarchy ← sort allDuplicates by namespace and take min;
12                 foreach dupFunction in allDuplicates do
13                     if dupFunction.access is 'override' && dupFunction is in the namespace of
                         highestHierarchy) then
14                         finalCalls ← finalCalls ∪ dupFunction;
15             else
16                 finalCalls ← allDuplicates;
17             finalCalls
18         if function returns Receive && function.name not exist in funcReceive & is in the
             same namespace then
19             funcReceive ← funcReceive ∪ FunctionNode(function.name, function.namePath,
                 true, funcMatch);
20         else if function exists in funcReceive and is not Receive then
21             do nothing & continue;
22         else
23             funcNodesTemp ← funcNodesTemp ∪ FunctionNode(function.name,
                 function.namePath, false, funcMatch);
24     funcInheritReceive ← foreach funcR in funcReceive do
25         propagateReceive (funcR, funcNodesTemp) .distinct
26     funcNodes ← funcReceive ∪ funcInheritReceive ∪ filter funcNodesTemp that are not in
         funcInheritReceive & in the same namespace;
27     return funcNodes.toList;
```

The "Construct Call Graph" algorithm6 builds a call graph from a list of **FunctionTemp** objects, generating a list of **FunctionNode** objects that represent a call graph. It begins by filtering and sorting the input list of **FunctionTemp** objects based on namespace position. Two temporary sequences are initialized to hold the function nodes (`funcNodesTemp`) and functions that return "Receive" (`funcReceive`). Lastly, a list of non-private nodes is produced by filtering out non-private nodes from the input list.

The algorithm then iterates over each function in the sorted list. For each '`function`', it processes the function calls to identify the potential target functions of that call by name. If there are multiple matches, it first selects the function definition with the highest position as a root definition of a function (line 11). Then each match is added to a returning list '`finalCalls`' if it contains **override** keyword

and is within the same namespace as the root function definition. If there is only one match, it adds it directly to the returning list. As a result, the resulting list 'funcMatch' should contain only a list of **FunctionTemp** objects that could be called by the current 'function' with all the possible function definitions due to inheritance.

Next, the algorithm checks if the 'function' returns "Receive" and is not already in the 'funcReceive' list while being in the same namespace. If so, it adds a new function node to the 'funcReceive' list. If the 'function' is already in the 'funcReceive' list and does not return "Receive", it skips further processing for that function. Otherwise, it adds the function node to the temporary function nodes list 'funcNodesTemp'.

After processing all functions, the algorithm propagates "Receive" through the call graph by calling the 'propagateReceive' function on each function in the 'funcReceive' list, ensuring each function node is included only once. It is done to correctly identify what functions are called by a function that eventually returns a receive type which becomes important when processing the actor definitions and their possible states.

Finally, it combines the 'funcReceive' list, the propagated "Receive" functions, and the remaining function nodes 'funcNodesTemp' that are not in the propagated list within the same namespace, returning the complete list of function nodes. This process constructs the call graph, accounting for function dependencies and namespace hierarchies.

3.4.4 Actor class analysis

After the call graph is constructed, all the necessary data structures are present for the actor class analysis to begin. It is a combination of syntax tree exploration searching for certain patterns, relevant information extraction from the syntax tree, and reference operations using the previously constructed data structures, i.e. call graph, class dependency graph, points-to lists, etc. The implementation itself is a series of functions handling certain processes when creating the **IntermediatActor** objects, with each function depending on the next one.

Algorithm 7: Process Topologically Sorted Classes

Input: topologicalSortedClassD: List[MetaClass], actorInst: List[ActorInst], callGraph: List[FunctionNode]
Output: List[IntermediateActor]

```
1 foundIActors ← empty list of IntermediateActor;
2 foreach classDep in topologicalSortedClassD do
3     returnedIActor ← analyzeAClass(classDep, actorInst, foundIActors, callGraph);
4     foundIActors ← foundIActors ∪ returnedIActor;
```

The algorithm 7 is part of the main function which only loops through the topologically sorted class dependency list and analyzes the given class by calling 'analyzeAClass' 8. The returned **IntermediatActor** instance is added to the 'foundIActors' list which is used later on when refactoring.

The "Analyze Actor Class" algorithm 8 analyzes a given actor class to produce an intermediate representation of the actor, called **IntermediateActor**. It begins by initializing an empty list to store the states inherited from parent classes. The algorithm then extracts the states and actor references from the current class by calling the extractStatesActor function. This function returns a tuple containing the resolved actor references and the extracted states.

If the current class has parent classes, the algorithm iterates over each parent class. For each parent class, it filters the list of currently resolved intermediate actors to find those that match the name of the parent class. The algorithm then iterates over each state of the parent intermediate actor, checking whether the state should be included based on several conditions: if the state is not already in the parent states list, if it has more interactions than a state with the same name in the list, or if it has 'override' keyword while a state in the list does not. If any of these conditions are met, the state is added to the parent states list. Note, the last two checks are redundant due to the **Actor Definition** assumption,

and the if statement is implemented as a Scala pattern matching partial function (ensures that the or statements are evaluated sequentially)

Algorithm 8: Analyze Actor Class

Input: classDep: MetaClass, actorInstList: List[ActorInst], intermediatActorList: List[IntermediateActor], callGraph: List[FunctionNode]
Output: IntermediateActor

```
1  Function analyzeAClass(classDep, actorInstList, intermediatActorList, callGraph):
2      parentIAStates ← empty list of State;
3      extractedStatesAndActorRefs ← extractStatesActor(classDep, actorInstList,
         callGraph);
4      resolvedActorRefs ← extractedStatesAndActorRefs[2];
5      extractedStates ← extractedStatesAndActorRefs[1];
6      if classDep have parent classes then
7          foreach parentClass in classDep.parentC do
8              parentIAList ← filter intermediatActorList for intermediate actors that are named
                 parentClass.name ;
9              foreach parentIA in parentIAList do
10                 parentAIState ← empty list of State;
11                 foreach parentState in parentIA.states do
12                     if parentIAStates not contains parentState.name or
13                     parentIAStates contains parentState.name but parentState has more
                         interactions or
14                     parentIAStates contains parentState.name and parentState has override
                         keyword while the state in parentIAStates with the same name does not
                         have override keyword then
15                         parentAIState ← parentAIState ∪ parentState;
16                 parentIAStates ← parentIAStates ∪ parentAIState;
17     finalStates ← extractedStates ∪ filter parentIAStates to not contain duplicates with
         extractedStates;
18     return IntermediateActor(classDep.name, resolvedActorRefs, finalStates);
```

After processing all parent classes, the algorithm combines the extracted states from the current class with the parent states, ensuring no duplicates. The final list of states is created by merging these two lists. Since the currently extracted states will always overwrite inherited states with the same name, the '`override`' keyword can be essentially ignored. Finally, the algorithm returns an **IntermediateActor** instance, which includes the class name, the resolved actor references used in the constructor, and the final list of states. This process constructs a comprehensive intermediate representation of the actor class, considering both its own states and those inherited from parent classes.

To understand how the states of each actor are extracted, we will look at the '`extractStatesActor`' 9 function algorithm. As mentioned, the algorithm is called for each class in a topologically sorted class dependency graph with a points-to set of all actor instances, and a call graph.

The algorithm implemented by a function 9 processes a given actor class to identify its states and resolve its actor references passed down through constructor parameters. The function begins by initializing two empty lists: '`currentlyExtractedState`' for the extracted states and '`currentlyResolvedActorRefs`' for the resolved actor references.

Then, each node in the class dependency tree is checked to determine whether the node is a class or a trait definition. If it is, the algorithm resolves the actor references by calling the '`resolveActorRefs`' function with the class name, the list of actor instances, and the class constructor.

Algorithm 9: Extract States and Resolved Actor References

Input: classDep: MetaClass, actorInstList: List[ActorInst], callGraph: List[FunctionNode]
Output: Tuple (List[State], List[ActorRefs])

```
1 Function extractStatesActor(classDep, actorInstList, callGraph):
2     currentlyExtractedState ← empty list of State;
3     currentlyResolvedActorRefs ← empty list of ActorRefs;
4     foreach node in classDep.tree do
5         if node is Trait or Class definition then
6             currentlyResolvedActorRefs ← resolveActorRefs(node.name, actorInstList,
                node.constructor);
7             currentlyExtractedState ← extractReceiveMethods(classDep.name, node.body,
                currentlyResolvedActorRefs, callGraph);
8     return (currentlyExtractedState, currentlyResolvedActorRefs);
```

The 'resolveActorRefs' function essentially just extracts necessary information from the syntax tree, iterates through every constructor parameter of type **ActorRef**, and calls another function 'identifyActorRef' that actually resolves and identifies the actor instance that the **ActorRef** points to. Now, the function 'identifyActorRef', taking an actor reference name and the points-to set of all actor instances, first identifies all instances that are instantiating the given actor class name and collects a list of every raw parameter variable name that is passed to the constructor. Then, each actor instance's list of variables is checked whether any of them exist in the previously collected list of raw variable names. If it does exist, then the actor class of that instance is added to a list of target classes which is eventually returned as a potential list of target actor classes.

Such implementation allows us to resolve actor references by one degree of indirectness, however it also introduces possible inaccuracies. Because the construction of points-to lists and resolution of actor references here do not take into account namespace scopes, identical variable names with different values can exist in different scopes while being mistakenly recognized as a possible actor reference value. On the other hand, because an intermediate actor is an abstracted actor representation of all observed actor class configurations and communication patterns, the variable namespace scope is not of concern since the points-to list of actor instances already ensures that all instances of a given actor class will be evaluated with its own constructor parameters.

Following actor reference resolution, the states are extracted from the class body by invoking the 'extractReceiveMethods' 10 function, which takes the class name, the class body, the currently resolved actor references, and the call graph as parameters. Eventually, the resolved actor references and extracted states are appended to respective lists.

To avoid double computation of actor reference resolution, the algorithm returns a tuple containing the list of currently resolved actor references and a list of extracted and inherited states.

The function 'extractReceiveMethods' 10 with the help of a call graph recognizes and extracts relevant information from functions that either return a **Receive** type or are called by a receive function.

The algorithm begins by iterating over each function definition in the class body. For each definition, it first checks if the definition is a function. Then, if the function's return type is **Receive**, the algorithm calls the 'exploreReceiveMethod' 11 function with the class name, definition modifiers, name, and its body as parameters, adding the result to the empty state list 'stL'.

Algorithm 10: Extract Receive Methods

Input: className: String, body: List[Stat], actorRefs: List[ActorRefs], callGraph: List[FunctionNode]
Output: List of State

```
1  Function extractReceiveMethods(className, body, actorRefs, callGraph):
2      stL ← empty list of State;
3      foreach definition in body do
4          if definition is a function definition then
5              if definition return type is Receive then
6                  stL ← stL ∪ exploreReceiveMethod(className, definition.mods,
                     definition.name, definition.body);
7              else
8                  functionNode ← filter callGraph by definition.name and className;
9                  if functionNode is not empty and is called by a receive function then
10                     stL ← stL ∪ exploreReceiveMethod(className, definition.mods,
                         definition.name, definition.body);
11     return stL;
```

If the definition does not return a **Receive** type, the algorithm filters the call graph to find nodes matching the definition's name and class name. If such a node is found and the definition is called by some function definition that returns a **Receive** type, the algorithm again calls '`exploreReceiveMethod`' with the appropriate parameters and adds the result to the state list. After processing all definitions in the class body, the algorithm returns the list of identified states.

Algorithm 11: Explore Receive Method

Input: className: String, mods: List[Mod], defName: Term.Name, rBody: Term
Output: State

```
1  Function exploreReceiveMethod(className, mods, defName, rBody):
2      inL ← empty list of Interaction;
3      isOverride ← Boolean flag identifying if definition mods contain 'override' keyword;
4      foreach Case statement in rBody do
5          cs ← Tuple of the name of a message that triggers this case statement, an indication
               whether the case name can be treated as a message type for when refactoring message
               definition, and observed type definition;
6          s ← extractCallStatements(case.body, className, actorRefs);
7          inL ← inL ∪ Interaction(cs.name, cs.type, cs.option, s);
8      return State(defName, isOverride, inL);
```

Next, the "Explore Receive Method" algorithm implemented by the function '`exploreReceiveMethod`' 11 processes a function definition to extract relevant interactions, i.e. '*case*' statements of a partial function, and constructs a **State** object. The input parameters of this algorithm are actor class name, list of modifiers of the given function definition, definition name, and definition body.

The algorithm iterates over each '*case*' statement within the body to identify the variable name of a message that triggers the case statement, an option of boolean value that indicates if the message could be treated as a variable type indicator of the received message, and observed type definition if any. This is extracted from the syntax subtree of the '*case*' statement as a tuple of three.

Next, the algorithm calls the '`extractCallStatements`' 12 function, passing the case statement body, class name, and resolved actor references as parameters. This function identifies and extracts all the call statements within the body, returning a list of **MessageCall** objects.

Using the identified message name and the extracted call statements, the algorithm constructs an **Interaction** object. This object encapsulates the details of the interaction, including the name of a

message that triggered the case statement, the type of the message, any option parameter, and the call statements. The interaction is then added to the list of interactions.

After processing all case statements in the definition body, the algorithm constructs a **State** object using the definition name, the override flag, and the list of identified interactions. The **State** object represents the actor's behavior in terms of its interactions with other actor classes within this function definition, reflecting how it handles messages and the actions it performs. Finally, the algorithm returns the constructed **State** object.

Algorithm 12: Extract Call Statements

```
   Input: classBody: Term, className: String, actorRefs: List[ActorRefs]
   Output: List of MessageCall
 1 Function extractCallStatements(classBody, className, actorRefs):
 2     s ← empty list of MessageCall;
 3     foreach term in classBody do
 4         if term expression is using scala syntax sugar for '!' or '?' to send a message then
 5             s ← identifyCall(s, term, actorRefs);
 6         else if term expression is call to some object function then
 7             if term is a call to "become" or "unbecome" function then
 8                 s ← s ∪ MessageCall("self", "self", params);
 9             else if term is a call to "tell" function then
10                 s ← identifyCall(s, term, actorRefs);
11             else if term is a call to "ask" function then
12                 s ← identifyCall(s, term, actorRefs);
13             else if term is a call to "forward" function then
14                 s ← identifyCall(s, term, actorRefs);
15     return s;
```

The goal is to gather all instances where messages are sent or received using specific actor communication patterns in Scala.

The algorithm 12 begins by iterating through each term in a given class body. For each term, it checks if the term expression uses Scala syntax sugar, specifically the '`!`' or '`?`' operators, which are commonly used for sending messages. If such an expression is found, the algorithm calls the '`identifyCall`' 13 function to process and add the message call to the list.

If the term expression is a call to an object function, the algorithm further examines if the function call is '`.become(...)`' or '`.unbecome(...)`' which are methods used in Akka actors to change the actor's behavior. In such cases, it directly adds a **MessageCall** object with '*self*' as the message type and receiver to the list, indicating that the actor is changing its behavior with '*params*' holding the new behavior receive function. While it is not exactly the intended use of the **MessageCall** object, later on, we will see that **MessageCall** objects with message type '*self*' are ignored for the most part.

In addition, the algorithm also checks for other specific function calls such as '`.tell(...)`,' '`.ask(...)`,' and '`.forward(...)`,' which are methods used in Akka for different types of message passing. For each of these function calls, it again invokes the '`identifyCall`' function to process the term and add the corresponding message call to the list.

After processing all terms in the class body, the function returns the list of identified **MessageCall** objects. These objects represent the various message interactions found within the actor class, capturing how the actor communicates with other actors or itself.

Algorithm 13: Identify Call Message

```
   Input: oldS: List[MessageCall], term: Term, actorRefs: List[ActorRefs]
   Output: List of MessageCall
 1 Function identifyCall(oldS, term, actorRefs):
 2     newS ← empty list of MessageCall;
 3     message ← extract message and message parameters from term.params;
 4     target ← extract message send target from term;
 5     if target is "sender" then
 6         newS ← oldS ∪ MessageCall(target, message.name, message.params);
 7     else if target is "self" then
 8         newS ← oldS ∪ MessageCall(className, message.name, message.params);
 9     else
           /* Resolved Actor references                                    */
10         foreach actorRef in actorRefs do
11             if target matches actorRef then
12                 foreach targetClass in actorRef.targetClasses do
13                     newS ← newS ∪ MessageCall(targetClass, message.name,
                         message.params);
14         newS ← newS ∪ oldS
15     return newS;
```

The function '`identifyCall`' implements the algorithm 13 that processes a given term representing a message call and identifies the target of the message that a given actor is sending. That is achieved by storing the message target, message, and message parameters within a **MessageCall** object and saving it in a list. The algorithm takes in an existing list of message calls, a term representing the message call, and a list of actor references.

The algorithm begins by extracting the intended target, which indicates the recipient, the message, and its parameters from the term. If the target is identified as Akka '`sender()`' identifier, the algorithm creates a **MessageCall** object with the target as '*sender*' and message details, and adds it to the new list of message calls. Similarly, if the target is '*self*,' indicating that the actor is sending a message to itself, a **MessageCall** object is created with a given actor's class name as the target and added to the list.

For other targets, the algorithm iterates through the list of resolved actor references to identify the actual target class. If the target matches an actor reference, it iterates through the target classes associated with that actor reference and creates a **MessageCall** object for each target class, adding them to the new list. Otherwise, the target is assumed to be an actor instance that was created within the actor itself, or identified by the Akka actor search queries [55]. Finally, the algorithm combines the new list of **MessageCall** objects with the old list and returns the combined list.

Having identified message calls in a given interaction, the whole chain of functions finally allows us to analyze every actor class by identifying each of their communication patterns, gaining insight into message types that are being exchanged between actors, and resolving actor references that are passed as constructor parameters. However, as it stands now, the communication pattern [53] where a message contains an actor reference or an actor is created within another actor is not taken into account and resolved when constructing a Communication Flow Graph, i.e. populating Intermediate Actor list. Such instances of communication are left out and assumed to be correctly resolved by the original source code since it must have contained the actor reference variables and used them to process messages. Only during refactoring, could it present an issue due to message and actor type conflicts but as we will see later this can be resolved.

3.4.5 Identifying Message Types

Understanding all message type definitions in the source code is important for understanding the communication patterns within an actor-based system. Messages define the interactions between actors, and having a clear mapping of these types allows for better analysis of the actor system's behavior and message flow between them. In addition, it aids in verifying and extending the observed intermediate actors and their interactions, especially during refactoring. As a result, the algorithm 14 analyses the source code, identifies the message type definitions, and creates a Tuple of lists: one for globally observed sent messages, and another for **MessageType** objects.

Algorithm 14: Extract Message Types

```
   Input: iaList: List[IntermediateActor], source: Source
   Output: Tuple of (List of String, List of MessageType)
1  Function extractMessageTypes(iaList, source):
2      Function identifyMessageTypes(node, messages, isClass):
3          (isParentClassObject, parentNode) ← isNodeParentAnObjectOrClass(node.parent);
4          if isParentClassObject && parentNode has no inheritance then
5              if node has single inheritance then
6                  mst ← MessageType(node.name, node.params, Some(node.inherited),
                       node.template, isClass);
7              else if node has no inheritance then
8                  mst ← MessageType(node.name, node.params, None, node.template, isClass);
9          return mst
10     acMessages ← every distinct message sent by an actor in iaList;
11     allMessages ← extractAllSeenMessages (source);
12     messages ← distinct(allMessages ∪ acMessages);
13     mstList ← empty list of MessageType;
14     foreach node in source do
15         if node is object && node.name is in messages then
16             mstList ← mstList ∪ identifyMessageTypes(node, messages, false)
17         else if node is class && node.name is in messages then
18             mstList ← mstList ∪ identifyMessageTypes(node, messages, true)
19     return (allMessages, mstList);
```

The algorithm 14 implemented by '`extractMessageTypes`' aims to identify and list all message types used within the source code. It uses the list of previously found intermediate actors '*iaList*', the entire syntax tree '*source*', and a helper function '`identifyMessageTypes`'. The helper function checks if a given node's parent in the syntax tree is an object or a class definition (implemented by '`isNodeParentAnObjectOrClass`' function), determines if the parent has no inheritance, and if true, checks if the node has at most one inheritance. Based on these checks, the helper function creates a **MessageType** object that encapsulates the node's name, parameters, inheritance details, body, and whether it's a class.

The helper function follows a common pattern defined in Akka style guide [61] where message types should be defined in a companion object or within a class that contains only messages or parameters relevant to a specific actor class definition, by extension this means it should not inherit anything. By already expecting a specific kind of expression for message type definitions, we can simplify the logic for message type identification.

The algorithm itself starts by collecting all distinct messages sent by all Intermediate Actors and combines them with messages extracted from the entire source code using another helper function '`extractAllSeenMessages`'. This function simply traverses a given syntax tree (in this case it is the whole source code syntax tree) looking for message-sending expressions, similarly to the algorithm 12,

except the 'extractAllSeenMessages' returns a list of strings containing the sent message.

Then the algorithm iterates over each node in the source code syntax tree looking for nodes that are either an object or a class definition, and whose name matches a message in the combined list. It uses the previously mentioned helper function 'identifyMessageTypes' to identify its message type details, extract them, and add to a list of message types '*mstList*'. Finally, a tuple is returned containing the list of all globally observed sent messages '*allMessages*' and the list of identified message types '*mstList*'.

Such a result provides a comprehensive view of the message types used in the system which helps with detecting messages and resolving their types during refactoring. However, the output also depends on the expected message type definitions limiting the design space of developers. While this allows us to avoid exploring a large design space looking for message types, and as a result simplifying algorithm (and by extension implementation), when adhering to an officially recommended style guide there are cases when such expected expressions are not desirable and may even become troublesome [61, 52]. Nevertheless, for now, we accept such potential inaccuracies for the simplified process as we will see during evaluation in most cases it is good enough.

Eventually, once all the intermediate actors are extracted, sent messages observed, and message types recognized, a special function is called that will start the refactoring process expecting a list of intermediate actors, topologically sorted class dependencies, call graph, and message type list with globally observed sent messages. In addition to refactoring, the communication flow graph represented by a list of intermediate agents will be transformed into a graph description language format (DOT) [**?**] and saved locally as a DOT format file to provide both a window to see how the tool interpreted a system under analysis, as well as to give anyone using it an abstracted view of an actor system.

3.5 Refactoring

The next stage of our application involves refactoring the original source code to transform untyped actor classes into typed actor definitions. This stage leverages the data structures created during the static analysis phase, namely the list of intermediate actors, the topologically sorted class dependencies, the call graph, and the message type list. This transformation aims to enhance the safety and reliability of the actor-based system by utilizing the Akka Typed framework [48]. Refactoring in this context involves several key transformations due to core differences between Akka Typed and Classic that enable the compiler to enforce type safety and provide a robust hierarchical framework for defining actors in addition to readability and maintainability of the code [20].

Overall, the process of refactoring begins through the function 'transform' 15 by analyzing all the message types with intermediate actors to determine what core types can be assigned and defined for one or more actors. Then based on that dedicated objects for each base message type are defined and populated for each message type initially received by the transformer. Lastly, each actor class is refactored based on a number of conditions, rules, and observed actor class components or definitions that eventually get returned once the transformation is complete. Each of these steps will be examined in detail within this section along with their accompanying algorithms, interactions between them, codebase assumptions, and reasoning behind them.

In general, the algorithm 15 acts as the main entry point for the refactoring to begin. The first several steps of the process only prepare the data via the functions 'messageTypesDictByIA' 16, 'findSupersets' 17, and 'refactorBody' 20, all of which will be explained later. Then going through the topologically sorted actor class list, each actor class is refactored by calling 'refactorBody' through the *transformerActorObject* transformer, and a message object is defined which includes all the messages within the corresponding superset to that actor class. The refactored actor class and the corresponding message definition object are added to a list that may or may not contain duplicate definitions expressed as classes based on whether a given actor class inherits another actor class. This is because Scala objects cannot be inherited [64], and as we have to consider inheritance, a duplicate actor class will be included so that it can be inherited.

Algorithm 15: Refactoring Main

Input: iaList: List of IntermediateActor, topoMcList: List of MetaClass, mstList: List of MessageType, callGraph: List of FunctionNode, allMessages: List of String
Output: Tuple of List of Stat, List of Tree

```
1  Function transform(iaList, topoMcList, mstList, callGraph, allMessages):
2      messageTypeDictByIA ← messageTypesDictByIA(iaList, mstList);
3      mergedSets ← findSupersets(messageTypeDictByIA, empty Map);
4      transformerActorObject ← new Transformer that calls refactorBody(...) with various
         parameters based on if a node is Class or Trait definition
5      refactoredMcList ← empty list of Tree;
6      refactoredMsObj ← empty list of Stat;
7      foreach actorClass in topoMcList do
8          correspondingActorSet ← filter mergedSets where key contains actorClass.name;
9          correspondingMessageObj ← createMessageObjects(correspondingActorSet,
             mstList);
10         transformedActor ← transformerActorObject(actorClass.tree);
11         if actorClass.childC is not empty then
12             refactoredMcList ← refactoredMcList ∪ transformedActor and transformedActor
                 but expressed as a class;
13             if correspondingMessageObj not in refactoredMsObj then
14                 refactoredMsObj ← refactoredMsObj ∪ correspondingMessageObj;
15         else
16             refactoredMcList ← refactoredMcList ∪ transformedActor;
17             if correspondingMessageObj not in refactoredMsObj then
18                 refactoredMsObj ← refactoredMsObj ∪ correspondingMessageObj;
19     return (refactoredMsObj, refactoredMcList)
```

3.5.1 Actor Type Resolution

Identifying actor types is the first step in the refactoring process since addressing dependencies and inheritance structures effectively is essential to prevent potential type conflicts between parent and child actor classes. This ensures that type safety and message handling are preserved which simplifies the refactoring process. However, the Typed Akka documentation [51] presents several approaches to typed actors and their messages which in some cases substantially differ from the common approaches used in untyped actor systems. That resulted in taking a few liberties during refactoring to introduce a few additional function and object definitions aimed at accommodating these incompatible approaches. Unfortunately, as we will see when discussing limitations, there are still a number of edge cases and communication patterns that are either overlooked, incorrectly refactored or transformation does not preserve the actor's intended communication pattern.

As the first step, it is important to have an overview of all messages that each actor is capable of processing. We do that by creating a map where the key is a list of actor class names and the value is a set of tuples for each message where the first element is a message that the actor processes and a boolean indication if that message was sent to a '`sender()`'. Such a map is created by the function '`messageTypesDictByIA`' 16 which takes the intermediate actor list and a list of identified message types.

Algorithm 16: Return a Map of IA to Set of Used Messages

Input: iaList: List[IntermediateActor], mstList: List[MessageType]
Output: Map of (List of Strings, Set of Tuples (String, Boolean))

```
1  Function messageTypesDictByIA(iaList, mstList):
2      dict ← empty Map;
3      allActorMs ← list of all messages sent across all actors in iaList;
4      foreach ia in iaList do
5          messages ← empty Set of (String, Boolean) tuples;
6          foreach st in ia.states do
7              foreach in in st.interactions do
8                  notExists ← messages does not contain in.caseName or in.msType;
9                  if notExists and in.inMessageObj is true or in.caseName exists in mstList or
                    in.msType exists in mstList then
10                     matchingMs ← filter allActorMs to contain only messages that match
                        in.caseName;
11                     messages ← messages ∪ (in.caseName, across all messages if any
                        matchingMs.target is equal to "sender");
12         dict ← dict ∪ (List[String](ia.name), messages);
13     return dict;
```

The function 'messageTypesDictByIA' 16 implements an algorithm for generating a map that associates each intermediate actor with a set of used messages. Basically, the algorithm iterates over each intermediate actor in the '*iaList*' and analyzes interactions to produce a tuple where the first element is a message and the second is a boolean value indicating if a message was sent to a 'sender()'. That is achieved by iterating through the interactions of every intermediate actor's states. First, it is confirmed if the message in the current interaction was not already identified as we only care about unique messages that an actor processes. Then, the message is verified to match previously identified message types or that it was already identified as a message type that an actor processes. In both cases, the recognized message in the interaction is then filtered through the list '*allActorMs*' containing all messages sent throughout all observed actors in order to determine if any matching messages are sent to 'sender()'. At this point, the '*messages*' set is updated to include the newly discovered interaction message and the boolean value. After processing all interactions for an actor, the '*dict*' map is updated to include a new key-value map pair with the intermediate actor name being the key and the '*messages*' set as a value.

The importance of knowing if a message was sent to Classic Akka 'sender()' function stems from the fact that Typed Akka does not implement and support such a communication practice by default [51]. As a result, we need to take into account whenever a message is sent to a "sender" because that will require an additional step during refactoring in order to maintain the same communication pattern between actors.

The resulting structure is intended to represent a mapping between an actor class and a list of messages that an actor can process. The idea behind such structuring stems from the **Actor Responsibility** assumption that each actor will process their own messages and in an ideal application no message will be processed by more than one actor. Unfortunately, the Classic Akka does not enforce such a structuring [52], and as a result, we must expect that more than one actor can process a given message. As such, the algorithm must first explore and find the base messages used by multiple actors, group actors based on messages to encompass the largest number of actors with the least number of messages, and isolate those groups containing a set of messages representing a list of actors. The function 'findSupersets' 17 does exactly that by looking for supersets among different message combinations between different actors and merging actor message sets with its possible largest superset. However, cases, where an actor's message set partially belongs to some supersets without a true superset existing, remain unresolved, which results in separate message groups that could potentially contain duplicate messages across mul-

tiple groups. As a consequence of duplicate messages, if a subset belongs to several supersets, it is merged into the biggest fitting superset.

During the generation of message type definitions, as explained in the later section 19, each superset is used to generate its own respective type with each message definition extending that specific message type. This results in a type that encompasses a set of messages that one or more actors can process and interpret within a dedicated Scala object [64] holding the message definitions and message companion objects.

Algorithm 17: Find Supersets

Input: remaining: Map[List[String], Set[(String, Boolean)]], visited: Map[List[String], Set[(String, Boolean)]]
Output: Map of (List of Strings, Set of Tuples (String, Boolean))

```
1  Function findSupersets(remaining, visited):
2      if remaining.isEmpty then
3          return visited
4      else
5          finalSets ← empty map;
6          head ← remaining.head;
7          tail ← remaining.tail;
8          if visited.contains(head.key) then
9              return findSupersets(tail, visited)
10         else
11             subsets ← filter tail to return all subsets of head.value;
12             supersets ← filter tail to return all supersets of head.value;
13             newRemaining ← filter tail by removing all elements found in subsets;
14             if supersets.nonEmpty then
15                 biggestSuperset ← findSupersets(supersets, empty Map).head;
16                 subsetsHead ← subsets ∪ head;
17                 finalSets ← compareAndUpdateSets(biggestSuperset, subsetsHead, visited)
18             else
19                 finalSets ← compareAndUpdateSets(head, subsets, visited)
20             return finalSets
```

The algorithm implemented by the function '`findSupersets`' 17 tries to identify and process supersets within a collection of sets, where each set is represented as a mapping from a list of strings to a set of tuples containing a string and a boolean value. Here the key value of the mapping represents a list of actor classes and the values represent the messages that were used by those actor classes, with each message also indicating (via a boolean flag) whether it was sent to a standard Classic Akka '`sender()`' function.

The core of the algorithm involves recursively evaluating each actor class and message set within a given Map pair, and determining its relationship with other actors and their message sets in terms of message subsets and supersets. In addition to finding supersets, the algorithm also propagates the boolean values of each message within their respective supersets. Because the '`sender()`' function is no longer present in the Typed Akka, it is necessary to identify which messages are used with '`sender()`' so that during refactoring it would be taken into account. However, it is possible to observe a message that is not using '`sender()`' in some actor class only to discover later that another actor class is using the same message with '`sender()`' function. As such, the boolean indicator of a message within a respective superset must be updated whenever it is discovered that a message is sent to '`sender()`'.

The function begins by checking if the input '*remaining*' is empty which keeps track of unevaluated actor class and message set pairs. So, if it is, the function returns the '*visited*' map, which keeps track

of the sets that have already been processed. Otherwise, the algorithm proceeds by extracting the first element of the '*remaining*' map and the rest of the elements, referred to as '*head*' and '*tail*'. Additionally, the '*head*' is checked whether it was already visited and processed to avoid repeated iterations. If it was, the function continues to recursively process the '*tail*', effectively skipping '*head*'. In case it was not, the algorithm identifies subsets and supersets of '*head*' within the '*tail*'. The evaluation of subsets and supersets only considers the messages, not the boolean values of each message, and subsets with supersets in this context are represented from the perspective of the **current** iteration '*head*' value. Such identification helps in knowing which sets are not "captured" within the '*head*' set and will be processed later via the '*newRemaining*' variable by filtering the '*tail*' to remove the found subsets.

Then, if any supersets are found, we must identify the highest-order superset that the '*head*' belongs to, i.e. a superset that is the largest possible superset within all found supersets for the '*head*'. Because the algorithm looks for supersets, we can recursively call itself by passing supersets as '*remaining*' and '*visited*' as an empty Map. This results in finding the largest superset because with every iteration '*head*' points to another superset in the list while every time again identifying subsets and supersets of the new '*head*'. As a result, every identification of a superset will again trigger a recursive call to look for the highest-order superset, and this will repeat until there are no supersets found. Eventually, a highest-order superset is returned because only the head element is taken, and it will always be the superset that has no other supersets in the original list. Before evaluating the '*head*' with found superset, the '*head*' and discovered subsets are joined together into a single Map, and the highest-order superset messages are updated based on the messages' boolean values within the head and found subsets via '`updateBooleanAndTermValues`' helper function.

Next, because the highest-order superset may have been already discovered, it should be verified since up until now the '*visited*' map was not considered. As such the function '`compareAndUpdateSets`' 18 is called that will compare a given key-value message tuple with a message map and update or create a new map. In this case, '*biggestSuperset*' is compared to the existing '*visited*' values and based on the result a map is returned that is added to '*finalSets*'.

In cases where no supersets are found, the head is treated as the superset with some number of subsets within the '*remaining*' map. This means the step of identifying the highest-order superset is omitted, but the remaining steps, including checking whether the head was already evaluated, are still performed by again calling '`compareAndUpdateSets`' 18 function. That is because we still need to know whether the head was already observed when analyzing other sets or not, and consequently update boolean message values. The only difference is the missing consideration for supersets from the '*head's*' perspective. Eventually, every key-value map pair will be processed and removed from the '*remaining*' map which will return the accumulated '*visited*' map.

The algorithm '`compareAndUpdateSets`' 18 on its own implements a logic that will recognize if a given set was already evaluated at some point before. It is exclusively used by '`findSupersets`' 17 algorithm which uses the logic here to identify if a given '*head*' or '*biggestSuperset*' was already observed. However, the algorithm '`compareAndUpdateSets`' does not differentiate between them since it does not need to.

When '`compareAndUpdateSets`' is executed, it first updates the given '*map*' boolean values based on the found subset values via '`updateBooleanAndTermValues`'. Then the '*map*' is compared with '*visited*' to identify if all of the values in '*map*' exists anywhere within a single set of '*visited*'. If the given '*map*' does exist, then we must determine what key-value map pairs '*map*' belongs to in '*visited*'. Based on that we again re-evaluate the boolean values of each message and merge the keys of '*map*', '*subsets*', and '*visitedMatch*'. This results in a new key-value map pair which will be combined with filtered map '*visitedFiltered*' to effectively overwrite the old '*map*' found in '*visited*' and produce an updated map. Finally, the iterative process continues with the new remaining and visited maps by calling again the '`findSupersets`'.

Algorithm 18: Compare and Update Sets

Input: set: Map[List[String], Set[(String, Boolean)]], subsets: Map[List[String], Set[(String, Boolean)]], visited: Map[List[String], Set[(String, Boolean)]]

Output: Map of (List of Strings, Set of Tuples (String, Boolean))

```
1  Function compareAndUpdateSets(map, subsets, visited):
2      newVal ← updates all map.values boolean values via
         updateBooleanAndTermValues(subsets.values);
3      if map.values is already in visited.values then
4          visitedMatch ← filter visited to get all messages that are in map;
5          visitedFiltered ← filter visited to get all messages that are not in map;
6          newVal ← updates all visitedMatch.values boolean values via
             updateBooleanAndTermValues(newVal);
7          newKey ← map.keys ∪ subsets.keys ∪ visitedMatch.keys;
8          newSuperset ← Map(newKey, newVal);
9          newVisited ← visitedFiltered ∪ newSuperset;
10         return findSupersets(newRemaining, newVisited);
11     else
12         newKey ← map.keys ∪ subsets.keys;
13         newSuperset ← Map(newKey, newVal);
14         newVisited ← visited ∪ newSuperset;
15         return findSupersets(newRemaining, newVisited);
```

In case it was not discovered, only the keys are updated to include both the given '*map*' and '*subset*' keys. A new key-value map pair is created by taking new keys with '*newVal*' since it is already updated. We already know that the given '*map*' is a potentially new superset that still has not evaluated. This new pair is added to the '*newVisited*' list, and the iterative process continues by calling '`findSupersets`' with the '*newRemaining*' and '*newVisited*'.

Overall, the combined algorithmic process of '`findSupersets`' 17 and '`compareAndUpdateSets`' 18 represents a greedy approach that combines actor types into the biggest existing encompassing type. Such an approach works well when the actor system utilizes inherited actors since identifying and combining each actor's messages will produce a dedicated type for a family of actors that are derived from the same base actor traits/classes. However, if actor classes inherit possible actor states from multiple sources with different message types or every actor is using various combinations of messages that are shared between numerous actors, then in the worst-case scenario this would produce type definitions equal to the number of actor classes in the system with duplicate messages types between each type definition. On the other hand, following the **Actor Responsibility** assumption, the ideal scenario would also produce type definitions less or equal to the number of actor classes/traits, but each definition would contain unique messages specific to that actor's responsibility. One of the ways to improve the algorithm would be to evaluate the supersets and encompassing sets as they are discovered instead of having them processed again since the '*remaining*' map is not filtered to remove observed supersets.

It should be noted that the algorithm 17 has a list of helper functions that are used within the algorithm. These functions are generally self-explanatory and used to return a boolean value, and as such are not explained in detail. The functions are: `isSubset`, `isSuperset`, `containsStringSetTuple`, `equalsStringTuple`, `updateBooleanAndTermValues`.

3.5.2 Message Type Definition

Once the message types are known across the actor system, each message must be reconstructed with the respective type in a new object containing the message definitions. Since types are discovered based on actors' message usage, each new object will indirectly reflect the actors that are grouped under this type. The algorithm 19 details the process for reconstructing the new type objects.

The main goal of the algorithm is to define a new type of object through syntax tree reconstruction that captures these messages, including their parameters and structure, and returns this new object definition. The final object and its message types are then saved in a separate file, containing a detailed list of the message objects used in the system. The function '`createMessageObjects`' takes two input parameters: a '*superset*', which represents a list of actor names and messages that belong to this superset, and '*mstList*', a list that represents discovered different types of message definitions. The result of the function is the definition of a new message expressed as a syntax tree that encapsulates all relevant message parameters and their internal structures.

The algorithm starts by using two helper functions to set up default values representing the new type name and the name of the object class that will contain the messages. The function '`createMessageObjectType`' produces the type name by deriving it based on the actor names contained in the superset, while '`createMessageObjectName`' generates the name by combining the actor names into a single string in a pascal case format [45]. An additional helper function '`refactorParamsActorRef`' will be used later to process parameters and ensure that the actor reference type is consistently applied to all message parameters that are of type '**ActorRef**'.

Then, the algorithm iterates over each message in the superset, skipping any message that is equal to '*self*', which is an indicator for an actor behavior change where message parameters are parameters used for a behavior change. For each valid message, the algorithm attempts to identify its corresponding type from the '*mstList*', and then distinguish between it being a message class and companion objects in '*mstListFiltered*' and '*mstListCompanion*', respectively. If a message type is found, the algorithm refactors it, ensuring that the '*replyTo*' parameter is inserted, replacing the '*sender*' parameter, and adjusted with the correct '**ActorRef**' type using the '`refactorParamsActorRef`' function. This ensures that any actor message that used to use the '*sender*' parameter will use the correct type.

Finally, a new body is created for the message, using the abbreviation type and the original message definition body that may contain additional functions or default values. The result is either a new class or an object that depends on different scenarios based on the structure of the message. If the message has parameters, it is defined as a class. If it has no parameters but has a Scala companion object [64], both the class and object are created. Lastly, if neither message parameters nor a companion object exist, a case object is created. As a result, companion objects are preserved to ensure that the message and actor system retains all original interactions with each other, particularly for messages that are designed to have both a class and a companion.

Algorithm 19: Create Message Type Objects

Input: superset: Tuple(List of String, Set of (String, Boolean)), mstList: List of MessageType
Output: Object definition (Defn.Object)

```
1  Function createMessageObjects(superset, mstList):
2      abbreviationType ← createMessageObjectType(superset);
3      nameObj ← createMessageObjectName(superset);
4      statsObj ← list of Stat with Trait definition (name equal to abbreviationType));
5      foreach (message, bool) in superset.messages do
6          if message is not "self" then
7              msType ← MessageType(empty);
8              mstListFiltered ← filter mstList where name equals message and isClass is true;
9              mstListCompanion ← filter mstList where name equals message and isClass is false;
10             if mstListFiltered is not empty then
11                 msTypeOld ← mstListFiltered.head if exists, otherwise None;
12                 if msTypeOld.isDefined then
13                     if bool then
14                         msType ← remove 'sender' from msTypeOld.params and refactor
                            msTypeOld with refactorParamsActorRef;
15                     else
16                         msType ← refactor msTypeOld with refactorParamsActorRef;
17             newBody ← Body(abbreviationType, msType.body);
18             if bool then
19                 msType.params ← replyTo constructor parameter of type
                    ActorRef(abbreviationType) ∪ msType.params
20                 statsObj ← statsObj ∪ Defn.Class(message, msType, newBody);
21             else if msType.params is not empty then
22                 statsObj ← statsObj ∪ Defn.Class(message, msType, newBody);
23             else if mstListCompanion is not empty then
24                 msCompanion ← mstListCompanion.head;
25                 statsObj ← statsObj ∪ Defn.Class(message, newBody);
26                 statsObj ← statsObj ∪ Defn.Object(msCompanion.name, msCompanion.body);
27             else
28                 statsObj ← statsObj ∪ Defn.Object(message, newBody);
29     return Defn.Object(nameObj, statsObj);
```

3.5.3 Actor Class Refactoring

After message type definitions are created, then the actual actors are refactored. The process of refactoring is implemented in a function '`refactorBody`' 20 that essentially only looks at functions that return '**Receive**' type, is named '`receive`', or is called by a function that does return '**Receive**' type by utilizing call graph. Once such function is discovered in an actor class, they are refactored by calling another function '`createBehaviourObject`' which looks at intermediat actor's states and interactions that an actor contains and refactors message and actor references with appropriate types and message parameters that may have been added (i.e. parameter '*replyTo*'). The function outputs a new object or class definition that includes the refactored actor logic. The primary goal of this function is to convert untyped actors and their behaviors into typed ones, while ensuring compatibility with existing structures and maintaining proper message handling.

Looking deeper at the function, the first step involves initializing a custom transformer, which is later used to transform all actor reference parameters within the class into their typed equivalents. In this context, a transformer is a simple recursive function that explores a given syntax tree while pattern-matching each node, and performing some logic once a match is found. The main loop then iterates over the '*classTrait*' that actor body, examining each node in the syntax tree. If the node represents a function that either returns a '**Receive**' type or is explicitly named '*receive*', it is refactored by calling '`createBehaviourObject`' and passing down the function with its parameters, but with a new name of a function '`apply`'. This is necessary because, in the refactoring process, default receive functions are moved into an apply method which allows for the function to handle previously existing constructor parameters while ensuring that the whole actor class can exist as an object definition. Such resulting output of an actor class falls in line to the functional programming paradigm that the Akka library suggests using. However, this also necessitates that actors should be stateless, hence the existence of the initial assumption **Stateless Actors** 3.2.

When a function neither returns the '**Receive**' type nor is named '*receive*' without an explicit return type, the next step is to check whether this function is invoked by another function that does return the '**Receive**' type. If it is, the function undergoes refactoring in the same manner, but its name remains unchanged. This process operates under the assumption that any function called by one returning '**Receive**' is likely involved in message handling and message-processing logic. Consequently, these functions are also refactored to ensure they contain the correct message and actor reference types. On the other hand, if a function is abstract and either returns '**Receive**' or is named *receive*', only the function's definition is updated to return the appropriate actor behavior type, specifically '**Behavior[T]**'. After applying these changes, the refactored functions are appended to the '*newBody*' variable, which stores the list of nodes gathered during the traversal of the actor's body.

Eventually, the function concludes by determining whether to return a class, trait, or object definition based on the existence of class parameters and whether a '`receive`' function was found. If the class definition includes the '**abstract**' keyword or if no default '**Receive**' function was found while the class contains constructor parameters, then the result will be a class. If the actor was originally defined as a trait, it remains as a trait definition. In all other cases, the actor is returned as an object.

Through '`refactorBody`' function, a number of helper functions are used. Like in '`createMessageObjects`' 19 function, '`createMessageObjectName`' and '`createMessageObjectTypeName`' are used to produce actor type name. In addition, the function '`refactorParamsActorRef`' is responsible for ensuring that actor reference parameters are correctly transformed into typed actor references. Lastly, one of the core functions that handles refactoring of the message processing actor functions is '`createBehaviourObject`'.

The function essentially loops through the body of a given function looking for a few specific syntax structures that indicate that a message is being send or processed. If found, then with the help of custom transformers every message type definition, actor reference and message parameters are refactored to contain appropriate types by reconstructing syntax tree node and replacing the one that was matched. Since message targets were already resolved during the construction of the '**Intermediate Actors**', determining the target actor types when sending a message simply requires looking up the observed message in the relevant '**Intermediate Actors**' definition. Once the target actor class is identified, the actor type is retrieved from the supersets, which is again formatted using the helper function '`createMessageObjectTypeName`'.

A comprehensive breakdown of the '`createBehaviourObject`' algorithm, along with its associated helper functions and more specialized methods for handling various syntax tree expressions and edge cases, is provided in Appendix A.

Algorithm 20: Refactor Body

Input: isTrait: Boolean, classTrait: Tree, ia: IntermediateActor, actorSet: Tuple(List of String, Set of (String, Boolean)), supersets: Map of (List of String → Set of (String, Boolean)), callGraph: List of FunctionNode

Output: Function Definition

```
1  Function refactorBody(isTrait, classTrait, actorSet, supersets, callGraph):
2      transformerClassParams ← new Transformer(Node){...};
3      noReceive ← true;
4      importMsObj ← import definition for actorSet;
5      newBody ← empty list of Stat;
6      foreach node in classTrait.body do
7          if node is function definition and returns Receive or is called by function that returns
             Receive then
8              if node returns Receive or is named 'receive' without a return type then
9                  noReceive ← false;
10                 newBody ← newBody ∪ createBehaviourObject(node, node.mods,
                     "apply", classTrait.params, actorSet, supersets, true);
11             else
12                 newBody ← newBody ∪ createBehaviourObject(node, node.mods,
                     node.name, classTrait.params, actorSet, supersets, false));
13         else if node is abstract function and (returns Receive type or is named 'receive') then
14             actorTypeName ← createMessageObjectTypeName(actorSet);
15             newDecltpe ← Behavior[correspondingActorTypeName] definition;
16             node.declType ← newDecltpe newBody ← newBody ∪ node;
17         else
18             newBody ← newBody ∪ node;
19     newInits ← node.inits without 'Actor' and empty constructor parameters;
20     newParams ← transformerClassParams(classTrait.params);
21     if classTrait.mods contains "abstract" or (noReceive and classTrait.params is not empty)
         then
22         return Class(classTrait, newInits, importMsObj + newBody);
23     else if isTrait then
24         return Trait(classTrait, newParams, newInits, importMsObj + newBody);
25     else
26         return Object(classTrait.mods, classTrait.name, newInits, importMsObj + newBody);
```

3.6 Other attempts

Throughout the implementation process, various existing libraries for extracting a call graph, class inheritance graph, and dependency graph were considered in order to reduce implementation effort and ensure coverage of edge case scenarios. However, most of the tools presented limitations that rendered them unsuitable for this project's needs.

One of these tools we attempted to utilize is an open-source CLI tool "**callGraph**" [3] used for generating call graphs from source code from an array of code languages. While promising, "**callGraph**" struggled to recognize many Scala-specific expressions and syntactic sugar, often missing function calls that were important for accurately mapping the communication between actors. Additionally, as a CLI tool, it only outputs graphs in external file formats, which would require additional effort to integrate into a broader refactoring solution to utilize the resulting file.

Another tool we considered was **Scala Sculpt** [28], a compiler plugin designed to analyze the dependency structure of Scala source code. Scala Sculpt extracts extensive information about dependencies on every level, but its unconventional and verbose output makes it difficult to use directly. The output would have required custom logic and considerable effort to parse and group the extracted data into meaningful structures for automated refactoring. Furthermore, Scala Sculpt is an unfinished and unmaintained project, adding risk to its adoption due to potential bugs or lack of updates in the future.

We also explored using the **WALA** [35, 23] library for static analysis on JVM bytecode. Since Scala compiles to JVM bytecode, WALA could theoretically be used to analyze Scala code indirectly. However, the complexity of using WALA to parse and interpret bytecode generated from Scala presented significant challenges. Not only would it require extensive effort to adapt WALA for Scala, but its bytecode-based analysis would limit direct access to Scala-specific constructs, making it less ideal for accurately capturing actor communication patterns and interactions. As demonstrated by the paper ”Targeted Test Generation for Actor Systems” [26], for similar reasons requiring substantial effort, they only support Java Akka library instead of both Java and Scala.

While evaluating these options, there was a noticeable lack of robust, well-maintained tools in the Scala ecosystem for static code analysis capable of extracting dependencies and interactions from source code. The libraries that were found were either too generalized to meet the specific needs of this project, too simplistic to capture the complexities of a distributed actor system, or abandoned by their maintainers. This scarcity of appropriate tools reinforced the need to develop a custom solution tailored to the problem of refactoring untyped actor systems into typed ones.

3.7 Constraints and considerations

When analyzing and refactoring actor-based systems, there are several constraints and considerations that must be taken into account, both due to the implementation limitations and the innate differences between typed and untyped actor systems.

One aspect to consider is how the actor state is handled. We assume that actors would not have an internal state, as the goal was to align to a stateless, immutable, and in general more functional programming model [22]. However, it is possible that an actor was implemented with stateful values. To deal with that, an approach is required for a stateful actor to become stateless. Although not yet implemented, a proposed solution would involve creating a dedicated actor state class that encapsulates the various state variables of an actor. This class would be instantiated at runtime, and the object would be passed through function calls whenever state information is needed. By consolidating state values in a separate class used as a container, we ensure that actor instances remain stateless aligning with our initial assumptions and functional programming principles. Unfortunately, during evaluation a large majority of benchmark actors were indeed stateful, and as such each benchmark was manually modified to include these actor state classes containing all actor stateful variables that are used in a given benchmark.

Another important constraint relates to actor hierarchy. In the untyped actor system, actors can be spawned in a flat space, meaning that any actor could spawn another actor without considering a parent-child relationship, usually resulting in an actor system hosting a large number of actors in a flat space. However, the Akka Typed requires actors to spawn children within a hierarchical context, where each actor spawns children from its own system context. This introduced a challenge, as actors that previously had access to the system context needed to be adjusted to spawn child actors from their own context. Currently, there is no automated logic in place to handle the restructuring of actor hierarchies during refactoring. As a result, the actor system is expected to already meet actor hierarchy criteria to ensure that actors are correctly refactored to follow the Akka Typed hierarchical structure.

Despite these constraints, there are a number of limitations that the current implementation, and by extension the proposed algorithm, can not handle, either due to implementation issues, or logical errors.

3.7.1 Limitations

The list of limitations here arose from the discovered issues during evaluation, implementation logic that didn't consider all possible expressions or the innate differences between Akka Classic and Akka Typed that are not yet handled [51].

- **Lack of Refactoring for Ask Patterns**: Although the communication flow graph identifies ask patterns (i.e. when actors request responses from other actors), the current implementation does not refactor these patterns. The message and message type information is recognized and used, but ask patterns are left unchanged during refactoring.
- **Ambiguity in Actor References**: Actor references can not be ambiguous. An actor reference variable can not reference multiple actor types, because otherwise, it becomes impossible to determine the exact actor being referenced. Additionally, the target actor reference variable or instance itself must be used directly to invoke a `'tell', 'ask'`, or '`forward`' function (and their equivalent '`!`' and '`?`' syntactic sugars), and not call some other class, object or function that will return an '**ActorRef**' instance.
- **Reserved Keywords and Syntax Limitations**:
  - The reserved keyword '*sender*' is assumed to always reference the sender of a message.
  - When defining '`Props`' object in Akka Classic and using the keyword '`new`' to specify an actor class, then any subsequent parameters can not use the keyword '`new`' as they will be incorrectly recognized as a potential actor class.
  - Actor constructor parameters of type '`ActorRef`' cannot include type parameters (e.g., '`ActorRef[Message]` in Akka Classic.
  - When parameters are passed to a message object during sending or message instantiation, there should not be variables or functions containing a string '*self*', as it is reserved to recognize actor self-reference.
- **Message Type and Parameter Constraints**: Limitations listed here apply only to the Akka Classic since they are all related to the recognition of message types and their parameters.
  - Literal message types, such as string literals like "*EXIT*", are not recognized as message types and are ignored. This leads to incomplete refactoring for actors that rely on such literal types.
  - Messages with type parameters (e.g., '`Message[T]`') are not supported.
  - Message definitions cannot have standard types like `'Boolean', 'Int'`, '`String`', etc. as their names.
  - Message constructor parameters can not be of type '`Any`'.
  - Message definition must be within an object or a class (and not in a trait or some function).
  - Message parameters must be defined in the constructor of the message object/class definition.
  - Message definitions can only inherit a single trait or class with no mixins.
  - The body of each message type definition will not be analyzed and consequently will be ignored (since they are treated as static objects).
- **Function and Message Call Restrictions**:
  - Functions cannot call themselves because the call graph will not contain the edges that point to itself. As long as message processing functions are not recursive, this should not be an issue.

  - Props instances must be assigned to '`val`' variables, and all props constructor parameters must be passed with their true values directly (without any level of indirection or external function calls).
  - When invoking `tell`, `ask`, or '`forward`' actor call functions (and their equivalent '`!`' and '`?`' syntactic sugars), the first argument must be a message object directly, not the result of another function or object that returns a message.
  - In classical Akka, when changing the actor context/behavior and passing the to-be receive function, the '`context.become(...)`' must be called within the function that returns receive type, not within some other function that is called instead of '`context.become(...)`'.

- **Mutable Objects**: Mutable objects should not be passed between actors, as they are not accounted for in the refactoring logic. Refactoring assumes that actors interact through immutable messages and their parameters.
- **Immutability and Functional Programming Limitations**: Implicit values should not be passed in the same code line as tell, ask, or forward commands due to limitations in recognizing syntactic sugar (e.g., '`obj.func(...)(...)`').
- **Multiple Inheritance Limitation**: No trait, class, or object will override a function that is defined in more than one parent trait or class. If node A and node B in the hierarchy graph define a function with the same name, and a child node extends both parents and overrides that function, the call graphs can produce incorrect results which negatively affects the refactoring process.
- **No behavior reversal**: Currently the implementation does not take into account the Akka Classic capability to reverse an actor's behavior via the '`context.unbecome()`' function. This is one of the innate differences between typed/untyped libraries as such pattern is not available in typed library.
- **No parent-child relationship recognition**: The current implementation lacks recognition of parent-child relationships between actors, despite the ability to resolve actor references and detect instantiation. While adding this capability would possibly enhance the modeling of actor hierarchies and improve refactoring accuracy, particularly for message passing and fault tolerance, it is not currently implemented.

3.8 Implementation Setup

The implementation used the following libraries:

- `scala` - 2.13.12
- `akka-actor; akka-testkit; akka-actor-typed; akka-actor-testkit-typed` - 2.8.2
- `scalatest` - 3.2.13
- `spray-json` - 1.3.6
- `scalameta` - 4.5.13
- `graph-dot` - 1.13.0
- `sbt` - 1.9.7
- `JDK` - 21.0.2

Chapter 4

Evaluation & Results

The evaluation framework of this research aligns with research questions and will explore the developed source code refactoring in three domains: Effectiveness, Efficiency, and Applicability. Evaluating the refactored software across these three domains will present a comprehensive assessment of its capabilities and potential with respect to viability, performance, and practical applicability. The compounded result of each domain will allow us to draw meaningful conclusions, present the current limitations, and outline future work.

4.1 Evaluation Questions

Based on the three evaluation domains, each of the evaluation questions will fall into one of them, and as such, the questions are grouped below to represent their respective domains:

- **Effectiveness** - as the name suggests, the domain will be aimed at measuring the ability of the software to produce effective changes and improvements to the refactored code while ensuring similar or better code quality, essentially representing the software's ultimate goal. For that, the code would be evaluated based on a set of evaluation metrics and then tested through a set of unit tests for each of the benchmark actor programs [24]. As such, the evaluation questions for this domain are as follows:

    1. Are there cases where the software produces refactored code that introduces new bugs or unintended behavior?
    2. How does the refactored code's quality compare to the original code?

- **Efficiency** - as opposed to effectiveness, this domain will evaluate the speed and resource consumption necessary to provide the results that the effectiveness domain considers, providing insights into practical utility. For that, a diverse set of actor benchmark programs will be compiled in several corpuses that then will be used when measuring refactoring software's efficiency. The evaluation questions for this domain are as follows:

    1. What is the impact on resource (CPU, memory) consumption of the software after the refactoring?

- **Applicability** - at this point, it is important to measure how applicable is the refactoring software with its current limitations. As such, based on a set of preconditions and constraints, this domain will measure the ability of the refactoring software to refactor and maintain functionality of a select large-scale open-source project that can benefit from the refactoring software. Below you can find the evaluation questions:

    1. Is the refactoring software capable of refactoring and preserving the functionality of a large-scale open-source project?

4.2 Evaluation setup

In this section, we present the setup used to evaluate the effectiveness, efficiency, and applicability of the refactored actor-based systems. The evaluation is conducted using a set of benchmarks from the Savina Benchmark Suite [24], which provides a wide range of benchmarks to test parallelism, concurrency, and communication overhead in actor-based systems. Specifically, the benchmarks were chosen to test both the effectiveness and efficiency of the refactored code when compared to the original, whereas applicability is tested by refactoring real-world application and evaluating its efficiency.

In addition to Savina benchmarks, we will include four simple benchmarks derived from open-source GitHub projects that implement stateless actor systems, more inline with functional programming paradigm. The aim of these benchmarks are to see the effect that stateful actor systems have on the refactoring quality as Savina benchmarks are primarily stateful.

4.2.1 Benchmark applications

The Savina benchmark suite provides a collection of benchmarks designed to assess the performance of various conditions of actor-based systems. These benchmarks simulate real-world scenarios and computational challenges that are representative of typical workloads in distributed and parallel systems. For this evaluation, we selected 13 benchmarks out of 28 available: 10 parallelism benchmarks, 2 concurrency benchmarks, and 1 micro-benchmark. Each benchmark offers distinct computational patterns, making them suitable for testing various aspects of the actor model, such as communication, synchronization, and task distribution. The list of chosen benchmarks can be found in the table 4.1

Symbol	Name	Pattern being measured
Parallelism		
APSP	All-Pairs Shortest Path	Phased computation; Graph exploration
ASTAR	A-Star Search	Message priority; Graph exploration
NQN	NQueens first N solutions	Message priority; Divide-and-conquer style parallelism
TRAPR	Trapezoidal Approximation	Master-Worker; Static load-balancing
PIPREC	Precise Pi Computation	Master-Worker; Dynamic load-balancing
RMM	Recursive Matrix Multiplication	Uniform load; Divide-and-conquer style parallelism
QSORT	Quicksort	Non-uniform load; Divide-and-conquer style parallelism
RSORT	Radix Sort	Static Pipeline; Message batching
UCT	Unbalanced Cobwebbed Tree	Non-uniform load; Tree exploration
OFL	Online Facility Location	Dynamic Tree generation and navigation
Concurrency		
SBAR	Sleeping Barber	Inter-process communication; State synchronization
PCBB	Producer-Consumer with Bounded Buffer	Multiple message patterns based on Join calculus
Micro-benchmark		
PP	Ping Pong	Message delivery overhead

Table 4.1: List of Savina Benchmarks [24] used in the evaluation: 10 parallelism benchmarks, 2 concurrency benchmarks, 1 micro-benchmark.

The benchmark selection was guided by the need to capture a diverse range of actor-based computations that reflect real-world scenarios while maintaining a focus on parallelism as it is both reflective of the real-world patterns and generally difficult to implement correctly. As a result, the initial intention was to use all 13 of the original Savina parallelism benchmarks. Still, two of them

could not be refactored correctly without modifying the benchmark source code and altering the actors and their behaviors, and one had source code comments indicating deadlocking within the benchmark which could have impacted evaluation results, and as a result, it was omitted.

The original set of concurrency benchmarks, containing 8 in total, includes one specific and three general communication patterns that have more specific variations within them: Multiple message patterns based on Join calculus, Reader-Writer concurrency, Inter-process communication, and Synchronous Request-Response. For this evaluation, only the benchmarks focused on Inter-process communication state synchronization and Join calculus were selected for refactoring, as their implementations are representative of the broader concurrency set and feature a higher number of actors and messages, making them ideal for this study. The remaining patterns, such as Synchronous Request-Response and Reader-Writer concurrency, are somewhat covered by parallelism benchmarks, and adding more specialized benchmarks would require significant time without yielding substantial new insights. Moreover, the focus of the refactoring is on actor system complexity—specifically, the number of actors, their states, behaviors, and interactions—rather than their computational efficiency. This is because the refactoring process does not alter the internal logic of actors but instead identifies, groups, and refactors actor types and their interactions. Consequently, benchmarks with a larger number of actors, more diverse interactions, and a variety of message types are the most valuable for this evaluation.

Only the ping pong benchmark from the original 7 micro-benchmarks was selected as a baseline benchmark that has only two actors interacting with clearly defined behaviors and states. Since the ping pong benchmark actors are not performing computation, as a result being fairly simplistic, and do not modify their state without being triggered by message interactions, it was perfect for evaluating whether the refactoring could accomplish bear minimum and refactor actor types with their respective interactions and messages.

From the original set of seven micro-benchmarks, only the Ping Pong benchmark was selected as it provides a straightforward baseline with just two actors exhibiting clearly defined behaviors and states. Since the benchmark actors do not engage in complex computations and only modify their states through message exchanges, it is ideal for assessing whether the refactoring process can handle the fundamental task of identifying and refactoring basic actor types along with their corresponding interactions and messages. Such minimal complexity provides a clear test to confirm whether the refactoring could handle the fundamental task of identifying and restructuring actor types and their interactions at their most basic level.

Stateless Benchmarks

The need for the stateless benchmarks arises from the observation that all selected Savina benchmarks rely on stateful actors, which may significantly affect code quality after refactoring due to introduction of actor state class that will contain stateful variables. By including stateless benchmarks, we aim to evaluate the effects of refactoring on actor systems without the complexity of managing state, providing a clearer understanding of how the refactoring impacts code quality in simpler, stateless scenarios.

The four stateless benchmarks are derived from a GitHub repository containing exercise tasks for learning Akka Classic [13], specifically the ”changing actor behavior” and ”child actor exercise” tasks. These exercises were adapted to allow for precise metric evaluation and high-load simulations. The benchmarks can be found in the table 4.2

These additional benchmarks will undergo the same evaluation process as the Savina benchmarks to ensure consistent comparison and to measure the effects of refactoring on both stateful and stateless actor systems. This will help determine whether the introduction of state has a significant negative impact on code quality, particularly maintainability and understandability.

Name	Benchmark description
Counter	A single actor system that tracks the count of an integer value based on messages received from an external environment.
FussyKid	A two-actor system where one actor randomly sends one of two possible messages to another actor, which then changes its behavior based on the previous message and tracks the behavior change.
VotingSystem	A one-to-many actor system where multiple copies of a voting actor cast votes, and a vote aggregator actor collects all votes, ensuring that all actors have voted and responded.
WordCounter	A three-actor system involving one-to-many interactions, where a master actor creates worker actors to distribute tasks, and aggregates the results from the workers to respond to the original task initiator.

Table 4.2: List of stateless benchmarks adapted for evaluating refactoring impact on actor systems [13].

Effectiveness Domain

To assess the effectiveness of the refactoring process, unit tests were created and executed on both the original and refactored versions of the code for all 13 selected benchmarks. The primary goal of this step was to identify any unintended behavior or bugs introduced during the refactoring process by verifying that the same unit test passed both the original and refactored code. Unit tests for each benchmark were designed to ensure the successful termination of the benchmark with expected results. By comparing the results of tests for both code versions, any potential discrepancies in behavior were identified. It is important to note that the unit tests do not exhaustively test every possible actor behavior, individual actor states, and responses for each possible message and message configuration. Rather, each benchmark under test was treated as a black box with known input and configuration values, and expected output values, and as long as the benchmark was successful with the expected resulting value, then it was treated as success.

To evaluate the effectiveness of the refactoring process, unit tests were created and executed on both the original and refactored code versions for all 13 selected benchmarks. The primary goal was to detect any unintended behavior or changes introduced during refactoring by ensuring that the same unit tests passed in both the original and refactored code. These unit tests were designed to verify that each benchmark completed successfully and produced the expected results. By comparing the outcomes of the tests for both code versions, any deviations in behavior were identified. It's important to note that these tests did not aim to cover every potential actor behavior, state, or response for all message configurations. Instead, each benchmark was treated as a black box with predefined input, configuration, and expected output values. As long as the benchmark ran successfully and yielded the expected result, it was considered a pass.

In addition, to evaluate the code quality of the refactored versions, both code versions were statically analyzed and compared based on a set of metrics and rules to guide static code evaluation. The three metrics that were used for this evaluation are explained below:

- **Cohesion-Coupling Balance**: This metric assesses the relationship between an actor class's cohesion, which reflects how closely related its methods and functions are, and its coupling with other actor classes. Coupling is measured through an actor's interactions with others as well as the messages exchanged and the data contained within those messages [15, 27, 18]. A balance of high cohesion and low coupling is preferred, with values exceeding one ($>$ 1) indicating a favorable balance [33]. This metric is designed to evaluate the increased dependencies for each actor class arising from actor and message types, the introduction of actor state class, and the replacement of the '*sender()*' function with an additional message parameter '*replyTo*'.

- **Instability**: As the refactoring process introduces new classes for each actor, it is important to understand how these new dependencies may affect the class's susceptibility to changes stemming from modifications in dependent classes. This, in turn, provides insights into the overall maintainability of the code. To assess this, the Instability metric is utilized to quantify the likelihood of a given actor class undergoing changes, taking into account its dependencies as well as the classes that rely on it [30].
- **Cognitive Complexity**: This metric measures how difficult the code is to understand, effectively measuring the understandability of a code [12, 9]. Such measurement is important as it reflects a developer's capacity to interpret, utilize, and build upon the refactored code. If the code lacks clarity, developers may need to invest additional effort into further refactoring to make it useful.

These three metrics cannot be applied directly to the entire source code as the presence of existing benchmark infrastructure and calls to standard libraries would distort the results. To ensure the evaluation remains focused on the benchmarks themselves, we will restrict the scope of the assessment. Specifically, only the actor classes introduced by the benchmark, along with their configuration or custom classes specific to that benchmark, will be evaluated. Interactions with standard Java or Scala libraries, types, or functions, as well as any interactions with the underlying Savina benchmark suite (such as utility functions, abstract classes, or inherited methods not directly involved in the benchmark), will be excluded from consideration. However, any new class or object generated through the refactoring process will be included, including type definitions. Message definitions will be excluded from the analysis since no new messages are created or defined, only the data exchanged within messages will be considered for Data Coupling evaluation. Additionally, an actor class is considered dependent on another actor class if it creates child actors or sends messages to a target actor class, which implies a dependency on that class for message handling and processing.

Together, these metrics address the second evaluation question within the effectiveness domain by assessing the impact of refactoring on code quality. Meanwhile, unit tests are designed to tackle the first evaluation question, ensuring that the refactoring process does not introduce any unintended changes or errors in the behavior of the benchmark's actors.

Efficiency Domain

The efficiency domain focused on evaluating the impact of refactoring on resource usage, specifically CPU, memory consumption, and runtime performance. The same set of 13 benchmarks was executed under both the original and refactored code on the same hardware, and resource usage was recorded during the execution of each benchmark over ten iterations. The purpose of such evaluation is to see if the refactoring introduced any noticeable inefficiencies that could degrade performance, and identify any additional overhead generated by the refactoring process. By examining resource consumption, the evaluation would reveal whether the refactored code negatively impacted system efficiency or remained comparable to the original, ultimately ensuring that the benefits of refactoring did not come at the cost of significant performance degradation. With these goals in mind, the following measurements are taken during the execution of each benchmark:

- **CPU usage**: The percentage of processor time consumed by a benchmark was measured to determine any significant changes in computation possibly introduced by the refactoring.
- **Memory usage**: Memory usage was monitored to evaluate whether the refactored code leads to increased or decreased memory consumption as compared to the original code.
- **Runtime**: The total time needed to complete each benchmark was measured, allowing for a direct comparison of the execution speed of the original and refactored code.

Each benchmark was executed ten times, and the results were used to calculate the arithmetic mean for each metric. Additionally, the standard deviation of runtime was computed to provide insight into the variability of the execution times across runs.

Each benchmark was executed ten times to ensure consistent and reliable results. For every performance metric, an arithmetic mean was calculated, providing an average value that represents the benchmark's overall performance. In addition to the mean, two key parameters were derived from the runtime measurements: the standard deviation, which indicates the variability in execution times across the runs, and the skewness, which highlights any asymmetry in the distribution of the runtime data. Together, these statistics offer a better understanding of the stability and consistency of the code's performance while highlighting unintentional changes introduced by the refactoring regarding the code's efficiency.

4.2.2 Real-world application

The evaluation of applicability will follow a three-step process: successfully compiling refactored code, running and passing the project's unit tests, and successfully running the service locally. Since the goal of this evaluation is to verify whether the refactoring software can be applied to larger real-world projects, rather than just benchmarks, it is sufficient for this evaluation to confirm only the functionality of the refactored software, but all of the steps in the process must be completed for it to be considered success. However, it is important to note that assessing the refactoring's applicability should ideally be extended to a wider range of substantial real-world projects, rather than relying on a single case study to prove the usefulness of the refactoring software in practical applications. Due to time constraints, only one project will be used for this evaluation.

To evaluate the applicability of the refactoring process, a set of open-source GitHub projects was compiled, with only two meeting the selection criteria: "Hydra" [34] and "Tapir" [4]. The criteria included projects written 95% or more in Scala, using an Akka Classic actor system as the concurrent computing model, having over 200 files and 20,000 lines of code, and receiving updates within the last year. "Hydra" was chosen because it uses an actor system at its core and functions as a standalone service, making verification easier. "Tapir" on the other hand primarily serves as a library and only uses Akka for one of possible backend servers, which made it less suitable for evaluation.

Hydra itself is a real-time data replication platform that leverages the Scala programming language and the Kafka and Akka frameworks. It enables the efficient reception, transformation, and transmission of data streams by decoupling these processes into distinct, manageable components. The platform simplifies the complexities of data streaming by offering a simple REST API, allowing developers to interact with it seamlessly due to abstracted underlying implementation details. Additionally, Hydra is a substantial project, consisting of more than 400 files and over 46,000 lines of code, and is actively maintained.

4.2.3 Technical setup

The refactoring tool, along with all selected benchmarks and the "Hydra" project, was compiled and executed on a system with the following specifications:

- **Processor**: 13th Gen Intel(R) Core(TM) i9-13905H (20 CPUs), 2.6GHz
- **Memory**: 65536MB RAM
- **Graphics Processing Unit**: NVIDIA GeForce RTX 4070 Laptop GPU, 7962MB

4.3 Experimental results

This section provides a detailed presentation of the evaluation results across each domain. The Effectiveness results will discuss findings related to the unit tests, highlighting any changes or issues identified in the refactored code, alongside a comparison of relevant metrics before and after the refactoring. The Efficiency section will include performance metrics and an analysis of resource consumption differences between the original and refactored code. Finally, the Applicability results will outline the success of each evaluation step, including any failures, and look deeper into the underlying causes behind those failures.

4.3.1 Results on effectiveness

In answering the evaluation question "*Are there cases where the software produces refactored code that introduces new bugs or unintended behavior?*", each benchmark was tested by designing unit tests that verifies the functionality of benchmarks and correctness of results. Following the execution of unit tests, 3 out of 13 refactored benchmarks encountered issues. These problems arose either due to inherent differences between the Classical Akka and Typed Akka libraries, leading to compilation failures, or as a result of limitations within the refactoring implementation, which was unable to handle specific expressions or communication patterns accurately.

- **PCBB**: In the benchmark one of the actors stores a message in a local state variable list, and processes it after receiving a message from another actor that requests data. After refactoring, the two actors were assigned different types, but because of the implementation, both were required to process the same message. However, Typed Akka does not allow an actor to process messages belonging to types other than its own, nor can these actors be combined into one type because they handle different messages and serve distinct purposes.

  Typically, such scenarios in a typed actor system are resolved using message adapters or distinct messages for each actor type containing the same data elements [60, 53]. As these solutions were not supported by the refactoring software, the compilation failed due to mismatched message and actor types. To correct this, two lines of code were added that take the data message from a list, extract the data elements, and create a new message with those data elements. This new message is of a type that the target actor expects, and so it is as if the same data message was sent to the target that requested the data.
- **RSORT**: The benchmark is implemented as a pipeline of actors that one after another sequentially process messages. During actor creation, the actor reference is passed down as a constructor parameter which is actually a data object, and one of the data elements in the object contains the actor reference. In such case an actor is referenced through 2 degrees of indirectness, meaning that actor reference type resolution failed since only a single degree of indirectness is supported. This resulted in the unit test for this benchmark failing, and the benchmark had to be manually corrected with the correct actor type for this actor reference.
- **UCT**: In the benchmark, one of the actors has several constructor parameters where one of them references a root node actor of a graph tree and the other references a direct parent node actor, and it was implemented in such a way that the parent parameter can reference both a node and a root node. However, both the root actor and node actor are of a different type, have different behaviors, and serve different purposes which means that after refactoring the actors will be of different type. Because of that, it would be no longer possible for the same parent parameter to reference two different types of actor references. As a result, the unit test failed and the benchmark had to be manually corrected by removing the possibility for the parent parameter to reference a root node actor, the added logic ensured that the actors directly under a root node would not have a parent node parameter set and the root node

reference is used separately when the parent was not set.

This failure occurred because the benchmark's implementation reflects a pattern that is incompatible with a typed actor system. From the refactoring software's perspective, it was a coin flip when deciding which type it should be since that parameter can reference both the node actor and root actor. The refactoring software does not consider such usage pattern as these patterns are not valid in typed actor systems. Automatic refactoring of this scenario would require exploring possible design space, evaluating it, and selecting the most appropriate refactoring to handle this communication pattern.

Two of these failed benchmark tests happened because of the innate differences between untyped and typed actor systems that the Akka library defines, with an additional failure due to the limitations of refactoring software implementation. While these failures demonstrate that the software cannot refactor the actors without considering expressions that may not be possible in a typed actor system, these issues were within the expected limitations of the software and are not indicative of failures demonstrating malformed, unexpected, or wrong actor behaviors introduced by the refactoring.

The unit tests for the stateless benchmarks revealed that there were no direct issues related to incompatible design patterns between the untyped and typed actor systems. However, the refactoring software still encountered challenges due to the limitations outlined in 3.7.1. These issues required manual corrections during the refactoring process but the refactored actor classes maintained their intended functionality, and no unexpected changes to the actor behavior were observed. This indicates that while the refactoring software struggled with some limitations, the overall structure and logic of the stateless benchmarks remained intact post-refactoring.

Effectiveness metrics

After conducting unit tests and correcting the issues found within the benchmarks, the static analysis began to collect the metrics in answering the evaluation question "*How does the refactored code's quality compare to the original code?*". The tables 4.3 and 4.4 demonstrate the measured metrics of benchmark code before refactoring and after, with the table 4.5 containing the change.

The most notable change after refactoring is seen in the Cohesion-Coupling Balance. The refactored code introduces an actor state class, along with unique type definitions and corresponding messages for each actor across all benchmarks. This addition inherently reduces the cohesion of each actor, as new dependencies and calls to apply and getter/setter functions are introduced. As a result, the coupling between classes increases, leading to a rise in instability. With more dependencies, the likelihood of changes in one class affecting others grows, making the refactored code more susceptible to modifications when its dependent classes change. By extension, the reduced cohesion and higher instability suggest that the refactored code worsens code maintainability.

When comparing the changes in metrics, it becomes evident that the cognitive complexity increased consistently across all benchmarks. This rise is largely due to the refactoring process, which introduced additional logic: the creation of actor instances using Scala object classes, the implementation of actor state classes, and changes to the infrastructure responsible for handling actor and actor system shutdowns. Notably, the last modification was uniform across all benchmarks, while the first two showed slight variations, particularly in the RMM and UCT benchmarks. However, the cognitive complexity difference in the UCT benchmark was significantly higher than in other cases. Instead, this increase was largely due to the manual changes made to the code in order to address the issues identified during unit testing. These fixes, which introduced new conditional logic, substantially contributed to the cognitive complexity of the UCT benchmark.

Benchmarks	**Metrics**		
	Cohesion-Coupling Balance	**Instability**	**Cognitive Complexity**
APSP	0.923	0.500	45
ASTAR	1.019	0.667	33
SBAR	1.055	0.588	30
PCBB	1.449	0.583	33
OFL	0.824	0.556	61
NQN	1.274	0.571	38
PP	1.042	0.333	22
PIPREC	1.168	0.429	24
QSORT	0.487	0.667	48
RSORT	0.600	0.250	35
RMM	0.600	0.625	33
TRAPR	0.354	0.571	20
UCT	1.262	0.571	71
Stateless Benchmarks			
Counter	6.000	0.000	6
FussyKid	2.039	0.400	7
VotingSystem	1.729	0.250	3
WordCounter	2.047	0.444	2

Table 4.3: Table detailing effectiveness metric results of the **original** code

Benchmarks	**Metrics**		
	Cohesion-Coupling Balance	**Instability**	**Cognitive Complexity**
APSP	0.287	0.667	51
ASTAR	0.578	0.769	39
SBAR	0.660	0.720	36
PCBB	0.646	0.722	39
OFL	0.537	0.714	67
NQN	0.600	0.727	44
PP	0.481	0.667	28
PIPREC	0.527	0.667	30
QSORT	0.236	0.800	54
RSORT	0.151	0.600	41
RMM	0.335	0.750	37
TRAPR	0.205	0.727	26
UCT	0.630	0.750	81
Stateless Benchmarks			
Counter	6.000	0.500	7
FussyKid	1.985	0.571	11
VotingSystem	1.660	0.625	8
WordCounter	1.810	0.643	8

Table 4.4: Table detailing effectiveness metric results of the **refactored** code

Benchmarks	**Metrics**		
	Cohesion-Coupling Balance	**Instability**	**Cognitive Complexity**
APSP	-0.636	0.167	6
ASTAR	-0.440	0.103	6
SBAR	-0.395	0.132	6
PCBB	-0.804	0.139	6
OFL	-0.287	0.159	6
NQN	-0.674	0.156	6
PP	-0.562	0.333	6
PIPREC	-0.641	0.238	6
QSORT	-0.250	0.133	6
RSORT	-0.449	0.350	6
RMM	-0.265	0.125	4
TRAPR	-0.150	0.156	6
UCT	-0.631	0.179	10
Stateless Benchmarks			
Counter	0.000	0.500	1
FussyKid	-0.054	0.171	4
VotingSystem	-0.069	0.375	5
WordCounter	-0.237	0.198	6

Table 4.5: Table detailing effectiveness metric change from original to refactored code metrics

The changes in the Cohesion-Coupling Balance and Instability metrics are consistent with expectations, as introducing new dependencies naturally has a negative effect on these measures. However, the degree of impact varies from benchmark to benchmark, depending on their specific structure, implementation, and the frequency with which internal actor variables (that then got refactored and included in the actor state class) are accessed or used. Interestingly, the effects of refactoring on these metrics do not appear to correlate with the type of benchmark, that is parallelism, concurrency, or micro-benchmark. While this observation suggests some uniformity in the impact of refactoring, a larger sample size would be necessary to draw more definitive conclusions.

The effectiveness metrics of the refactored stateless benchmarks reveal a pattern similar to the stateful benchmarks: a general decrease in the cohesion-coupling balance. This decline is primarily due to the actor classes becoming more tightly coupled to the actor type definitions and associated messages, which are now defined outside of the actor classes. This new external dependency is also reflected in the instability metric, which shows an increase across all stateless benchmarks. However, in comparison to the stateful benchmarks, the decrease in cohesion-coupling balance is significantly smaller, indicating that stateless actor systems are less affected by this structural change. Instability changes remain roughly the same, except for the Counter benchmark, where instability significantly increased. This increase occurred because, prior to refactoring, the Counter actor had no external dependencies, while afterward, it became dependent on its type definition, thereby significantly raising the instability score.

4.3.2 Results on efficiency

To answer evaluation question "*What is the impact on resource (CPU, memory) consumption of the software after the refactoring?*", each benchmarks' performance was measured over ten runs of each benchmark, and the efficiency evaluation can be seen in the below tables. In the same way as before, tables 4.6 and 4.7 show recorded metrics before and after refactoring, while table

4.8 shows percentile changes compared between the original code and refactored. All of the benchmark runs were executed using the default configuration parameters originally defined.

Benchmarks	**Metrics**			
	CPU (%)	**Memory usage (MB)**	**Runtime (ms)**	**Std. Dev. (ms)**
APSP	03	0067	181.40	020.17
ASTAR	57	0039	380.34	035.14
SBAR	03	0036	209.21	006.43
PCBB	76	0039	330.66	018.88
OFL	09	0169	457.32	044.87
NQN	67	0261	498.74	019.28
PP	03	0034	164.93	011.84
PIPREC	38	0073	099.23	006.25
QSORT	04	1067	12523.62	419.90
RSORT	13	0178	577.05	032.34
RMM	05	0027	467.50	018.08
TRAPR	36	0030	126.44	006.76
UCT	39	2244	732.96	058.69
Stateless Benchmarks				
Counter	2.05	0015	0206.60	015.11
FussyKid	2.1	0014	0240.96	015.05
VotingSystem	3.35	0146	1340.24	073.14
WordCounter	1.94	0521	2874.00	048.72

Table 4.6: Table with memory, CPU, and runtime of the **original** code

Upon examining both tables, it is evident that each benchmark yields distinct and varied results when compared to one another. However, despite these variations, the overall changes in resource usage remain relatively minor, with both positive and negative shifts across the benchmarks. As seen in Table 4.8, when expressed as percentage changes, none of the metrics experienced an increase or decrease greater than 30% post-refactoring, except for two notable outliers: the UCT and QSORT benchmarks in terms of CPU and standard deviation values.

These outliers aside, the variations in resource usage across benchmarks appear reasonable, as no consistent negative trend is observed. Instead, the differences seem more tied to the nature of each benchmark's computational task, with some experiencing a higher impact than others. While these evaluation metrics provide a useful indication of the refactoring's influence on application performance, the results should be viewed with some caution due to the limited scope of the evaluation, both in terms of the number of benchmarks and the number of test iterations.

The efficiency metrics of the stateless benchmarks align closely with the trends observed in the stateful benchmarks. While the percentile change in the metrics after refactoring was more drastic, there was no consistent positive or negative impact on resource usage metrics such as CPU and memory consumption. Some benchmarks showed increased resource usage, while others improved, without a clear pattern emerging. However, one notable result is that the execution time across all stateless benchmarks was better after refactoring, indicating that while the structural changes introduced by refactoring did not uniformly affect resource usage, they did improve runtime performance.

4.3.3 Results on Applicability

The evaluation of applicability was carried out once the open-source project was refactored. The first step in the evaluation was the compilation of the refactored code relevant to the actor system

Benchmarks	**Metrics**			
	CPU (%)	**Memory usage (MB)**	**Runtime (ms)**	**Std. Dev. (ms)**
APSP	03	0048	179.01	008.5
ASTAR	60	0039	395.08	032.64
SBAR	03	0043	215.24	014.89
PCBB	69	0045	325.72	017.37
OFL	10	0135	464.8	014.8
NQN	68	0282	363.22	021.82
PP	03	0043	164.78	015.57
PIPREC	42	0066	128.91	011.77
QSORT	02	1128	11580.4	0125.77
RSORT	14	0187	579.43	039.97
RMM	05	0029	452.16	029.28
TRAPR	38	0034	128.37	007.86
UCT	18	1824	663.01	028.33
Stateless Benchmarks				
Counter	1.55	0019	0157.12	010.01
FussyKid	01	0018	0158.71	019.71
VotingSystem	2.35	0097	1211.66	045.20
WordCounter	2.65	0542	2795.92	070.79

Table 4.7: Table with memory, CPU and runtime of the **refactored** code

Benchmarks	**Metrics**			
	CPU (%)	**Memory usage (%)**	**Runtime (%)**	**Std. Dev. (%)**
APSP	0.00	-28.36	-1.32	-57.86
ASTAR	5.26	0.00	3.88	32.64
SBAR	0.00	19.44	2.88	14.89
PCBB	-9.21	15.38	-1.49	17.37
OFL	11.11	-20.12	1.64	14.80
NQN	1.49	8.05	-27.17	21.82
PP	0.00	26.47	-0.09	15.57
PIPREC	10.53	-9.59	29.91	11.77
QSORT	-50.00	5.72	-7.53	125.77
RSORT	7.69	5.06	0.41	39.97
RMM	0.00	7.41	-3.28	29.28
TRAPR	5.56	13.33	1.53	7.86
UCT	-53.85	-18.72	-9.54	28.33
Stateless Benchmarks				
Counter	-24.39	24.68	-23.95	-33.75
FussyKid	-52.38	21.92	-34.13	30.96
VotingSystem	-29.85	-33.38	-9.59	-38.20
WordCounter	36.60	3.94	-2.72	45.29

Table 4.8: Table with memory, CPU, and runtime expressed in percentile change

used in the application, after which two more steps were supposed to follow. Unfortunately, the compilation of the project was not successful as there were numerous issues with the limitations of the software 3.7.1. Due to current assumptions, the software does not take into account the external environment interactions with the actor system(s) in the application, meaning that there are overlooked interactions, missing message types that were not discovered or incorrectly recognized as such, and some syntax expressions and sugars were not recognized as valid which should have been since they are part of message processing logic. Additionally, the software failed to recognize certain functions used in message handling, misidentified actor reference types, and overlooked inherited functions and values that actor classes rely on. Several new bugs were also discovered, further preventing the refactoring process from functioning as intended.

Given these complications, it is clear that the planned three-step evaluation process could not be completed. This highlights significant gaps in the software's ability to handle complex real-world applications and emphasizes the need for further improvements to handle diverse actor interactions, external dependencies, and more sophisticated syntax patterns. Current limitations and assumptions may serve as a starting point but there is still significant work needed for the refactoring software to effectively handle large-scale projects. These projects often have software components spread across multiple packages, files, or classes, and the software must accurately identify and manage these elements. Expanding the software's capabilities to handle distributed components and complex project structures will be essential for it to be truly effective in large real-world applications.

4.4 Discussion on evaluation

The evaluation of the refactoring software revealed several important insights, highlighting both its potential and limitations. In terms of **effectiveness**, the benchmarks showed a negative impact on maintainability and understandability, largely due to the increased complexity introduced by additional dependencies and actor-state interactions. These changes, though negative to cohesion and coupling metrics, were expected, as they stem from the transition to a typed actor system. Such trade-offs are a natural consequence of the shift from untyped to typed systems, especially in stateful actors, and while they complicate the structure, they bring the advantage of stronger type safety.

The unit tests exposed some significant limitations and assumptions inherent in the software, particularly regarding its ability to handle edge cases where untyped actors do not seamlessly translate to typed actor systems. Some of these cases required specific solutions that usually depend on the particular use case and interaction patterns. The inability of the refactoring process to account for all these nuanced scenarios emphasizes the need for more case-specific strategies, which are currently lacking in the software's design.

When considering **efficiency**, the changes after refactoring were not uniform across the benchmarks. The results showed variability based on the individual computational task and the specific communication patterns of each benchmark. While some changes in performance metrics did not exceed 30%, there were outliers. These findings suggest that the impact of refactoring on efficiency is contextual and cannot be generalized across different types of tasks or actor communication models.

From the **applicability** perspective, both the unit tests and the evaluation of the open-source "Hydra" project indicate that the refactoring software is still far from being robust enough to handle large projects. While it can identify and assign actor types based on their discovered behaviors, there are notable problems. The software currently struggles when external interactions, distributed project structures, or complex message types are involved, particularly when components involved in actor behavior processing are not correctly recognized. These shortcomings reveal

that the software in its current state is limited to smaller or more straightforward systems where all interactions and structures are easily recognizable.

However, despite these challenges, the underlying **Communication Flow Graph** model successfully captures actor interactions within a system. It recognizes messages and their types and can resolve actor types based on message usage, serving as a proxy for distinguishing actor roles and purposes. This capability is a good starting point for actor system refactoring, documentation, or test generation, and with further refinement, it could form the backbone of a more effective and scalable solution.

That said, while these initial results offer a foundation, the refactoring software requires significant additional development to become usable in large-scale applications. The current limitations, inefficiencies, and unhandled cases point to the need for more comprehensive handling of actor-environment interactions, better recognition of involved components, and more sophisticated resolution of complex communication patterns. In addition, it is important to note that these evaluations were conducted using only 13 benchmarks, with 10 test iterations each, and a single real-world application. Such a sample size is not large enough to make definitive claims about the software's overall performance or generalize the findings. At best, these results provide an initial understanding of the refactoring process and its potential effects, but they should be interpreted cautiously and not taken at face value. Further testing on a larger scale is necessary to draw more reliable conclusions.

Chapter 5

Related Work

In the realm of concurrent programming and actor-based systems, numerous studies have explored the challenges and opportunities associated with system performance, maintenance and code quality. Previous research efforts have delved into various refactoring techniques aimed at improving the structure and performance of actor-based applications. Noteworthy contributions include works that address the intricacies of concurrent system implementations, highlighting the importance of maintaining a balance between complexity and maintainability. Additionally, over the years, a small focus of attention was brought upon the analysis of actor model interactions between components within an actor system, paving the way for enhanced comprehension and optimization of communication protocols within these systems. Despite that, the automated refactoring techniques were never truly explored in an actor model context, but rather partly adapted from classical concurrent techniques that deal with threads, locks, and similar techniques. Below we discuss the more recent proposals of an automated actor system analysis, the effectiveness of more classical refactoring techniques applied to active object languages [16], and the refactoring of a concurrent system code.

5.1 Actor system analysis

Li, Hariri, and Agha's paper "Targeted Test Generation for Actor Systems" addresses the intricate challenge of testing actor-based systems effectively [26]. At the core of their approach is the utilization of static analysis techniques to identify critical paths and potential points of failure within the actor system. By analyzing the structure of the actor hierarchy and the flow of messages between actors, the methodology aims to pinpoint troublesome areas of the codebase through the exploration of backward symbolic execution [11]. Because of the large exploration space due to concurrency, they prune the search space by utilizing two heuristics combined with a feedback-directed technique. This as a result enables the generation of tests that focus on discovered bugs or critical areas, thereby maximizing the effectiveness of the testing process while minimizing redundant test cases.

However, to benefit from the backward symbolic execution, the authors first extract the inter-actor message flow graph that represents potential interactions within the actor system, essentially producing the system's communication protocol. This is achieved by constructing a Message Flow Graph (MFG) that is generated in a multi-stage process. First, the author's implementation transforms the actor application JVM Bytecode into Wala IR [23]. By utilizing built-in hierarchy, call graph, and 1-object-sensitive points-to analysis, they then produce a reference map γ that maps actor references to the actual actors, and a graph G that maps a tuple of source actor reference, target actor reference and an operation type (either message sent or actor created operation) to a list of points-to sets of messages or constructor parameters (depending on the operation type). The graph G is then methodically constructed by observing all the outgoing events (i.e. connections/edges)

from every actor that directly received a message or was created by an external environment, and iteratively move from one observed actor to another analyzing its outgoing events, updating the work list (i.e. unvisited but observed actors) and G by adding a new actor or merging any duplicate actors that contain any new information. After the work list is empty, the reference map γ is collapsed with the graph G to produce the MFG.

The suggested approach can successfully extract the message flow of an actor system that represents true communication patterns with a diminishing number of false positives as the project size increases. However, as pointed out by the authors, the edges in the MFG do not represent acquaintances of actors, it is very well possible that an actor A is aware of an actor B but actor A neither creates nor sends a message to B. In addition, because the MFG is built by exploring actor interactions, it would miss abstract actors or actor classes that are not created or sent a message to. This is especially prevalent in frameworks or libraries that utilize actors as an underlying concurrency mechanism where developers do not interact with the actors [4].

In comparison to the approach outlined in the referenced research, both approaches share the foundational goal of understanding actor-system communication protocols, but our suggested approach effectively identifies all defined actors in the system, even those without direct interactions. However, it does not capture the strength or frequency of relationships between actors, nor does it account for an actor's awareness of others without communication. A notable advantage of the alternative approach is its ability to recognize parent-child actor relationships, which would benefit communication pattern modeling and refactoring accuracy.

"Inferring Ownership Transfer for Efficient Message Passing" by Negara, Karmani, and Agha focuses on optimizing message passing in actor-based systems through static analysis [32]. The paper proposes techniques to infer ownership transfer semantics, where an actor relinquishes ownership of a message to another actor, reducing message passing overhead (i.e. sending a message via *pass-by-value* or *pass-by-reference*). By conducting static analysis to recognize an actor's behavior and exchange pattern when receiving a message, the authors develop algorithms to detect ownership transfer points. The analysis itself is conducted by first constructing a message receiver context-sensitive call graph where message handlers in all actors act as a potential entry point. Then a flow-insensitive field-sensitive Andersen's points-to analysis [41], with the help of the call graph, is conducted to construct an interprocedural points-to graph that helps determine whether data referenced by an object is used by more than one procedure (i.e function). However, that does not answer if and what procedures actually access the data. For that, the authors utilize a custom interprocedural live variable analysis algorithm which will identify what, and in which procedures, messages should be sent via pass-by-value. In this context, a procedure can originate from any actor in the system but only a single actor will be the origin of it. The authors prove that the approach can successfully analyze and model the dataflow of the variables within an actor and the usage of the received data between them, but it does not recognize different possible interactions. In other words, it does not consider actor states and changes caused by the message, nor the relationship between actors.

On one hand the research paper examines actor interactions by analyzing how values are handled during message passing but on another it overlooks actor state and behavior changes, which are crucial for a complete understanding of communication patterns in actor systems. This is understandable since the paper's focus is on message value usage rather than full interaction awareness. Both approaches share methods for identifying messages, relying on accurate call graphs and points-to lists but incorporating variable analysis from their approach could enhance our refactoring, particularly in resolving edge cases involving variable types and values.

When examining actor systems from a broader perspective, semantic similarities become apparent within microservices architecture and their communication patterns. As a result, delving into research on microservices could yield valuable insights. The paper **"ARCHI4MOM: Using Tracing Information to Extract the Architecture of Microservice-Based Systems from**

Message-Oriented Middleware" by Snigdha Singh, Dominik Werle & Anne Koziolek proposes a method for automatic architecture extraction from Message-oriented Middleware (MOM) based microservice applications [40] that may be adapted for actor systems. It resolves the issue where continuous development and evolution of a software project often lead to architecture knowledge loss while the current efforts at architecture reconstruction from dynamic data still do not support the extraction for MOM-based systems [6, 39, 43]. To tackle that, ARCHI4MOM utilizes tracing data collected from the system at run-time, reconstructs the architecture model by extending existing frameworks like OpenTracing and Performance Model Extractor (PMX) to support MOM-based systems, and integrates them into the Palladio Component Model (PCM) [10]. The method workflow consists of data preparation, data processing, architecture extraction, and model construction phases. During data preparation, the system is instrumented with the tracing tool Jaeger to collect run-time data. The data processing phase analyzes trace structure, identifies communication patterns (that is, synchronous or asynchronous), and extracts and integrates into PMX relevant information for architecture reconstruction. Architecture extraction involves identifying additional PCM model elements based on the extracted data to support asynchronous communication and integrating it with previously defined PMX. Lastly, the model builder phase introduces the logic to construct the final PCM architecture model that supports MOM-based asynchronous communication. That is necessary because the PMX lacks the logic to construct a PCM model that supports asynchronous communication

While such an approach can successfully extract and reconstruct architecture from dynamic data within a MOM-based microservices system, it may as well be applicable for the actor model to extract communication protocol in an actor system since in both cases it is a distributed asynchronous communication through a middleware (i.e. actor dispatchers [58]). As such, the same methodology with some effort can be adapted to instrument an actor system, trace the communication data at run-time, recognize the communication patterns and actor interactions, and eventually export and construct the observed communication protocol. Naturally, such an approach would be susceptible to a risk of missing rare communication patterns or interactions if sufficient observation time is not given, and as a result, would be more suitable as a long-running and continuous observation effort.

The main distinction between our and their approaches is the subject of the analysis but the goal remains the same: extract the communication architecture, or pattern, within a system. However, to apply their suggested approach in extracting the 'architecture' of actor system, it would require different tools and techniques to capture the underlying mechanisms and interactions. The whole nature of analysis would transition from static to dynamic as it would require the system to run for the extraction to happen, which is not exactly compatible when you want to refactor the running software. As a result, it may introduces a disconnect between analysis and refactoring phases, potentially requiring manual intervention to bridge the gap.

5.2 Automated software refactoring and refactoring of concurrent and distributed systems

As shown in **"Refactoring and Active Object Languages" by Volker Stolz, Violet Ka I Pun & Rohit Gheyi**, classical Martine Fowler's refactoring techniques, such as *Extract Class*, *Move Method*, etc., have been proven to benefit concurrent systems, yet it is possible for them to inadvertently introduce behavior changes presenting yet another challenge for automated refactoring [42, 36, 16]. As a result, demonstrated by **Baqais, A.A.B. & Alshayeb, M in their literature review "Automatic software refactoring: a systematic literature review"**, attempts at automated refactoring encounter difficulties in ensuring behavior preservation. In addition, the survey highlights that the most common techniques are search-based and the majority of validation approaches utilize metrics, even though balancing competing metrics may become difficult

since improving one metric may degrade another. Despite that, automated refactoring can help to streamline the refactoring process, reduce maintenance effort, and potentially lower the number of bugs in a concurrent system [7, 38].

The paper **"Study and Refactoring of Android Asynchronous Programming" by Yu Lin, Semih Okur & Danny Dig** exemplifies the issues that refactoring can solve in concurrent applications as it helps with transitioning from shared-memory communication into distributed-style communication through refactoring [29]. Specifically, it presents a comprehensive formative study, quantitatively and qualitatively evaluating the asynchronous constructs of Android applications, understanding developer practices and usage of asynchrony, and identifying barriers encountered during this process. To solve several issues identified in the analysis, the paper introduces a refactoring tool *AsyncDroid* that implements an algorithm designed to tackle the challenges of refactoring from shared-memory communication, as exemplified by *AsyncTask*, to distributed-style communication, such as *IntentService* [1, 2].

To correctly apply and recognize the opportunities for refactoring *AsyncTask* to *IntentService*, the authors define four preconditions that act as guidelines to ensure an accurate transformation process. First, all variables that either enter or exit the *doInBackground* method within *AsyncTask* must be capable of being marked as serializable (P1). Secondly, all methods invoked within *AsyncTask* must also be accessible by *IntentService* (P2). Then, the refactored task should directly extend *AsyncTask* and not be subclassed (P3), and lastly, the usage of *AsyncTask* instances should be limited to invoking *AsyncTask.execute* (P4).

The algorithm itself begins by analyzing the objects that can potentially benefit from the refactoring. It identifies the incoming variables (IV) flowing into *doInBackground* and the outgoing variables (OV) escaping from it. Next, the algorithm generates the *IntentService* class, moving the method body of *doInBackground* to *onHandleIntent*. It creates fields for each variable in IV and OV, and copies or moves methods invoked within *doInBackground* to the *IntentService* class. *AsyncTask* instances are replaced with *IntentService* instances, and the receiver for the task result is created and registered, typically in lifecycle event handlers of the top-level or static inner class. Finally, other *AsyncTask* handlers like *onPreExecute* and *onProgressUpdate* are handled accordingly, ensuring that the refactored code retains the original semantics and functionality of the *AsyncTask*.

Authors' and ours approaches to refactoring require specific preconditions and assumptions to ensure correctness, including the recognition of incoming and outgoing variables for the refactoring target. However, their focus is primarily on refactoring the structural elements of classical object-oriented programming by moving code bodies to new locations, such as lifecycle event handlers or static inner classes, based on expected expressions. They prioritize refactoring shared-memory communication into distributed communication without tracking state transitions and data type changes since such refactoring does not necessitate the understanding the class states that can occur during its lifetime, as compared to actors in an actor system. In comparison, our approach deals with more dynamic elements like actor behavior transitions. Nevertheless, the current implementation could benefit from having a broader understanding of all variables influencing actor's behavior and not just the actor reference variables. This would allow, with combination of call graph, to reason about actor's logic and type in finer detail without only relying only on message types that an actor can process. The refactoring itself in both cases rely on identifying syntactical expressions and refactoring them while maintaining equivalent functional purposes.

In a similar fashion, **Y. Zhang, S. Dong, X. Zhang, H. Liu & D. Zhang in "Automated Refactoring for Stampedlock"** address the challenge of selecting suitable refactoring targets, i.e. synchronized locks, and statically analyzing the source code in Java programs to leverage the advanced features offered by StampedLocks [47]. They propose an approach that involves analyzing the JVM Bytecode to locate synchronized methods and blocks, which are potential targets for conversion to StampedLocks. The authors define preconditions for refactoring that must be met

due to the nature of StampedLocks, mainly the conditional operations, such as *wait()*, *notify()* and *notifyAll()*, and reentrance, meaning the ability of a thread to acquire the same lock multiple times. Both of these preconditions are not supported by the StampedLocks [46].

The whole process is divided into three stages: static code analysis, lock inference, and transformation. The first stage analyzes the JVM bytecode to conduct visitor-pattern analysis for the identification of locks, then the discovered locks are checked whether they meet the preconditions, during which the reentrance analysis also identifies the communication operations that cannot be refactored to the StampedLock. In the lock inference stage, the remaining locks undergo a side effect analysis [37] which will model changes and convert them into respective SpampedLock lock models: read lock, optimistic read lock, upgrading lock, or downgrading lock. Lastly, the transformation stage uses a syntax tree to locate all targeted synchronized locks and refactor them into the respective StampedLock with a few additional '*try*...*finally*...' structures.

The research paper's approach shares a common goal with ours: analyzing source code to identify refactoring targets, determining their impact, and applying refactoring based on the most suitable expressions through syntax tree manipulation. Both use similar techniques since they conduct visitor-pattern analysis which could be compared with actor reference resolution for knowing what interacts with class definition under refactoring. However, their focus is on refactoring structural expressions without needing to identify underlying type changes between classes. Instead, they concentrate on how refactoring affects application logic. A key strength of their approach is the use of side-effect analysis, which helps predict the consequences of refactoring and select an appropriate locking model. In essence, this would be akin to different expressions that actor behaviors could take after refactoring which would provide a deeper understanding in available options for refactoring. However, it would be the most beneficial when refactoring ask patterns since they implement '**Futures**' that allow to reason about parallel operations that may be finished in the future - a way to eventually synchronize tasks [62].

Refactoring in current context

Automated refactoring of concurrent applications implies several critical considerations to ensure the correctness and effectiveness of the transformation process. One essential aspect is the analysis of the call graph and application structure, particularly in concurrent programs where dependencies and interactions between threads are parallel and can significantly impact the behavior of the system. Understanding the concurrency patterns and dependencies through a call graph analysis is crucial for identifying synchronization points and potential areas for refactoring. Additionally, establishing preconditions for the refactoring process is essential to guide the transformation and ensure that the refactoring is applied only to suitable code segments which as a result would prevent unintended consequences or errors in the transformed code.

Semantic equivalence between the original and refactored structures is another key consideration, where refactoring techniques aim to preserve the behavior and functionality of the code while improving concurrency constructs. By refactoring semantic-equivalent structures with a set of preconditions, the need for search-based techniques to explore alternative refactorings is minimized, leading to more predictable and reliable transformations. Moreover, the findings from studies such as **"Study and Refactoring of Android Asynchronous Programming"** and **"Automated Refactoring for Stampedlock"** highlight the potential benefits of automated refactoring for asynchronous and concurrent communication that demonstrate how automated refactoring tools can effectively identify and refactor asynchronous and concurrent code patterns.

And as such, There are notable parallels between our proposed refactoring approach and the methods discussed in the reviewed papers, especially when identifying the refactoring targets and evaluating effects of transformation. However, the actor behavior changes over time and the transition form untyped to typed system requires broader system understanding so that actor class behavior change could be modeled and types inferred, which presents itself as a unique challenge. That is not to say that there is nothing that actor refactoring could benefit from existing approaches,

specifically the side-effect analysis would benefit greatly in understanding ask pattern transformation, especially when exploring possible design space. In addition, improved variable recognition influencing actor behavior would allow for finer inference of actor types. By leveraging automated refactoring techniques tailored to the characteristics of concurrent applications, it allows us to streamline the process of optimizing and managing concurrent code which leads to improved code maintainability, readability, and performance.

Chapter 6

Conclusions and Future Work

This chapter begins by summarizing the key contributions presented throughout the thesis, highlighting the approach taken to refactor untyped actor systems into typed actor systems and the methodologies developed for extracting actor communication patterns. Following this, we discuss these contributions within the context of the results observed during the evaluation phase. The chapter examines how effectively the proposed solutions address the challenges of actor system refactoring, as well as any limitations or areas where improvements are needed. Finally, we take a look at possible future research, exploring potential advancements and extensions to this work.

6.1 Contributions and Discussion

One of the primary contributions of this thesis is the definition and implementation of the communication flow graph, alongside an algorithm that demonstrates the process of extracting it. The communication flow graph serves as a comprehensive representation of the interactions between actors within an (untyped) actor system, detailing message exchanges, actor references, and the overall structure of actor communication. This tool helps in visualizing and understanding how messages flow through the system, which is crucial for performing a safe automated refactoring to a typed actor system. This is possible because of the several advantages that the communication flow graph presents. One of which being a precise tracking of message passing and interaction, helping to identify and map out communication patterns between actors. This is particularly useful in automated systems where manual inspection would be too costly or inefficient, such as in large and complex systems. However, the communication flow graph also has limitations. For instance, while it effectively captures asynchronous message passing, it does not fully account for ask-patterns [59], limited comprehension of external environments, and its capability to handle parent-child actor relationships remains underdeveloped. Furthermore, some ambiguities in actor references and certain Scala syntax expressions are not fully resolved by the current extraction process limiting its usability in real-world projects.

In addressing ***RQ1*** *"How to refactor an actor-based system to a typed actor based system?"*, a structured approach is demonstrated for refactoring untyped actor systems into typed ones. The proposed solution leverages the communication flow graph to inform the refactoring process by ensuring that communication patterns remain intact. That is facilitated by answering ***RQ1.1*** *"How can a message flow be extracted from an actor system?"* as the communication flow graph itself is the answer. It presents a clear method for extracting message flow by analyzing actor behaviors, references, and message interactions, which was validated during the evaluation phase. The algorithm's effectiveness in producing a reliable communication flow graph confirms that the message flow can be systematically extracted, even though some limitations persist.

Another key contribution is the automated refactoring process that transforms untyped actor systems into typed ones. The evaluation shows that the refactored systems maintain communication

integrity and performance due to extracted communication flow graph, though they introduce a slight reduction in maintainability. This decrease stems from the addition of a new Scala Object that holds relevant message and actor type definitions. Scala Objects by themselves are defined as classes that will have only a single instantiation of it and is treated as a singleton object [64]. This results in a tighter coupling between an actor and the new object, which slightly impacts the cohesion-coupling balance, making the system less maintainable in the long term. But the performance impact of refactoring varies based on the underlying application. The results from the Savina and stateless benchmarks do not indicate a consistent pattern of performance improvement or degradation. The effect of refactoring on CPU usage, memory consumption, and runtime differs across benchmarks, highlighting that performance change depends on application.

Answering the ***RQ2*** *"How effective is the implemented automated actor refactoring?"*, the evaluation demonstrates that the refactoring process is effective in terms of preserving communication patterns and overall system functionality. However, in answering ***RQ2.1*** *"What effect does refactoring have on the performance of the refactored actor-based systems?"* the efficiency of the refactoring process depends on the underlying application in terms of utilized communication patterns, actor statefulness, and computational tasks that impact performance. The maintainability impact is understood by through ***RQ2.2*** *"How does refactoring influence the maintainability of actor-based systems?"* which is answered by noting that while communication is preserved and performance is generally stable, maintainability suffers due to the increased coupling introduced by the new Scala Objects. Finally, answering ***RQ2.3*** *"Are the implemented refactoring strategies applicable to real-world applications?"* suggests that while the communication flow graph was extracted the refactoring could not be applied effectively to complex real-world actor system. The current refactoring limitations prevented the refactoring process from functioning as intended due to missed interactions, unrecognized message types, and overlooked syntax patterns. While it certainly is capable at refactoring smaller projects within limited scope, the current implementation can not handle applications with large structures and external dependencies.

6.2 Future work

The exploration of refactoring approaches and surrounding issues has revealed several directions of future development and research. This section discusses both the improvements required for the refactoring tool's implementation and the broader research directions that can emerge from this work.

6.2.1 Research improvements

There is significant work required to improve the refactoring tool for larger-scale projects. To make it usable in real-world applications, it is essential to address the current limitations addressed in 3.7.1. Future development efforts should aimed at broadening the scope of actor systems that can be refactored by reducing the number of assumptions and introducing more sophisticated logic that can interpret different code expressions, handle varied actor system patterns, and manage edge cases, especially those emerging from object-oriented programming paradigms.

6.2.2 Follow up research

The communication flow graph and its associated call and inheritance graphs can also serve as valuable tools beyond refactoring. For instance, these graphs can be adapted for test generation, as demonstrated by the previous research [26]. Additionally, the same graphs could be utilized to generate detailed documentation about actor system communication patterns, both at a high level and in terms of the specific functions and behaviors of actor classes. This would provide develop-

ers with a clearer understanding of how actors interact, aiding in debugging, system maintenance, and further development.

In addition, during the evaluation process, it became clear to us that current methods for assessing maintainability, coupling, and cohesion in actor systems are insufficient. Traditional definitions of these metrics are not well-suited to actor-based systems, particularly in the Scala ecosystem, where analysis requires awareness of syntax handling both functional and object-oriented programming definitions. Evaluating coupling and cohesion between actors is especially challenging since there are no robust coupling metrics designed to account for the unique interactions between actors, such as their data dependencies, relationship types, and the intensity of their coupling. Cohesion, though easier to assess, is still problematic because much of an actor's functionality is embedded within the actor framework itself, which limits traditional cohesion evaluations. For example, traditional cohesion metrics might overlook the fact that an actor's behavior change as it processes messages. If an actor has a single receive function that handles incoming messages but calls external singleton objects, traditionally its cohesion score would be low. However, its behavior transitions and changes with each message, suggesting a higher level of cohesion than what traditional metrics would indicate. Similarly, message definitions that reside outside the actor class could reduce cohesion scores even if those messages are exclusively processed by the specific actor type.

These considerations indicate that current maintainability, coupling and cohesion metrics for actor systems need to be reexamined. Traditional metrics should be adjusted to reflect the unique properties and provide new ways to interpret actor systems. Here too the communication flow graph could play a role in facilitating static analysis and improving actor system quality metrics, providing more accurate assessments of maintainability, coupling, and cohesion. These adjusted metrics could help developers better evaluate and optimize actor systems in the future.

6.3 Conclusions

This thesis presents an implementation that focuses on automating the refactoring of untyped actor systems into typed actor systems by introducing a communication flow graph. The implementation demonstrates an approach for analyzing communication patterns in actor-based systems, laying the groundwork for extracting and utilizing message flows and actor interactions. Through this analysis, the communication flow graph was created and used to guide the refactoring process. Although the tool showed success in extracting communication patterns and performing refactoring on smaller-scale projects, it struggled with complex, real-world applications due to the inherent limitations of typed and untyped actor system differences, external interactions, and complex syntax patterns.

The contributions of this research include the development of the communication flow graph and the implementation of an automated refactoring process. While the tool provides benefits in understanding actor communication patterns and improving the refactoring process, limitations still remain. Future work should focus on expanding the tool's capabilities, improving the accuracy of analysis, and refining code quality metrics specifically designed for actor-based systems.

Bibliography

[1] Asynctask, . URL https://developer.android.com/reference/android/os/AsyncTask.

[2] Intentservice, . URL https://developer.android.com/reference/android/app/IntentService.

[3] Callgraph: Generate static call graphs for multiple languages, . URL https://github.com/koknat/callGraph.

[4] Tapir, . URL https://github.com/softwaremill/tapir.

[5] Gul A Agha and Wooyoung Kim. Actors: A unifying model for parallel and distributed computing. *Journal of Systems Architecture*, 45(15):1263–1277, 1999. ISSN 1383-7621. doi: https://doi.org/10.1016/S1383-7621(98)00067-8. URL https://www.sciencedirect.com/science/article/pii/S1383762198000678.

[6] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51. IEEE, 2016.

[7] Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. Actor concurrency bugs: a comprehensive study on symptoms, root causes, api usages, and differences. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–32, 2020.

[8] Abdulrahman Ahmed Bobakr Baqais and Mohammad Alshayeb. Automatic software refactoring: a systematic literature review. *Software Quality Journal*, 28(2):459–502, 2020. ISSN 1573-1367. doi: 10.1007/s11219-019-09477-y. URL https://doi.org/10.1007/s11219-019-09477-y.

[9] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. An empirical validation of cognitive complexity as a measure of source code understandability. *CoRR*, abs/2007.12520, 2020. URL https://arxiv.org/abs/2007.12520.

[10] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[11] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[12] G. Ann Campbell. Cognitive complexity: an overview and evaluation. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, page 57–58, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357135. doi: 10.1145/3194164.3194186. URL https://doi.org/10.1145/3194164.3194186.

[13] Daniel Ciocîrlan. The official repository for the rock the jvm akka essentials course (udemy edition). URL https://github.com/rockthejvm/udemy-akka-essentials/tree/master.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition, 2009. ISBN 978-0-262-03384-8.

[15] Johann Eder, Gerti Kappel, and Michael Schrefl. Coupling and cohesion in object-oriented systems. Technical report, Technical Report, University of Klagenfurt, 1994.

[16] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley Object Technology Serie. Addison-Wesley, 1999.

[17] Adam Freeman. *Namespaces*, pages 337–353. Apress, Berkeley, CA, 2010. ISBN 978-1-4302-3172-1. doi: 10.1007/978-1-4302-3172-1_11. URL https://doi.org/10.1007/978-1-4302-3172-1_11.

[18] Enrico Fregnan, Tobias Baum, Fabio Palomba, and Alberto Bacchelli. A survey on software coupling relations and tools. *Information and Software Technology*, 107:159–178, 2019. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2018.11.008. URL https://www.sciencedirect.com/science/article/pii/S0950584918302441.

[19] Philipp Haller. On the integration of the actor model in mainstream technologies: The scala perspective. In *Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions*, AGERE! 2012, page 1–6, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316309. doi: 10.1145/2414639.2414641. URL https://doi.org/10.1145/2414639.2414641.

[20] Jiansen He, Philip Wadler, and Philip Trinder. Typecasting actors: From akka to takka. In *Proceedings of the Fifth Annual Scala Workshop*, SCALA '14, page 23–33, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328685. doi: 10.1145/2637647.2637651. URL https://doi.org/10.1145/2637647.2637651.

[21] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[22] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 01 1989. ISSN 0010-4620. doi: 10.1093/comjnl/32.2.98. URL https://doi.org/10.1093/comjnl/32.2.98.

[23] IBM. Wala. URL https://github.com/wala/WALA.

[24] Shams M Imam and Vivek Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, pages 67–80, 2014.

[25] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. doi: 10.1109/MC.2006.180.

[26] Sihan Li, Farah Hariri, and Gul Agha. Targeted Test Generation for Actor Systems. In Todd Millstein, editor, *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:31, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-079-8. doi: 10.4230/LIPIcs.ECOOP.2018.8. URL `http://drops.dagstuhl.de/opus/volltexte/2018/9213`.

[27] Xiao-dan Li and Yong-feng Yin. A unified framework for software coupling measurement. In *2014 2nd International Conference on Software Engineering, Knowledge Engineering and Information Engineering (SEKEIE 2014))*, pages 156–161. Atlantis Press, 2014.

[28] Inc. Lightbend. Sculpt: dependency graph extraction for scala. URL `https://github.com/lightbend-labs/scala-sculpt/tree/main`.

[29] Yu Lin, Semih Okur, and Danny Dig. Study and refactoring of android asynchronous programming. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 224–235. IEEE, 2015.

[30] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, Upper Saddle River, NJ, 2002. ISBN 9780135974445.

[31] Michael Nash and Wade Waldron. *Applied Akka patterns: a hands-on guide to designing distributed applications*. " O'Reilly Media, Inc.", 2016.

[32] Stas Negara, Rajesh K Karmani, and Gul Agha. Inferring ownership transfer for efficient message passing. *ACM SIGPLAN Notices*, 46(8):81–90, 2011.

[33] Matheus Paixao, Mark Harman, Yuanyuan Zhang, and Yijun Yu. An empirical study of cohesion and coupling: Balancing optimization and disruption. *IEEE Transactions on Evolutionary Computation*, 22(3):394–414, 2018. doi: 10.1109/TEVC.2017.2691281.

[34] Pluralsight. Hydra. URL `https://github.com/pluralsight/hydra`.

[35] Joanna C. S. Santos and Julian Dolby. Program analysis using wala (tutorial). In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 1819, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.3569449. URL `https://doi.org/10.1145/3540250.3569449`.

[36] Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. Correct refactoring of concurrent java code. In *ECOOP 2010–Object-Oriented Programming: 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings 24*, pages 225–249. Springer, 2010.

[37] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Refactoring java programs for flexible locking. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 71–80, 2011.

[38] Raed Shatnawi and Wei Li. An empirical assessment of refactoring impact on software quality using a hierarchical quality model. *International Journal of Software Engineering and Its Applications*, 5(4):127–149, 2011.

[39] Snigdha Singh, Yves Richard Kirschner, and Anne Koziolek. Towards extraction of message-based communication in mixed-technology architectures for performance model. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, pages 133–138, 2021.

[40] Snigdha Singh, Dominik Werle, and Anne Koziolek. Archi4mom: Using tracing information to extract the architecture of microservice-based systems from message-oriented middleware. In Ilias Gerostathopoulos, Grace Lewis, Thais Batista, and Tomáš Bureš, editors, *Software Architecture*, pages 189–204, Cham, 2022. Springer International Publishing. ISBN 978-3-031-16697-6.

[41] Manu Sridharan and Stephen J Fink. The complexity of andersen's analysis in practice. In *International Static Analysis Symposium*, pages 205–221. Springer, 2009.

[42] Volker Stolz, Violet Ka I. Pun, and Rohit Gheyi. Refactoring and active object languages. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles*, pages 138–158, Cham, 2020. Springer International Publishing. ISBN 978-3-030-61470-6.

[43] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S Bigonha. Recommending refactorings to reverse software architecture erosion. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 335–340. IEEE, 2012.

[44] Vaughn Vernon. *Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka*. Addison-Wesley Professional, 2015. ISBN 9780133846904.

[45] Yanqing Wang, Shengbin Wang, Xiaojie Li, Hang Li, and Jin Du. Identifier naming conventions and software coding standards: A case study in one school of software. In *2010 International Conference on Computational Intelligence and Software Engineering*, pages 1–4, 2010. doi: 10.1109/CISE.2010.5676869.

[46] Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for reentrancy. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, pages 173–182, 2009.

[47] Yang Zhang, Shicheng Dong, Xiangyu Zhang, Huan Liu, and Dongwen Zhang. Automated refactoring for stampedlock. *IEEE Access*, 7:104900–104911, 2019.

[48] Inc. © 2011-2023 Lightbend. Typed actors, . URL `https://doc.akka.io/docs/akka/current/typed/index.html`.

[49] Inc. © 2011-2023 Lightbend. Actor references, paths and addresses, . URL `https://doc.akka.io/docs/akka/current/general/addressing.html`.

[50] Inc. © 2011-2023 Lightbend. Typed actors: Behaviour, . URL `https://doc.akka.io/docs/akka/current/general/actors.html#behavior`.

[51] Inc. © 2011-2023 Lightbend. Learning akka typed from classic, . URL `https://doc.akka.io/docs/akka/current/typed/from-classic.html`.

[52] Inc. © 2011-2023 Lightbend. Classic actors, . URL `https://doc.akka.io/docs/akka/current/actors.html#classic-actors`.

[53] Inc. © 2011-2023 Lightbend. Interaction patterns, . URL `https://doc.akka.io/docs/akka/current/typed/interaction-patterns.html`.

[54] Inc. © 2011-2023 Lightbend. The top-level actors, . URL `https://doc.akka.io/docs/akka/current/general/supervision.html#the-top-level-actors`.

[55] Inc. © 2011-2024 Lightbend. Identifying actors via actor selection, . URL `https://doc.akka.io/docs/akka/current/actors.html#identifying-actors-via-actor-selection`.

[56] Inc. © 2011-2024 Lightbend. Actor props type, . URL https://doc.akka.io/docs/akka/current/actors.html#props.

[57] Inc. © 2011-2024 Lightbend. Actorref object type, . URL https://doc.akka.io/api/akka/current/akka/actor/ActorRef.html.

[58] Inc. © 2011-2024 Lightbend. Typed actors: Dispatchers, . URL https://doc.akka.io/docs/akka/current/typed/dispatchers.html.

[59] Inc. © 2011-2024 Lightbend. Ask pattern, . URL https://doc.akka.io/docs/akka/current/typed/interaction-patterns.html#request-response-with-ask-between-two-actors.

[60] Inc. © 2011-2024 Lightbend. Message adapter, . URL https://doc.akka.io/japi/akka/current/akka/actor/testkit/typed/Effect.MessageAdapter.html.

[61] Inc. © 2011-2024 Lightbend. Style guide, . URL https://doc.akka.io/docs/akka/current/typed/style-guide.html.

[62] Inc. © 2011-2024 Lightbend. Futures and promises, . URL https://docs.scala-lang.org/overviews/core/futures.html.

[63] Inc. © 2011-2024 Lightbend. None object type, . URL https://www.scala-lang.org/api/current/scala/None$.html.

[64] Inc. © 2011-2024 Lightbend. Singleton object, . URL https://docs.scala-lang.org/tour/singleton-objects.html.

Appendix A

Helper Algorithms

This appendix contains most of the helper function and their algorithms used in the algorithms covered in the implementation section 3.

Create Behaviour Object

The **Create Behaviour Object** algorithm is implemented in a function `createBehaviorObject` 21 designed to generate a new function definition from an existing function definition that either returns or is called by a function that returns '**Receive**' type. Its inputs include the syntax tree of the existing function '*node*', keywords '*newMods*' (e.g. '*private*'), parameters '*oldParams*', actor type containing function that will be refactored '*actorSet*', all actor types '*supersets*, and a boolean flag '*isReceive* indicating whether the function returns '**Receive**' type. The algorithm outputs an updated function definition that aligns with the actor's typed using helper functions to transform specific terms.

The algorithm operates by iterating through each term in the original function body, checking for messaging patterns like '`tell`', '`ask`', and `forward`' function calls (including Akka specific syntax sugars '`!`' and '`?`'), and refactoring them. In addition, the each '*sender*' reference is refactored into '*replyTo*' to replace any message sent to a sender of a message with a reference that will be included as a message data after refactoring. Helper functions like '`updateTermCall`', '`updateTermSelect`', and '`updateMessageCallShort`' refactor these terms while maintaining type consistency for actors and messages.

In addition, when dealing with case statements, the algorithm ensures that `Behaviour.same` is included based on '*isReceive*' boolean value, adding proper handling for messages due to the typed system requirement of an actor to always return a behavior after message processing. After processing the function body, it assigns the new actor type to the function and updates parameters and definitions accordingly, ensuring that the actor's type matches the defined communication pattern.

Algorithm 21: Create Behaviour Object

Input: node: Tree, newMods: List of Mods, newName: String, oldParams: List of Param, actorSet: Tuple(List of String, Set of (String, Boolean)), supersets: Map of (List of String → Set of (String, Boolean)), isReceive: Boolean

Output: Function Definition

```
1  Function createBehaviourObject(node, newMods, newName, oldParams, actorSet, supersets,
     isReceive):
2      transformerTermNameMessage ← new Transformer that updates old messages with new
         messages;
3      transformerArgsMessage ← new Transformer that updates message parameters w.r.t. 'self'
         and 'sender';
4      actorTypeName ← createMessageObjectTypeName(actorSet);
5      trnsfBody ← foreach term in node.body do
6          if term expression is using scala syntax sugar for '!' or '?' to send a message then
7              return updateCallMessageShort (term, supersets)
8          else if term is a call to object function and function is "tell" or "ask" or "forward"
             then
9              newTermSelect ← updateTermSelect(term);
10             return updateTermCall(newTermSelect, term, supersets,
                 transformerArgsMessage);
11         else if term is a 'case' statement then
12             return updateTermCase(term)
13         else if term is value or variable definition that uses "sender" then
14             return term.value.replace("sender", "replyTo")
15         else
16             return term
17     transfBodyMessages ← transformerTermNameMessage(trnsfBody);
18     newDeclTpe ← assign new type "Behavior[actorTypeName]" if isReceive is true. Else
         keep the same node.type;
19     newParams ← empty list of parameters;
20     newParamsAndNewBody ← updateDefinitionAndParams(node, newMods, oldParams,
         transfBodyMessages, actorTypeName, isReceive);
21     return Function(newMods, node.name, newParamsAndNewBody.params, newDecltpe,
         Block(newParamsAndNewBody.body))
```

Refactor Message Send Call

The **Message Call Refactoring** algorithm implemented by a function '`updateMessageCallShort`' updates message-sending expressions when syntax sugars '`!`' and '`?`' are used in the actor system. It takes a term representing a message send operation '*term*' and a map of '*supersets*' for message type identification. The algorithm first identifies whether the message was send at any point in the system to a *sender()* expression, indicating that the message must also contain the '*replyTo*' parameter. Then, it is checked if the message target is expressed as a '*sender*' or '*self*' reference, refactoring its arguments and targets references to '*replyTo*' and '*context.self*' accordingly, and returns the updated term.

This helper function is primarily used in the '`createBehaviourObject`' algorithm when message send calls need to be refactored to accommodate typed actor systems. The helper functions like '`identifyCorrespondingMessageType`' and '`transformerArgsMessage`' are used in identi-

fying the correct message type and refactoring message parameters. The algorithm ensures that all send messages, whether targeting '*sender*', '*self*', or other actors, are properly refactored to match the updated actor types.

Algorithm 22: Refactor Message Send Call with Syntax Sugar

Input: term: Term, supersets: Map of (List of String → Set of (String, Boolean))
Output: Term

```
1  Function updateMessageCallShort(term, supersets):
2      newArgs ← empty list of Term;
3      currentMs ← term.message;
4      messageType ← identifyCorrespondingMessageType(currentMs, supersets);
5      sentMessage ← messageType.find(currentMs);
6      if sentMessage boolean value is true then
7          msArgsRefactored ← transformerArgsMessage(currentMs.args);
8          newArgs ← Param("context.self", String) ++ msArgsRefactored;
9      else
10         newArgs ← transformerArgsMessage(currentMs.args);
11     if term.target is "sender" then
12         return Term("replyTo", term.operation, newArgs)
13     else if term.target is "self" then
14         return Term("context.self", term.operation, newArgs)
15     else
16         return Term(term.target, term.operation, newArgs)
```

Refactor the Case Statement

The **Case Statement Refactoring** algorithm implemented in a function '`updateTermCase`' modifies case statements and their body in an actor's message processing function. It takes a case term and outputs an updated case. The algorithm refines the case pattern to ensure the inclusion of the '*replyTo*' parameter within message handling if needed. This ensures compatibility with typed actor systems, where '*sender()*' is replaced by '*replyTo*'.

The body is updated to check for the presence of '`context.become`' expressions and ensure that it transitions to the next behavior properly (since '`context.become`' will always contain reference to function that returns new behavior, it is enough to just replace '`context.become`' with referenced function). If no state transition exists, the body is appended with Behaviors.same, ensuring that each case block concludes by returning a behavior.

Algorithm 23: Refactor the Case Statement and Case Body

```
Input: term: Term
Output: Case
Function updateTermCase(term):
    newPat ← term.pattern;
    if term.pattern contains messages that should contain "replyTo" then
        newPat ← identify message element in term.pattern and return new pattern that contains
          "replyTo" within message parameters
    newTermBlock ← term.body;
    if isReceive then
        contextBecomeExists ← false;
        if term.body contains "context.become" then
            contextBecomeExists ← true;
            term.body ← replace "context.become(...)" in term.body with the parameter that
              was within "context.become(...)"
            newTermBlock ← Block(term.body)
        else
            newTermBlock ← Block(term.body ++ Term("Behaviors.same"))
    return Case(newPat, term.condition, newTermBlock)
```

Refactor the Function Definition and Parameters

The **Function Definition and Parameter** refactoring algorithm implemented by the function '`updateDefinitionAndParams`' is designed to refactor the definition and parameters of a function that returns or is called by a function that returns '**Receive**' type in an actor. It takes as input the original function definition '*node*, a list of keyword modifiers '*newMods*', old parameters '*oldParams*', the transformed function body '*transfBodyMessages*', the actor type '*actorTypeName*', and whether the function handles returns '**Receive**' type '*isReceive*'. The algorithm outputs a tuple consisting of a list of refactored parameters and the function body.

This algorithm refines function definitions by adding necessary parameters like the actor's '*context*' and adjusting '*ActorRef*' parameters using '`updateTermParams`' helper function. If the function returns '**Receive**' type ('*isReceive*' is true), it integrates the `Behavior.receive` expression as a first entry point to the function, as it is crucial for handling message types and having access to actor's context and type definition. For functions that do not directly return '**Receive**' types but are called by other functions that do return '**Receive**', it adds the '*context*' parameter explicitly to resolve references like '*self*'. The algorithm ensures that functions operate correctly within the refactored system by updating both the parameters and function definition to fit Akka's typed actor model.

Algorithm 24: Refactor the Function Definition and Parameters

Input: node: Term, newMods: List[Mod], oldParams: List[Param], transfBodyMessages: Term, actorTypeName: String, isReceive: Boolean

Output: Tuple of (List of Param; Term)

```
1  Function updateDefinitionAndParams(node, newMods, oldParams, transfBodyMessages,
     actorTypeName, isReceive):
2      if isReceive then
3          newBody ← if transfBodyMessages is expressed as partial function definition (i.e.
             block) or a call to some function then
4              if newMods contains "override" then
5                  newParams ← updateTermParams(node.params, actorTypeName)
6                  newDefaults ← updateDefaultConstructorParameters(oldParams)
7                  return (newParams ,Term.Apply("Behavior.receive", newDefaults ++
                     transfBodyMessages))
8              else
9                  newParams ← updateTermParams(oldParams ++ node.params,
                     actorTypeName)
10                 return (newParams, transfBodyMessages)
11         else
12             return (Empty list of Param, transfBodyMessages)
13     else
14         contextParam ← new parameter "context" of a type "ActorContext[$actorTypeName]"
15         newParams ← contextParam ++ updateTermParams (node.params, actorTypeName)
           return (newParams, transfBodyMessages)
```