Dynamic Composition and Concurrency in I/O Automata: The ioa++ Framework

Sowmith Reddy Thukkani

Eastern Illinois University
sowmithreddy.th@gmail.com

Abstract. This paper presents `ioa++`, a C++ framework that extends the formal I/O automata model to support dynamic composition and concurrency in practical distributed systems. I/O automata provide a rigorous basis for modeling reactive systems with well-defined interfaces and compositional semantics. The ioa++ framework builds on this foundation with runtime support for dynamic system reconfiguration, concurrency-aware scheduling, and non-blocking I/O via POSIX file descriptors.
We demonstrate the utility of ioa++ through a layered implementation of a TCP-based leader election protocol using the asynchronous LCR algorithm. Empirical results show that ioa++ achieves near-ideal speedup (up to 1.95× with two threads) when sufficient independent actions exist, and incurs minimal overhead (average dispatch time of 4316ns in single-threaded and 6067ns in multi-threaded settings).
The framework is open-source, supports simulation and real deployments, and encourages direct reasoning from code due to its close alignment with formal models. Our work bridges the gap between theoretical models and real-world systems, offering a practical tool for building modular and concurrent distributed software.

1 Introduction

Advances in platforms and networks are paving the way for distributed reactive systems of unprecedented sophistication. Cyber-physical systems go beyond the traditional embedded system paradigm by explicitly integrating into the dynamics of the physical system in the computational task. As more devices are equipped with wireless network interfaces, omnipresent computing environments are emerging in homes, offices, and hospitals. The interactions these systems have with the physical environment, user communities, and each other require a degree of interoperability, configurability, adaptability, scalability, robustness, and security not readily supported by existing approaches.

In particular, the activities performed by such reactive distributed systems are intrinsically concurrent and asynchronous. Nodes sense and manipulate the environment and interact with other nodes by sending and receiving messages. To produce more sophisticated systems with current levels of effort, we wish to construct *correct* systems by *composing* modules whose execution is concurrent and asynchronous. The ease with which asynchronous and concurrent modules

can be *developed* and *reused* is thus essential. Modularity, well-defined interfaces, and rigorous composition rules are also crucial for facilitating reasoning about the interactions among system components and allowing a group of interacting components to be understood as a cohesive unit.

Limitations of the state of the art. Existing component frameworks are often implemented using threads, event loops, or a combination of both. We limit our scope to imperative frameworks but note that functional approaches are also gaining popularity [1], [3]. Although threads provide a natural synchronous sequential processing model, the difficulties of using threads in practice are well known [5], [8]. For example, a component that doesn't acquire locks in the same order as others may cause deadlock [4]. Event loops execute handlers in response to external (I/O) and internal events and have been proposed as a simpler alternative to threads [7]. However, that approach lacks common semantics for how events are generated, distributed, and consumed, thus complicating composition. *A key challenge, then, is to provide a component framework for programming asynchronous and concurrent systems that avoids the complications of thread-based development while providing common event semantics and opportunities to exploit concurrent execution.*

Contributions of this work. To address this challenge we present the ioa++ framework for building distributed reactive systems, which is based on the I/O automata formal model. This paper makes three main contributions to the state of the art: (1) a model for *dynamic* systems of I/O automata, which is necessary for implementing (as opposed to only modelling) many distributed systems, (2) a C++ implementation of the model that allows one to build real systems using the facilities provided by POSIX environments and (3) a preliminary investigation into the degree of *exploitable concurrency* in a system as a function of the automata interactions and framework overhead.

In Section 2, we motivate I/O automata as a suitable foundation for components and introduce a model for dynamic systems. Section 3 describes the design and implementation of the ioa++ framework, and Section 4 describes how component automata are programmed within it. In Section 5, we show how ioa++ can be used to build a simple but representative example of distributed software by implementing a leader election protocol for a ring of processes connected by TCP sockets. We evaluate the ability of ioa++ to exploit potential concurrency with a series of micro-benchmarks.

2 I/O Automata Component Model

This section describes how the I/O automata model provides a natural foundation for components, interfaces, concurrency, and composition. Additionally, we introduce extensions supporting dynamic systems where constituent automata change over time.

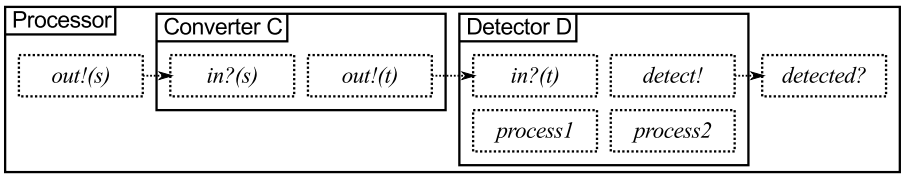

Fig. 1. Example event detection system.

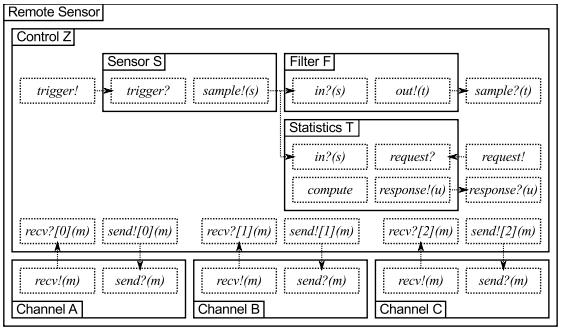

Fig. 2. Example remote sensor system.

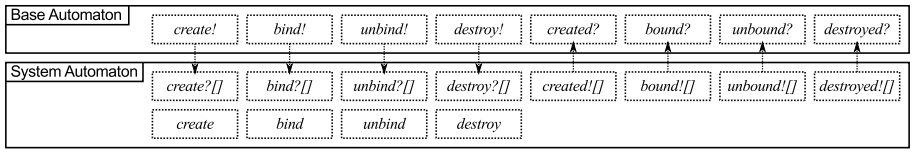

Fig. 3. System automaton and actions.

I/O Automata. The I/O automata model [6] is designed for modeling reactive, concurrent, and asynchronous systems. An I/O automaton consists of state variables and atomic actions, classified as internal, input, or output. Internal actions modify only the automaton's state, output actions modify state while signaling other automata, and input actions react to received signals. Automata are composed by merging state variables, consolidating input effects, and integrating input actions into corresponding output actions. Execution is non-deterministic, where a fair scheduler ensures that local actions are selected infinitely often.

Motivation for I/O Automata-Based Components. I/O automata provide independent states, precise interfaces, and well-defined event semantics, ensuring predictable interactions. Hierarchical decomposition enables a parent automaton to manage multiple children, with composition applied recursively. Illustrates a simple event detector with three automata: a root Processor, and two children, a Converter (C) and a Detector (D). Automata are shown as rectangles, actions as dotted boxes, and bindings as arrows linking outputs.

Explicit Composition. Composition is defined through explicit *bindings* between output and input actions. These must satisfy four constraints: (1) type consistency, (2) an input action is bound to at most one output, (3) an output cannot bind multiple inputs within the same automaton, and (4) an automaton cannot bind its own actions.

Concurrent Execution. Concurrency is achieved when automata execute independently without shared state modifications. Two actions can proceed concurrently if their state variable sets are disjoint. For example, in Figure 1, $out!(s)$ and $process1$ execute concurrently, while $process1$ and $detect!$ cannot.

Action Types and Parameters. Actions may be *valued* (carrying data) or *unvalued* (signals), and *parameterised* (indexing multiple bindings) or *unparameterized*. These distinctions yield ten possible action types. Parameterised input actions, such as $in?[i](s)$, allow multiple outputs to bind to a single input.

Example. Figure 2 presents a remote sensor system comprising automata representing a sensor (S), filter (F), statistics (T), control (Z), and communication channels (A, B, C). The control automaton triggers sampling ($Z.trigger!$), and data propagates through bindings, ensuring synchronization. Statistical computations proceed independently via internal actions, maximizing concurrency.

3 Design and Implementation

The ioa++ framework is a C++ implementation of the component model described in Section 2 for POSIX environments. Figure 4 illustrates its architecture, where the *scheduler* selects and executes local actions, and a *dispatcher* executes them concurrently, ensuring disjoint automata sets. A *controller* enforces a fair

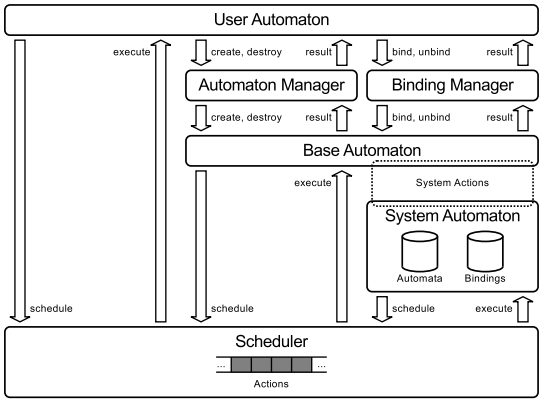

Fig. 4. Framework architecture.

policy for action selection. The framework supports both single-threaded and multi-threaded schedulers for improved performance (Section 6).

User automata inherit from a common *Base Automaton* class, providing system-level communication. The *Automaton Manager* and *Binding Manager* simplify child automaton and binding management, abstracting low-level system actions.

Explicit dynamic scheduling. Automata dynamically declare actions to the scheduler. Executing an action may change an automaton's state, leading to new actions. Scheduling constraints prevent unbounded memory growth.

Binding predicates and scheduling. In dynamic systems, binding predicates affect scheduling. To address situations where an automaton waits for bindings, the framework ensures output actions are scheduled after successful bind/unbind events.

Delayed actions. The framework allows scheduling actions for future execution, enabling timer-like automata. Actions are queued until their designated execution time.

File descriptor events. To maintain non-blocking I/O, file descriptors notify the scheduler when ready. Automata configure descriptors for asynchronous reads/writes, triggering actions upon readiness.

System actions. Automata interact with a system automaton using a request-response model for creating/destroying automata and managing bindings. *Automaton Managers* and *Binding Managers* abstract these interactions, facilitating complex system compositions.

4 Programming Model

The ioa++ framework enables component-based systems using I/O automata. All automata inherit from `ioa::automaton`, implementing the system action interface.

State variables. Automaton state variables are private class members, ensuring independent state encapsulation. Initialization occurs in constructors, which can also bootstrap scheduling.

Actions. Actions include preconditions, effects, and scheduling functions, with traits defining input, output, and internal actions. Type-safe bindings ensure input/output compatibility at compile time.

Listing 1.1. Scheduling an output action in ioa++

```
void my_automaton::schedule_actions() {
    if (can_send()) {
        this->schedule<send_output>();
    }
    if (should_recalculate()) {
        this->schedule<compute_internal>();
    }
}
```

Scheduling. Automata schedule actions using `ioa::schedule`, typically encapsulated in a common scheduling function that checks preconditions before submitting actions.

Dynamic composition. Creating and removing automata/bindings is managed using `ioa::automaton_manager` and `ioa::binding_manager`, which facilitate structured runtime composition.

Binding predicates. Automata inspect bindings using `ioa::binding_count`, ensuring output actions are only triggered when bound.

Managing dynamic constellations. Managing dependencies between system actions ensures correct execution order, particularly in dynamic binding scenarios.

5 Example: The TCP LCR Automaton

In this section, we illustrate how ioa++ can be used to build real distributed systems. In practice, distributed systems are designed using a layered approach where the highest layers contain (often semantically rich) application logic. The lower layers provide services such as TCP/UDP sockets (for communicating between end systems) and timers (for detecting time-outs and performing periodic

tasks). Intermediate layers offer generic coordination mechanisms that serve to abstract away the low-level details from the application logic.

We apply this layered architecture technique to the simple but representative network programming problem of electing a leader in a unidirectional ring of processes connected by TCP sockets. In order to focus on how ioa++ supports distributed applications in general rather than on the details of any particular application, the application logic in this example consists simply of periodically circulating a token around a ring of nodes. If a node has not received a token in a specified amount of time, it initiates a leader election protocol. Once a leader has been elected, the leader injects a token, and the protocol resumes.

Despite the simplicity of the application logic, the resulting *dynamic* distributed system serves to motivate why and illustrate how a number of crucial features are provided by ioa++. For example, nodes can join and leave the ring at any time, and yet the protocol should eventually succeed for complete rings when they stabilize. These dynamics admit fault situations in which a ring might lose its leader or temporarily contain multiple leaders, but each ring should stabilize to having a single leader. Another possible and valid (if potentially rare) situation is a ring circulating a token without a leader. In this situation, the leader sends a token, leaves the ring, and is replaced before the token completes a full cycle. We divide these problems into two sub-problems: leader election and ring formation.

Nodes in a distributed system may be arranged in a ring with one of the nodes as the leader of the ring, e.g., elected per the asynchronous LCR algorithm [6]. The LCR algorithm assumes that every node has a unique identifier (UID). When the protocol begins, each node sends its UID to its successor. When a node receives a UID from its predecessor that is greater than its own, it forwards the UID to its successor. When a node receives its own UID, it elects itself the leader of the ring. We add actions for starting and withdrawing an election, which are necessary to cope with the dynamics discussed previously..

A node in the ring passively waits for a connection from its successor and actively tries to connect to its successor. The tcp_ring_automaton implements this logic and is a composition of an automaton for accepting TCP connections (acceptor), an automaton for making TCP connections (connector), and two automata for TCP connections representing the channel from the predecessor and the channel to the successor. The TCP automata are provided by the ioa++ package and are ordinary automata implemented using file descriptors and the techniques described in Section 3. The tcp_ring_automaton uses timeouts and the techniques listed in Section 4 to manage the dynamic constellation of automata and recover from errors. The tcp_ring_automaton has actions for receiving messages from a predecessor and for sending messages to a successor.

The tcp_lcr_automaton implements the complete protocol by composing an asynch_lcr_automaton and a tcp_ring_automaton. In the resulting layered architecture of the TCP LCR system, the bottom layer consists of component automata for TCP sockets, the middle layer consists of a component automaton that supports coordination via a unidirectional ring, and the top layer imple-

ments the application logic. As this simple but representative example illustrates, the ioa++ framework readily supports the development of architecturally layered component-based distributed systems involving non-trivial dynamics.

In this section, we explore whether ioa++ offers a suitably convenient representation for reifying I/O automata and contains the features necessary for building real distributed systems. Our approach is to (1) express an automaton in ioa++ using an existing listing, (2) simulate a distributed system built using the automaton as a debugging step, and (3) build a real distributed system by composing the automaton with automata for network services. For continuity, we discuss further the asynchronous LCR automaton described in Section 4.

Translating the asynchronous LCR automaton into ioa++ The first version of the asynchronous LCR automaton was translated directly from a listing in [6] but was then refined with (1) an additional action to reinitialize the automaton and (2) modifications that allow an automaton that elected itself the leader to nullify its election. The motivation for these modifications will be explained when we discuss its use in a real system. The listing in [6] consists of approximately 24 lines in a formal language that must be translated to a production language. The raw listing for our final version consists of 121 lines ($\tilde{5}$x). Scheduling accounts for 22 of the 121 lines (20%) and is the main source of code that cannot be traced back to the formal listing. On average, every local action contributed six lines of scheduling code, and every input action contributed three lines of scheduling code.

We observe that having a representation of I/O automata in a production language (such as ioa++) that also closely matches a formal representation is very beneficial. This encourages one to reason about an automaton directly from the source code. For example, an invariant in the asynchronous LCR automaton is that all of the UIDs in the send queue of automaton i must be greater than or equal to the UID of automaton i. This invariant can be quickly checked against the constructor and four actions of the asynch_lcr_automaton. Upon construction, the send queue contains the UID of the automaton so the invariant holds initially. The leader action does not modify the send queue so the invariant holds trivially. The init action adds the automaton's UID to the send queue. The receive action only adds a UID to the send queue if the UID is greater than the automaton's UID. The send action removes a UID from the send queue. Reasoning about an automaton directly from the source code allows one to use formal methods to develop and debug systems *at the point of implementation directly.*

Unit testing with simulation After the asynchronous LCR automaton was implemented, we developed a simulation to test it. File in the ioa++ package contains the driver for the simulation. The simulation creates a unidirectional ring of a specified size consisting of alternating asynch_lcr_automaton and channel_automaton. The channel_automaton was also a direct translation from [6].

The simulation showed that the initial asynch_lcr_automaton was not correct. The first bug we encountered was a silent failure where the ring failed to elect

a leader. The problem was that the leader action was never scheduled, so it was added to the scheduling function. The second bug manifested itself by electing the wrong leader, i.e., one whose UID was not the maximum. A closer inspection showed that certain UIDs were being sent but never received. The key insight into solving the problem was understanding that the automata were executing as the ring was being constructed; that is, we had originally failed to consider dynamic composition. The solution was to add binding predicates to the outputs of the asynch_lcr_automaton and channel_automaton.

From abstraction to real system The next phase involved using the asynchronous LCR automaton to elect the leader of a real unidirectional ring of processes connected by TCP sockets. The leader of the ring periodically sends a token, which is forwarded by all nodes in the ring. If a node has not received a token for a specified duration of time, it calls for an election. This is the motivation behind the action that resets the asynch_lcr_automaton. Nodes listen for connections from their predecessor and attempt to connect to their successor. Nodes can join and leave the ring at any time. The protocol should eventually succeed for complete rings and restart when a ring becomes incomplete. These dynamics admit fault situations where a ring might lose its leader or contain multiple leaders. A ring containing multiple leaders should stabilise to have a single leader. This is the motivation behind the asynch_lcr_automaton being able to withdraw its election. A rare (but possible and valid) situation is a ring circulating a token without a leader. In this situation, the leader sends a token, leaves the ring, and is replaced before the token completes a full cycle. This situation requires no action as a token is circulating in the ring.

The TCP automata provided by the ioa++ package are ordinary automata implemented using file descriptors and the techniques described in Section 3. The tcp_lcr_automaton uses the techniques for managing dynamic constellations described in Section 4.

The tcp_lcr_automaton case study confirms that I/O automata can be a good foundation for building reusable concurrent modules. The key idea is that all of these modules have well-defined interactions when composed while being inherently concurrent. For example, one could substitute the asynch_lcr_automaton with a different automaton to realize a different leader election protocol or a different protocol altogether. Alternatively, the tcp_lcr_automaton could be refactored to use a different communication mechanism, e.g., UDP. When composing an automaton with an existing system, one need only focus on the protocol implemented by the automaton rather than worrying about interacting threads and race conditions.

6 Evaluation

The ioa++ framework permits independent actions to be executed concurrently. The goal of this evaluation is to show that the degree of exploitable concurrency depends only on (1) the interactions of the I/O automata comprising a system

and (2) the overhead of the framework, which this evaluation serves to quantify for our current implementation, of ioa++.

To evaluate concurrent execution, we examine a system whose actions can be configured to span the range from having no independent actions to having only independent actions. We measure the time required to execute a fixed number of actions using a scheduler with a configurable number of threads. We then calculate the speed-up by comparing an execution using one thread to an equivalent execution using two threads. We also vary the complexity of each action to gain insight into how overhead and synchronization affect concurrent execution.

An automaton of type R contains an input, output, and internal action. The output and internal action effects execute an algorithm whose complexity is proportional to the parameter N. The automaton executes a fixed number of local actions. The automaton schedules the internal action with probability $\sqrt{\rho}$ and schedules the output action otherwise. The system S to be executed consists of two R automata composed so the output action of one is composed with the input action of the other. If we divide the execution into rounds where each automaton executes a single action, the probability that both R automata execute an internal action is $\sqrt{\rho} \times \sqrt{\rho} = \rho$. Thus, we can use the parameter ρ to vary the independence of the automata.

For this experiment, we used a simple multi-threaded scheduler that assigns each local action to a thread based on the automaton identifier of the local action using the techniques described in Section 3. When the scheduler is configured to use a single thread, all actions are executed by that same thread. When two threads are used, one thread executes the actions of one R automaton while the other thread executes the actions of the other R automaton.

A trial consists of a choice for ρ, N, and the number of actions to be executed. For our experiments, the parameter ρ was varied from 0.0 to 1.0 in increments of 0.1, and the parameter N was varied from 1 to 1000000 by factors of 10. The number of actions executed by each automaton was fixed at 1000. Each trial was repeated 1000 times. All calculations were performed assuming a normal distribution at a confidence level of 95%.

The driver for a single trial can be found in examples/random.cpp of the ioa++ package. The trials were performed on a Mac Pro running OS X 10.5.8, which has two 2.66 GHz Dual-Core Intel Xeon processors and 2GB of memory. The code was compiled and linked with i686-apple-darwin9-g++-4.0.1 build 5493, which includes -O2 optimisation and the -DPROFILE flag to enable profiling in ioa++.

Figure 5 shows the speed-up when system S was executed with two threads versus one thread. Confidence intervals are shown but are negligible, with the largest being a speed-up range of ± 0.0165. When $\rho = 0$, every action is a bound output action and, therefore, depends on both R automata. Consequently, every action must be serialized, yielding a maximum speed-up of 1. When $\rho = 1$, every action is an internal action and can be executed concurrently with a corresponding maximum speed-up of 2. The speed-up shows a strong dependence on the duration of each local action, which is proportional to N. A small value

for N means that very little time is spent executing automaton code. Consequently, a greater fraction of time is spent executing framework code, which includes various synchronization calls to the threads library. For a small enough N, this overhead dominates the execution time. Multiple threads exacerbate this situation since they actively interfere with one another, as can be seen in the slowdown for small values of N. Conversely, when N is large, relatively little time is spent executing the framework and synchronization code. Consequently, contention is reduced, allowing for greater speed-ups as indicated by Figure 5.

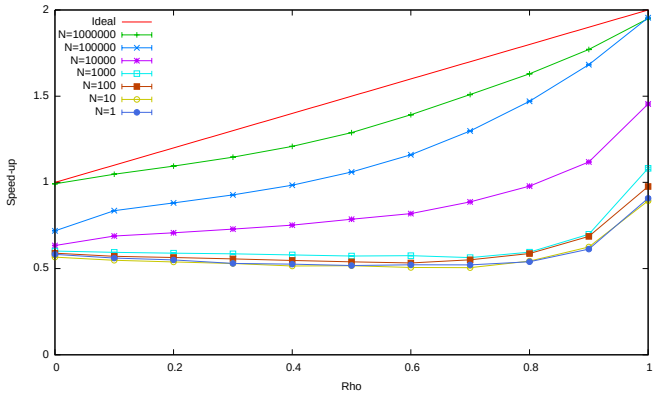

Fig. 5. Speed-up for system S.

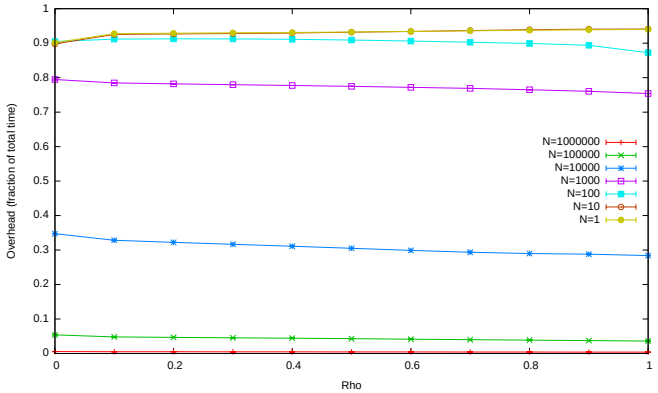

Fig. 6. Overhead for system S when using one thread.

Figures 6 and 7 show the average per thread of the fraction of time devoted to framework code and synchronization calls for the single and multi-

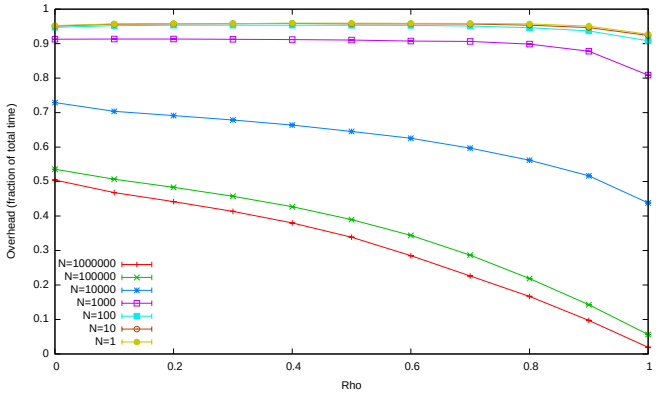

Fig. 7. Overhead for system S when using two threads.

threaded executions; confidence intervals are again negligible, with the largest being ± 0.002155 for Figure 6 and ± 0.001781 for Figure 7. Since the number of actions to be executed is constant, the single-threaded execution shows a relatively consistent fraction for each value of N. The fraction decreases as ρ goes from 0 to 1 because the system transitions from always acquiring two locks to always acquiring one lock. The multi-threaded execution shows that the system overhead decreases as ρ goes to 1 and is more pronounced when N is large. For example, when $N = 1000000$, one thread spends half of its time waiting on the other thread when $\rho = 0$ and spends very little time on synchronization when $\rho = 1$.

This experiment indicates that although concurrent execution is possible with ioa++, a significant speed-up will only be achieved if (1) enough independent actions are enabled and (2) the duration of the independent actions is large relative to the overhead of ioa++. The overhead of ioa++ can be divided into three components corresponding to the time it takes to dispatch an action, the time required to synchronize using the threads library, and the time required to add an action to the scheduler via the ioa::schedule call. From the results, we calculated the average overhead per action of the ioa++ framework when dispatching an action and found it to be 4316ns $\pm$ 3.5ns for the single-threaded experiments and 6067ns $\pm$ 3.6ns for the multi-threaded experiments. Similarly, we calculated the scheduling overhead and found it to be 247ns $\pm$ 0.678ns for the single-threaded experiments and 307ns $\pm$ 0.357ns for the multi-threaded experiments. Both of these potentially can be improved with better scheduler design. Synchronization time depends on the interactions among the automata comprising the system, the scheduler, and the locking scheme used by the framework. We plan to optimize scheduling and synchronization in ioa++ in future work.

7 Related Work

Traditional concurrency models include threads, Communicating Sequential Processes (CSP), and Actors, each with varying support for modularity and safety. More recent approaches include reactive programming frameworks like RxJava and Reactivex, which emphasise event-driven design but lack a strong formal foundation for reasoning about system-level correctness.

Our work differs in that it provides a practical runtime (ioa++) rooted in the I/O automata formalism, enabling correctness reasoning directly on executable systems. Spectrum and IOA enable execution and simulation of automata but focus on algorithm design and static composition. By contrast, ioa++ emphasises runtime dynamics, compositional layering, and direct implementation of POSIX-based real systems.

Akka (for the JVM) and Erlang are notable for concurrency and fault-tolerance but offer limited formal verification compared to I/O automata. ioa++ combines low-level POSIX capabilities with formal semantics, offering a unique trade-off between rigour and realism.

8 Future Work

We plan to extend the IOA++ framework in several directions. First, we aim to develop visualisation tools to help debug and trace automaton interactions, state transitions, and scheduling decisions. Second, integrating ioa++ with formal specification and verification tools like TLA+ or Alloy could allow developers to check system invariants during early design phases.

Third, we plan to explore distributed deployments of ioa++ across multiple hosts, enabling fault-tolerant and scalable execution. Additionally, we envision applying ioa++ in larger, industry-relevant use cases, such as orchestrating microservices or managing Internet of Things (Iot) ecosystems. These efforts will help us assess and optimise performance, fault recovery, and developer usability at scale.

9 Conclusions

I/O automata are a good basis for asynchronous and concurrent components because they support independent states, well-defined interfaces, and well-defined interactions under composition. The ioa++ framework facilitates concurrent execution via dynamic composition, and the degree of concurrency is limited only by the interactions of the automated and the overhead of the framework. Two key challenges when moving from the formal model to an actual implementation were introducing features for managing dynamic constellations of automated and providing a concrete scheduling mechanism. We agree with [2] that having a model that can be compiled is beneficial for reasoning about the program directly from the source code and helps to close the gap between the developer's mental model and the code listing. Our primary goal moving forward is to use

the ioa++ framework to gain further experience building real systems with I/O automata and to support the optimisation of their performance.

References

1. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent programming in ERLANG. Prentice Hall (1996)
2. Georgiou, C., Lynch, N., Mavrommatis, P., Tauber, J.: Automated implementation of complex distributed algorithms specified in the ioa language. International Journal on Software Tools for Technology Transfer (STTT) **11**(2), 153–171 (2009)
3. Halloway, S.: Programming Clojure. Pragmatic Bookshelf (2009)
4. Havender, J.: Avoiding deadlock in multitasking systems. IBM systems journal **7**(2), 74–84 (1968)
5. Lee, E.: The problem with threads. Computer **39**(5), 33–42 (2006)
6. Lynch, N.: Distributed algorithms. Morgan Kaufmann (1996)
7. Ousterhout, J.: Why threads are a bad idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference (1996)
8. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal **30**(3), 202–210 (2005)