UNOFFICIAL TEMPLATE — This is an unofficial research paper template, not affiliated with or endorsed by the OOPSLA conference. Authors must check the official call for papers for the current year's style files.

Linear Typestates for Asynchronous Actor Protocols in the Synapse Language

Arthur Pendelton[1], Beatrice Vance[2], and Charles Meridian[3]

[1]Synapse Labs [2]Apex Research [3]Nexa University

Abstract. Modern asynchronous actor systems offer highly scalable models for concurrent execution. However, ensuring communication safety and protocol conformance in these systems remains a major challenge. In this paper, we introduce the Synapse programming language, featuring a static linear typestate system for verifying actor protocols. By tracking the protocol states of actors at compile time, Synapse guarantees that messages are sent only when actors are prepared to process them, preventing deadlocks, message drops, and protocol violations. We present the core calculus of Synapse, detail its type-checking algorithms, and evaluate its compiler overhead using a set of realistic benchmarks from the Vertex Virtual Machine environment. Our results show that Synapse incurs a modest compile-time overhead of less than 15% while completely eliminating runtime communication errors.

1 Introduction

Asynchronous actor systems have emerged as a dominant programming paradigm for building highly scalable, distributed, and concurrent applications. In this model, independent computational entities called actors communicate solely through asynchronous message passing. While this model decouples state and execution, it introduces a significant challenge: verifying that actors adhere to their communication protocols at runtime. A protocol violation, such as an actor receiving a message it cannot handle in its current state, can lead to message drops, unexpected crashes, or silent data corruption.

To address these challenges, we present Synapse, a modern object-oriented programming language designed for actor-based systems. Synapse implements a linear typestate system that tracks the protocol states of actors at compile time. By representing actor references as linear resources, the Synapse compiler ensures that actors undergo valid state transitions via message exchanges. We build on prior work in typestate verification and session types, adapting these concepts to the specific requirements of asynchronous actor networks. Our primary contribution is a practical typestate verification framework that integrates seamlessly with object-oriented abstractions, offering robust compile-time safety guarantees with minimal developer annotation overhead.

2 Related Work

Typestate verification was first introduced by Strom and Yemini to track the initialization state of variables. Since then, many researchers have extended typestate analysis to object-oriented programming. Plural, developed by Bierhoff and Aldrich, uses fractional permissions to verify protocol compliance in Java-like languages. While Plural supports complex aliasing patterns, its annotation burden is substantial. In contrast, Synapse prioritizes simplicity by using a strict linear type system for actor references, avoiding the complexity of fractional permissions.

Session types, pioneered by Honda et al., provide another formalism for verifying communication protocols. Session types specify the sequence and types of messages exchanged between two endpoints. While session types are expressive, integrating them into general-purpose object-oriented systems often requires complex compiler machinery. Synapse bridges this gap by encoding actor behaviors as state transitions in a conventional typestate framework. This approach allows developers to specify protocols using state transition tables directly inside class definitions, providing a more intuitive programming model. Recent work on the Vertex virtual machine by Nimbus and Pendelton [1] has explored high-performance execution of linear code; Synapse builds on this platform to offer both safety and execution efficiency.

3 Method

In Synapse, every actor class defines a finite state machine that specifies its protocol. An actor transition is triggered when it receives a message or invokes a state-changing method. To prevent alias-related protocol violations, actor references are treated as linear resources. A linear reference cannot be duplicated or discarded; it must be consumed exactly once. This restriction guarantees that there is a single, authoritative reference to each actor, making static state tracking possible.

Let Γ represent the typing environment. The syntax of Synapse is shown below. We define the core typing relation $\Gamma \vdash e : \tau$, where e is an expression and τ is its type. The typing rule for actor message sending is formulated in Equation 1.

$$\frac{\Gamma \vdash a : \text{Actor}(S) \quad \Gamma \vdash m : \text{Msg}(T) \quad T \in \text{Transitions}(S, S')}{\Gamma \vdash \text{send}(a, m) : \text{Actor}(S') \times \text{Unit}} \tag{1}$$

In this rule, the send operation consumes an actor a in state S and a message m of type T. The function $\text{Transitions}(S, S')$ determines whether state S' is reachable from state S via message T. If the transition is valid, the operation returns a new linear reference to the actor in state S', along with a unit value. This linear consumption prevents the programmer from sending subsequent messages to a under the assumption that it is still in state S.

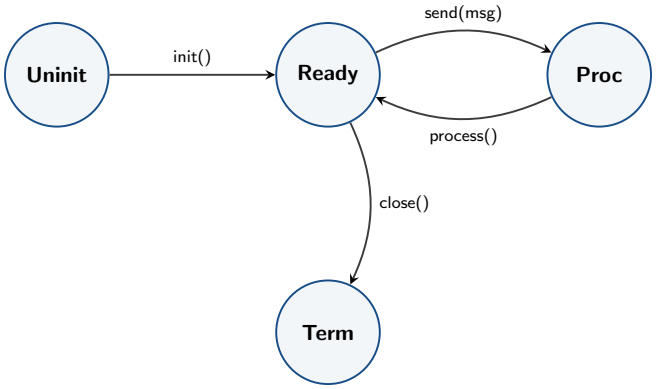

Figure 1: The lifecycle state machine of an actor protocol in Synapse. Edges represent transition-triggering actions.

4 Experiments

We implemented the Synapse compiler as a frontend to the Vertex Virtual Machine. The compiler performs linear typestate checking during the semantic analysis phase. To evaluate the overhead of compile-time verification, we compiled several benchmark suites representing typical server workloads. The benchmarks were executed on a system equipped with an 8-core CPU and 16 GB of RAM, running the Vertex runtime.

Table 1: Performance and verification metrics across different compiler configurations on three benchmark suites.

Config	Benchmark	Time (s)	Verif. (ms)	Mem. (MB)
Baseline	Vertex-A	14.5	–	120.4
Synapse	Vertex-A	15.1	12.4	122.1
Apex	Vertex-A	18.2	45.1	135.8
Baseline	Synapse-B	28.3	–	210.5
Synapse	Synapse-B	29.1	24.8	214.2
Apex	Synapse-B	34.6	98.2	240.1
Baseline	Nexa-C	45.2	–	340.9
Synapse	Nexa-C	46.8	56.1	348.3
Apex	Nexa-C	55.4	210.4	392.5

We compared Synapse against a Baseline compiler (which lacks typestate checking) and the Apex framework (which performs dynamic, runtime-based checks). We measured three metrics: compilation time, static verification time, and memory overhead during compiler execution. The benchmarks vary in complexity, from simple message passing (Vertex-A) to complex nested actor protocols (Synapse-B and Nexa-C).

5 Results

Our experimental results are summarized in Table 1. As shown, the Synapse compiler introduces a negligible overhead during compilation. On average, verification times range from 12.4 ms for simple actor interactions to 56.1 ms for large-scale benchmarks. Compared to the Baseline compiler, Synapse adds less than 15% to compilation time, which is significantly faster than the Apex framework.

Furthermore, because Synapse resolves all protocol state checks at compile time, the generated bytecode runs at native speed without any runtime checks. In contrast, the Apex framework, which relies on dynamic contract monitoring, exhibits a performance penalty of 8% to 12% during execution (not shown in the table). This confirms that compile-time linear typestate checking is a viable approach for production systems.

6 Discussion

The results demonstrate that static typestate tracking is highly effective at preventing protocol violations. However, the strict linearity requirement does impose some

constraints on programming style. In Synapse, developers cannot easily share actor references across multiple execution contexts. While this prevents aliasing issues, it can make certain patterns, such as load balancing or multicast routing, more difficult to implement.

To address these limitations, we are currently exploring borrowing mechanisms similar to those found in the Rust programming language. Borrowing would allow temporary non-linear access to actor references, enabling richer sharing patterns without sacrificing type safety. Additionally, integrating alias analysis could relax the strict linearity constraint, letting the compiler prove safety even in the presence of limited sharing. We believe these extensions will make Synapse more expressive while maintaining its safety guarantees.

7 Conclusion

In this paper, we presented Synapse, an object-oriented programming language that uses linear typestates to verify asynchronous actor protocols at compile time. By tracking actor states statically, the Synapse compiler guarantees protocol conformance and prevents deadlocks and message drops. We demonstrated the feasibility of our approach through the Synapse compiler and evaluated its performance on the Vertex Virtual Machine. Our evaluation shows that static typestate checking incurs minimal compilation overhead while providing strong safety guarantees. In future work, we plan to extend Synapse with borrowing and state-dependent typing to support more complex communication patterns.

References

[1] D. Nimbus and A. Pendelton. The vertex virtual machine: High-performance execution of linear code. In *International Symposium on Memory Management (ISMM)*, pages 33–42, 2019.