MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering
and Computer Science
Cambridge, Massachusetts 02139

Proposal for Thesis Research in Partial Fulfillment
of the Requirements for the Degree of Doctor of Philosophy

Title: Verifiable Code Generation from Abstract I/O Automata Models for Distributed Computing[1]

Submitted by: Joshua A. Tauber
NE43-369
Cambridge, MA 02139

(Signature of author)

Date of submission: March 21, 2001

Expected Date of Completion: May 2002

Laboratory where thesis will be done: Laboratory for Computer Science

Brief Statement of the Problem:

Reasoning about and building distributed systems is notoriously difficult. I/O automata provide a simple mathematical basis for formally modeling and understanding distributed systems. Using a rich set of proof techniques, I/O automata have been used to verify a wide variety of distributed systems and algorithms and to express and prove several impossibility results. IOA is a formal language for describing I/O automata that has been introduced to promote I/O automata-based techniques and to support an integrated software development environment for distributed systems. This environment, the IOA toolset, will support algorithm design, development, testing, and formal verification using automated tools. The toolset connects I/O automata together with both lightweight (syntax checkers, simulators, model checkers) and heavyweight (theorem provers) tools.

In this thesis, we propose to develop and evaluate an IOA-to-Java compiler for distributed algorithms. We will enable programmers to write their specifications at a high level of abstraction, validate the specification using other tools in the IOA toolset, refine the specification to a low-level design, and then automatically translate the design into code that runs on a collection of workstations communicating via standard networking protocols. A major contribution of this work is that the compiler will overcome the existing disconnect between formal specification and system implementation. We will prove that the compilation method preserves safety properties proved about the IOA code in the generated Java code. The proposed work has three major components. We will design an implementation of the compiler. We will model and prove the correctness of the compiler. Finally, we will experimentally evaluate the compilation environment.

[1]This research supported in part by NSF grants ACI-9876931, CCR-9909114, 1784-18454LD, and CCR-9804665; AFOSR grants SA2796PO 1-0000243658, F49620-00-1-0097, F49620-97-1-0337; NTT grant MIT9904-12; and DARPA grant F19628-95-C-0118.

1 Overview

Reasoning about and building distributed systems is notoriously difficult. I/O automata provide a simple mathematical basis for formally modeling and understanding distributed systems. Using a rich set of proof techniques, I/O automata have been used to verify a wide variety of distributed systems and algorithms and to express and prove several impossibility results. IOA is a formal language for describing I/O automata that has been introduced to promote I/O automata-based techniques and to support an integrated software development environment for distributed systems. This environment, the IOA toolset, will support algorithm design, development, testing, and formal verification using automated tools. The toolset connects I/O automata together with both lightweight (syntax checkers, simulators, model checkers) and heavyweight (theorem provers) tools.

In this thesis, we propose to develop and evaluate an IOA-to-Java compiler for distributed algorithms. We will enable programmers to write their specifications at a high level of abstraction, validate the specification using other tools in the IOA toolset, refine the specification to a low-level design, and then automatically translate the design into code that runs on a collection of workstations communicating via standard networking protocols. A major contribution of this work is that the compiler will overcome the existing disconnect between formal specification and system implementation. We will prove that the compilation method preserves safety properties proved about the IOA code in the generated Java code. The proposed work has three major components. We will design an implementation of the compiler. We will model and prove the correctness of the compiler. Finally, we will experimentally evaluate the compilation environment.

In the remainder of this proposal, we will explore the background and motivation, clearly state the problem and its importance, lay out our approach, propose specific work to be done, and list the key milestones to be achieved.

2 Background and Motivation

Distributed systems are increasingly prevalent and complicated. Complex and interconnected distributed services are being rolled out daily in areas as diverse as view-oriented group communication, mobile IP routing, computer-supported cooperative work (CSCW), and transportation control systems. Many such systems are useful tools for computer scientists as well as objects of study. Such systems have a variety of correctness, performance, and fault-tolerance requirements. Unfortunately, the concurrent nature of these systems intertwined with their scale and complexity make building and reasoning about them notoriously difficult.

Formal methods and modeling can greatly aid understanding, analyzing, and designing distributed systems. In this mathematical and state-machine-based approach, one describes systems in a *structured* way, viewing them (orthogonally) as *parallel compositions* of interacting components, and as sequences of descriptions at different *levels of abstraction*. These structured descriptions make the systems easier to understand and serve as the basis for documentation and validation. In turn, this understanding make it easier to build complex systems correctly. Furthermore, it becomes tractable to understand the consequences of extensions of complex systems.

2.1 I/O Automata

Lynch *et al.* have developed a mathematical technology for designing, analyzing, and describing distributed computing systems [LT87, LT89]. I/O automata provide a simple mathematical basis for understanding distributed systems. I/O automata model the behavior of systems of interacting components. Complex systems are decomposed into simpler pieces whose structure can be understood using levels of abstraction and, orthogonally, parallel composition. I/O automata are mathematical entities that provide a sound basis for understanding and reasoning about distributed systems.

An *I/O automaton* is a labeled state transition system. It consists of a (possibly infinite) set of *states* (including a nonempty subset of *start states*); a set of *actions* (classified as *input*, *output*, or *internal*); a set of *transitions*, which are (state, action, state) triples specifying the effects of the automaton's actions; and a set of *tasks*, which are sets of non-input actions. Input actions are required to be *enabled* in all states. The operation of an I/O automaton is described by its *executions*, which are alternating sequences of states and actions. The externally-visible behavior occurring in executions constitutes its *traces* (sequences of input and output actions).

One automaton is said to *implement* another if all of its traces are also traces of the other automaton. I/O automata admit a *parallel composition* operator, which allows an output action of one automaton to be performed together with input actions in other automata; this operator respects the trace semantics. The result of applying the composition operator to a collection of compatible automata is a new automaton semantically equivalent to the original collection.

2.1.1 Proof Techniques

The I/O automaton model supports a rich set of proof techniques. *Invariant assertion* techniques are used to prove that properties of automata are true in all reachable states. (These date back at least to Owicki and Gries [OG76].) Pairs of automata can be related using various forms of *simulation relations* that preserve external behavior. In particular, a simulation relation defines a mapping between the states of two automata that is maintained in all reachable states, thereby showing one automaton implements the other. To prove a simulation relation, one demonstrates a *step correspondence* between the two automata, that is, one shows that for every step (state transition) of the implementation automaton there is an equivalent (sequence of) steps that the specification automaton can take that will maintain the simulation relation. (Forward and backward simulation methods are discussed in [Lam83, LV95, dRE98].) A succinct explanation of the model and many of its proof techniques appears in Chapter 8 of [Lyn96].

2.1.2 Using I/O automata

To use I/O automata, one typically begins by describing a distributed system as a global, high-level application or service. If possible, use a single, centralized I/O automaton to capture the most general description of the externally discernible behavior of the system. For example, a bank can be described as a set of accounts with owners and balances into which deposits can be made and from which withdrawals can be taken. To be as general as possible, use nondeterministic choices whenever possible. Then describe key properties of the system using invariants. For example, one might assert that no balance may ever be negative and that the total amount of funds in the bank is never changed by a transfer between accounts.

A process of *successive refinement* then follows to describe the system as made up of lower-level services. These lower-level automata may be simpler to understand (individually), a more realistic depiction of a real system, or a model of an existing system we wish to use. Two orthogonal methods of refinement apply. First, *levels of abstraction* may be used to define interfaces between low-level services and high-level applications. So, a bank may depend on a secure, low-level, wire transfer service to move money around. A distributed bank branches application can be described on top of that wire transfer service. Second, one can apply *parallel decomposition* to describe a service or application as a collection of components. Thus, the wire transfer service may be made up of a collection of computing nodes and communication networks. In either case, the result is a set of lower-level I/O automata that are intended to implement the previously specified high-level service.

Having refined the high-level, global system specification into a low-level distributed system description, one can prove that the low-level description implements the high-level specification. Doing so shows that all behaviors of the low-level system could be interpreted as valid behaviors of the specification. So for example, no combination of accepted deposits and withdrawals at any set of branches can ever cause a balance to become negative. To perform this proof, one applies the composition operator to the various pieces of the low-level, distributed system specification. One then demonstrates a simulation relation between the resulting automaton and the high-level, global system specification.

2.2 Successes with the I/O Automata Model

I/O automata have been used to successfully model and verify a wide variety of distributed systems and algorithms and to express and prove several impossibility results. Examples include [Lyn96, LMWF94, Her91, BGLR01, CR96, FKL98, FLS97a, CHD98, Smi96, Smi97]. The model was developed for reasoning about theoretical distributed algorithms but has since been applied to many practical services. For example, I/O automata have been applied to distributed shared memory [FKL98, FGL+96, FGL+99], group communication [FLS97a, FLS97b, CHD98, DFLS98, HLvR99], and standard networking [Smi97, Smi96, Smi98]. The resulting expositions and proofs have resulted from a structured, rigorous approach that has resolved ambiguities and uncovered errors. Logical errors have been found in algorithms underlying Orca [BKT92] and Ensemble [HvR96, Hay96] while unexpected behavior was found in T/TCP [Bra94].

2.3 Related Models

These experiences applying the model highlight several key features useful for this style of work. The external behavior of automata are based on simple linear traces. Composition is based on synchronized external actions. Levels of abstraction are easily described with successive levels related by inclusion of trace sets. However, the I/O automaton model is just one of a variety of mathematical models developed for specifying and verifying distributed systems. A number of models describe systems as automata performing transitions with preconditions and effects (*i.e.* guarded commands). The effects can be described operationally (by an imperative program) or axiomatically (by predicates that relate pre-states to post-states). I/O automata can be described by either or both methods as needed.

Lamport's TLA [Lam94b] describes the effects of transitions using the latter assertional style. TLA describes automata as constraints on the universe of all possible outcomes and composition of automata simply as the conjunction of such constraints. Thus, there is no clear analogy to traces as descriptions of external behavior. While Chandy and Misra's UNITY language [CM88] and Manna and Pnueli's language SPL [MP95] are operational in style, their automata combine via shared variables rather than shared actions. Thus, neither model uses sets of traces as its notion of external behavior. All three of these languages have been used to prove both safety and liveness properties using temporal logic.

Process algebras also use automata models. CSP [Hoa85] and CCS [Mil89] compose by synchronizing external actions. However, the process algebra proof style is quite different from that used with I/O automata. They build concurrent systems from single-step processes using algebraic expressions and operators. Proofs consist of the application of a rich set of inference rules to algebraic expressions denoting processes and tend to emphasize the equivalence of expressions.

A variety of other methods have been used for formally specifying distributed systems. For example, see work by Harel [Har87], Meseguer [Mes92], and Ostroff [Ost97]. Estelle [Int89] exemplifies "Formal Description Techniques (FDT)." FDTs are high-level, highly-expressive programming languages with formal semantics. Estelle's semantics are based on a guarded command style automaton model [Dem97]. While the semantics are carefully defined, proofs are not generally done with these systems, possibly due to the complications of the expressive semantics.

2.4 Tool-based work

Many of the modeling and verification techniques based on state machines are highly stylized. The properties researchers choose to identify and the structure of the proofs they use to verify these properties follow simple, common patterns. This success in combining simple, formal models with simple, recurring patterns of proofs has led a variety of researchers to explore using automated tools to work with these models. These computerized assistants have relieved researchers of some of the burden of rote and repetitive tasks associated with these models and the associated proofs. These tools range from verification-oriented tools such as automated proof assistants and model checkers to more design-oriented tools such as simulators and even compilers. They have been used to increase the level of detail and reusability in the produced proofs, to find errors in existing designs, to explore design spaces, and to connect specifications with produced code. However, no system has yet combined all these techniques for a single model.

A number of tools have been based on the CSP model [Hoa85]. The semantics of the Occam parallel computation language is defined in CSP [OCC84, OCC88]. While there are a number of Occam compilers that target the Transputer architecture we have found no evidence of verification tools for Occam programs.

Formal Systems, Ltd., has developed a machine-readable language for CSP that is accepted by a number of tools. The FDR model checker allows the checking of a wide range of general safety and liveness properties of CSP models [RGG+95]. The ProBE tool enables the user to "browse" a CSP process by following events that lead from one state of the process to another. The user controls the resolution of non-determinism and the choice of actions.

Cleaveland *et al.* have developed a series of tools based on the CCS process algebra [Mil89]. The Concurrency Workbench [CPS93] and its successor the Concurrency Factory [CGL+94] are toolkits for the analysis of finite-state concurrent systems specified as CCS expressions. They include support for verification, simulation, and compilation. A model checking tool supports verifying bisimulations. A compilation tool translates specifications into Facile code.

Lamport developed TLA+[Lam99] as a formal language for describing TLA automata in a modular fashion. With Engberg and Grønning, Lamport developed the TLP [EGL92] theorem prover based on the Larch Prover. Yu and Manolios collaborated with Lamport to develop the TLC model checker [YML99] for TLA automata specified in a subset of TLA+. TLC has been used to find errors in the cache coherence protocol for a Compaq multiprocessor. Lamport has no desire to generate code from TLA+. Kalvala has formulated TLA for Isabelle [Kal95].

Bjørner *et al.* have developed the Stanford Prover (STeP) for verifying SPL automata [BMSU98, BBC+99]. STeP combines model checking with deductive methods to allow the verification of a broad class of systems, including programs with infinite data domains. We do not know of any STeP work on code generation.

2.5 I/O automata-based tools

2.5.1 Theorem Provers

A number of researchers have used automated theorem proving assistants such as the Larch Prover [GG91], PVS [SOR93], or Isabelle [Pau93] to revisit proofs of algorithms previously described in the distributed systems literature.

At MIT, researchers have encoded I/O automata theory in the Larch Shared Language (LSL) for use by the Larch Prover (LP). Søgaard-Andersen *et al.* developed the method and applied it to connection management protocol examples [SAGG+93]. Söylemez used Larch to verify the timing properties of MMT automata [Söy94]. Luchangco *et al.* extended the method to timed I/O automata and applied it to mutual exclusion and leader election algorithms [LSGL95, Luc95]. Probably the most complicated LSL-based proof done in this style was accomplished by Petrov *et al.* in verifying the Bounded Concurrent Timestamp algorithm of Dolev and Shavit [PPG+96]. In all these proofs, the researchers specified the algorithms of interest as I/O automata. The researchers then encoded each automaton and the relevant invariants and simulation relations in LSL for input to LP.

Müller, Nipkow, and Slind have encoded I/O automata theory using temporal logic for use with the Isabelle theorem prover [NS95, MN97, Mül98]. Müller has used the system to derive and verify many theorems about I/O automata including standard proof methods. Müller has connected the system to two model checkers, translating I/O automata embedded in Isabelle into inputs for the the STeP and μcke model checkers [BBC+96, Bie97a, Bie97b]. He uses abstractions from finite to infinite automata to use model checkers to prove general properties. The model checker is used to verify the properties of the finite state automata while the theorem prover is used to verify the abstraction. Each automaton must be hand encoded directly in the Isabelle logic.

Vaandrager *et al.* have encoded I/O automata theory for the PVS theorem prover [HSV94, DGRV00]. As with previous efforts, the authors hand encoded individual automata in the input logic of the prover.

Archer and Heitmeyer also encoded I/O automata theory for PVS [AHR00, RAH00]. However, they used PVS's support for user-defined strategies to create a special-purpose interface to PVS. The tool TAME (Timed Automata Modeling Environment) provides an interface to the prover PVS to simplify specifying and proving properties of automata models (including both timed and untimed I/O automata). TAME aims to allow a software developer who has basic knowledge of standard logic, and can do hand proofs, to use PVS to represent and to prove properties about an automaton model without first becoming a PVS expert. In addition TAME produces mechanical proof transcripts that largely correspond to the structure of Lamport-style hand proofs [Lam93, Lam94a].

2.5.2 Simulators and Code Generators

Goldman's Spectrum System introduced a formally-defined, purely operational programming language for describing I/O automata [Gol90a, Gol91]. He was able to execute this language in a single machine simulator. He did not connect the language to any other tools. However, he suggested a strategy for distributed simulation using expensive global synchronizations. More recently, Goldman's Programmers' Playground also uses a language with formal semantics expressed in terms of I/O automata [GSM+95].

Cheiner and Shvartsman experimented with methods for generating code from I/O automaton descriptions [Che97, CS98]. They selected a particular distributed algorithm from the literature (the Eventually Serializable Data Service of Luchangco *et al.* [FGL+96]) and generated by hand an executable, distributed implementation in C++ communicating via the Message Passing Interface (MPI [For94]). They describe a generalized method for generating code for I/O automata described by operational pseudocode. Unfortunately, the general implementation strategy described uses costly reservation-based synchronization methods to avoid deadlock and a probabilistic, exponential back-off to avoid livelock in the reservation system itself. For certain automata, they are able to optimize this reservation system. Their methods do not rely on a formal language to describe I/O automata and have no direct connection to any verification support.

2.6 IOA Language

The history of success using I/O automata to analyze and verify complex distributed systems and the various efforts exploring tool-based verification methods clearly indicates the importance of the techniques being developed. In the previous modeling work, I/O automata were described using *pseudocode*. By introducing programming style notations, pseudocode simplified and clarified a naive application of the strictly mathematical set notation used to define the I/O automaton model. For example, states were represented by state variables rather than just by members of an unstructured set; state transitions were described in precondition-effect style, rather than as state-action-state triples.

To promote wider application of the I/O automata-based techniques and to support the application of automated verification tools to I/O automata, Lynch and Garland introduced the *IOA language* [GLV00]. IOA is a formal language for describing I/O automata and their properties. IOA serves as both a formal specification language and a programming language. I/O automata described in IOA may be considered either specifications or programs. In either case, IOA yields precise, direct descriptions of I/O automata constructs. The language has explicit constructs for the signature, state variables, and precondition-effect style transitions. Since the language is intended to serve both as a specification and programming language, it supports both axiomatic and operational descriptions of programming constructs. Thus state changes can be described through imperative programming constructs like variable assignments and simple, bounded loops or by declarative predicate assertions restricting the relation of the pre-state to the post-state.

The language also directly reflects the nondeterministic nature of the I/O automaton model. Rather than add a few constructs for concurrency and interaction onto a basically sequential language, IOA is concurrent from the ground up. One or many transitions may be enabled at any time. However, only one is executed at a time. The selection of which enabled action to execute is the source of *implicit nondeterminism* in the language. The **choose** operator provides *explicit nondeterminism* in selecting values from (possibly infinite) sets. IOA includes explicit constructs for composing automata and hiding internal transitions of the resulting composition. In addition, explicit description of simulation relations between specifications and their implementations supports description by levels of abstraction.

IOA has been designed to work closely with the Larch Shared Language (LSL) [GHW85, GHG+93]. All datatypes used in IOA programs are described formally in LSL. This gives a full axiomatic description of each datatype and its operators in first-order logic. These specifications provide sound bases for proofs. An LSL specification defines a theory in multisorted first-order logic. The basic unit of specification in LSL is a *trait*, which defines a set of axioms for a logical theory and which makes claims about the consequences of that theory. The semantic meaning of each IOA datatype is given by a corresponding LSL trait (which may reference other traits).

2.7 IOA Toolset

Creation of the IOA language is the first step in building an integrated software development environment for distributed systems. This environment will support algorithm design, development, testing, and formal verification. The IOA language enables designers to precisely specify automata, their properties, and their relations while spanning many levels of refinement. In addition, the IOA language admits the application of a variety of automated tools to support verification and analysis. The *IOA toolset* will connect both lightweight tools (syntax checkers, simulators, model checkers) and heavyweight tools (theorem provers) [GL98]. We will use IOA as a common basis to allow a designer (or design team) to apply any or all of these tools to a single design. Since IOA is a formal language with rigorous, mathematically-based (as opposed to linguistically-based) semantics, the toolset can automatically translate between the IOA description of an I/O automaton and the equivalent "native" language description for each tool. We hope that the IOA toolset will encourage distributed system designers to use a variety of formal methods in the design, analysis, and verification of their systems. The toolset will lower the barriers between disciplines such as model checking and theorem proving and make both types of tools more accessible to distributed system developers. The IOA toolset is designed to allow developers to explore a design space, refine from high-level conception to low-level implementation, and define and verify system properties — all within a common framework.

We have identified a representative collection of tools to target for the initial toolset development. The toolset is divided into a common *front-end* for inputting IOA programs and a variety of *back-end* tools. The front-end consists of the parser, static semantic checker, and the composer. Tools that can be connected at the back-end include an IOA simulator, model checkers, theorem provers, and a compiler.

The first module to which any IOA program is submitted is the *parser* and *static semantic checker* (or simply the *checker*). In addition to the obvious functions, the checker produces an intermediate representation suitable for use by other tools. This S-expression-based intermediate language (IL) has a simpler parse tree than the more readable IOA source language [RR00]. As the checker acts as a front-end to just about all other elements of the toolset, the IL provides a convenient interchange language within the IOA toolset. A checker prototype has been implemented. The IL representation is semantically equivalent to the source representation. In the future, the checker will be able to emit *proof obligations* that the language specification requires to be true but cannot be easily checked. These obligations can form the basis of further verification work with other tools.

The *composition tool (composer)* is designed to convert the description of a composite automaton (a collection of other automata) into *primitive form* by explicitly representing its actions, states, transitions, and tasks. The IOA language includes a **components** statement, which defines an automaton to be the composition of referenced automata. The composer expands the composition statement by instantiating and combining the referenced automata as described by the logical composition operation on the model. In the resulting description, the name of a state variable is distinguished by the names of the components from which it arises. The input to the composer must be a *compatible* collection of automata; for example, the component automata should have no common output actions. Note that composition is a semantically "neutral" operation. That is, the I/O automata described by the input program using the **components** statement is equivalent to that described by the output primitive program. A prototype of the composer is actively under development and will be integrated as part of the checker.

The *simulator* [Che98, RR00] performs sample executions of an IOA program, running them on a single machine. The user can help select the executions that are run. The simulator checks that proposed invariants are indeed true in all states reached during the course of the simulation. A novel aspect of the simulator is its ability to perform *paired simulations*. In this mode, two automata — a specification and a purported implementation — are run simultaneously. The simulator verifies that a proposed step correspondence maintains the given simulation relation between the two automata (again, only in the states reached during the run).

Initially, the IOA toolset will connect to the Larch Prover (LP) [GG91].[2] Theorem provers can be used to prove validity properties for IOA programs, facts about the datatypes manipulated by programs, invariants of automata, and (forward and backward) simulation relations between automata. In the interface under development, the checker will translate IOA descriptions of automata into axioms that can be used by the theorem prover. When a validity condition for an automaton is too hard to establish by static checking, the checker can formulate this condition as a theorem that must be proved. It can also formulate sets of lemmas that imply that asserted invariants and simulation relations are indeed invariants and simulation relations. Users can interact with a theorem prover to prove that the lemmas follow as consequences of the axioms. In some cases, the theorem prover can prove a lemma automatically, but usually the user must interact with the prover to suggest proof strategies and other useful information.

Model checkers provide an approach to validation that is complementary to theorem proving and to simulation. They work completely automatically, and can be used to validate all executions of a finite-state system with a sufficiently small number of states. The IOA toolset is designed to utilize existing model checkers. The first model checking prototype interface was targeted to Spin [Hol91]. A model checker can be used as a validator of invariants and possibly simulation relations provided by the user.

All of the above tools work within the IOA framework. These tools allow designers to perform a variety of design, specification, and verification tasks. Even with all this technology and these tools, however, the

[2]Independent of this toolset project, Devillers has used the IOA language specification [GLV00] to build a tool that translates IOA into input for PVS [Dev99].

task of the distributed system builder remains outside the model. The builder must translate the designers' requirements (now formally described) into a standard imperative programming language. In essence, the system builder starts over and recodes the whole project. As a result, a disconnect exists between the specification and the actual running code. Part of the goal in designing the IOA language and toolset is to bridge this gap. The focus of the work in this thesis will be to build a tool to bridge that gap, an *IOA compiler*. The IOA compiler will translate a restricted subset of IOA programs into a standard imperative language (Java). The resulting code will run on a collection of workstations communicating using standard networking protocols.

3 Problem

The goal of the proposed work is to develop and evaluate an IOA-to-Java compiler for distributed algorithms. In the context of the IOA toolset project, this compiler will be a key component in a software development environment. We will demonstrate the it is feasible for programmers to write their specifications at a high level of abstraction, in terms of input/output automata; validate the specification using a variety of tools; refine the specification to a low-level design; and then automatically translate the design into code that runs on a collection of workstations communicating using standard networking protocols.

We are not aware of any current system that integrates specification, light-weight validation tools, automated proof assistance, and automated code-generation. While specifications can conform closely to actual system code, there remains a disconnect. A human must translate the specification into an imperative language description. This disconnect between the formal specification of a distributed system and the actual implementation of that system creates several problems.

In the following section, we will discuss these existing difficulties and the benefits to be expected from such integration. In Section 3.2 we will describe some of the key challenges we expect to face in building the compiler and explain why we are optimistic that we can overcome these challenges.

3.1 Importance of an IOA-to-Java Compiler

Overcoming the existing disconnect between formal specification and system implementation will solve several existing problems and create many new opportunities for distributed system work.

At a minimum, coding the system in a programming language distinct from the specification language requires a duplication of effort. The task of the system implementor has significant overlaps with that of the specification writer. They are, after all, both striving to construct descriptions of the same system.

The separation of the specification and implementation processes allows for the introduction of transcription and interpretive errors or the failure to notice modeling errors. Transcription errors are simple copying errors that result because humans do not perform copying reliably. Interpretive errors are errors of understanding of one sort or another. The formal specification may contain ambiguous abstractions. The system implementer, less familiar with the specification language than the specification writer, may misinterpret the intent of said writer. Alternatively, the implementer may introduce "optimizations" that cause errors. (The optimizations may either be to the coding process or to the code itself.) Modeling errors occur when the specification does not adequately reflect the behavior of the real world. The overhead of the specification process may discourage the system implementer from pointing out such errors in the specification model. As a result, any confidence in the correctness of the specification transfers to the implemented system only in substantially diminished form.

Automating the process of translating IOA specifications into running code has power beyond eliminating errors and duplicated effort. By substantially increasing the automation in the specification process, the IOA toolset will substantially lower the barriers between formal methods and system building work. With the addition of the IOA compiler, the IOA toolset becomes a testbed for distributed system designers. Algorithm writers will have a relatively simple path to create and measure running prototypes of high-level designs. As a result, we hope to lower the implementation expertise necessary to create even a simple prototype of a distributed system. Similarly, the formal IOA language, automated validation assistants, and clear connection to running code decrease the amount of mathematical sophistication needed to perform basic validation work. Ultimately, we hope these tools lower the barriers and increase the amount of cooperation and cross-fertilization between these two aspects of the distributed systems field.

By decreasing the costs involved, the IOA toolset should promote the iterative nature of the formal specification and system building process. Making changes to the specification becomes easier to carry through. Where previously any change required tediously revisiting each step of hand proofs and hand translations, in many cases we can revalidate the design automatically. This should raise the level of abstraction at which system design takes place and should result in more maintainable designs.

3.2 Feasibility

Translating high-level, nondeterministic IOA specifications into running Java will present several challenges. We will have to produce a schedule — a deterministic order for operations to occur. We will have to implement axiomatically described datatypes. We will have to cope with some of the more expressive features of the language such as explicit nondeterministic choice from (possibly infinite) sets. We hope to produce code that runs efficiently.

Previous experience provides a basis for optimism. Work by Lynch with Hickey and van Renesse at Cornell [HLvR99] on modeling and validation of the Ensemble system using IOA-based methods suggests that it is not hard to write and verify algorithms in IOA in such a way that they are essentially isomorphic to the actual running ML code for the Ensemble system [Hay96]. The existing ML code has good performance.

Cheiner and Shvartsman [Che97, CS98, CS99] defined a structured method for hand translating I/O automata into sequential code running on workstations connected via standard networking protocols. The method outlined in Cheiner's work is sufficiently straightforward that automating the process seems entirely tractable. The work included implementation of a fairly sophisticated, layered distributed system. Cheiner reports the running system had good performance.

IOA is sufficiently flexible to be used at different levels of abstraction, including a very low level that can permit extensive optimization within IOA itself. By requiring and assisting the user to transform the specification into another IOA program but at a low level of abstraction and using a restricted subset of the language, we expect to overcome many of the above challenges. We give the user flexibility in controlling the order in which actions are performed in order to yield more efficient schedules for local node programs than would arise from a general scheduling discipline. In addition we will use a source-to-source translation to Java to take advantage of prior work on optimizing compilers. Also, by proving some properties statically, we should be able to avoid the expense of checks that would typically be done at runtime.

One aspect of this work will be to evaluate the expressive power of the resulting system. It is unclear how much overhead will be introduced by these transformations.

4 Approach

We divide the proposed work into three areas: design and implementation of the compiler, modeling and correctness analysis of the compiler, and experimental evaluation of the compilation environment. In this section we describe our overall approach to each of these tasks. In Section 5 we revisit each area to describe the specific work required.

4.1 Compiler Design

The compiler currently under development is designed to translate IOA programs into executable programs written in a standard imperative language. The programmer starts with a distributed algorithm expressed in IOA and uses the tool to produce executable code that runs in a physically distributed computational environment. In our current target environment, the resulting collection of programs runs on a group of networked workstations. Each host runs a Java interpreter and communicates via (a subset of) the Message Passing Interface (MPI) [For94, BCKL] or a TCP/IP-based library [KK00]. This collection of programs can be proved equivalent to the original algorithm, subject to stated assumptions about the behavior of externally provided system services (*e.g.*, communication services), the correctness of hand-coded datatype implementations, the threading and locking model of Java, and the correctness of the code-generator implementation.

Our compilation method follows the general strategy of relying on the programmer (with support from IOA tools) to put the program in the proper form for input to the compiler. To prepare for the compilation process, the programmer performs a series of successive refinements to transform the original expression of the algorithm in IOA into a more detailed form of the program that is suitable for automated translation. Often, the process begins with a simple, global description of the behavior of the system and its interface to the environment. This high-level model tends to be easy to understand and to have important global properties that can be proved. The programmer refines that model into a low-level version of the program that is a collection of interacting automata whose form corresponds to the distributed nature of the target environment while preserving the important properties and interface of the system. The programmer can use the validation tools included in the toolset to confirm the correctness of these refinements.

Once the programmer has an automaton designed to perform all local computation for a single node of the distributed system, the code generation process can begin. We call such a program an *algorithm automaton.* If some computation nodes in the distributed system must execute different algorithms, the compilation process is repeated for each such distinct algorithm automaton. For example, the programs implementing clients and servers would be compiled separately while the code for parameterized peers can be compiled just once.

The compilation process consists of a series of transformations of the algorithm automaton. The compiler is structured as a set of small modules, each of which performs one of these transformations. During these transformations, the algorithm automaton is combined with interfaces to external system services and transformed to eliminate implicit nondeterminism. The programmer then resolves the remaining explicit nondeterminism. The resulting IOA program can then be automatically translated into the target imperative language and linked to libraries that implement the external system services. Each transformation is source-to-source within the IOA language (or in an internal intermediate form) up to the last step. Only in that last step is the IOA program translated into the target language.

In the last step, the *code generator* translates the IOA code into a Java program. The program starts from a unique initial state and performs a loop in which, at each iteration, it executes the unique action enabled in the current state. More specifically, it executes a distinguished scheduler transition to determine the next transition, then executes that transition. Since there is no nondeterminism, this uniquely determines the next state.

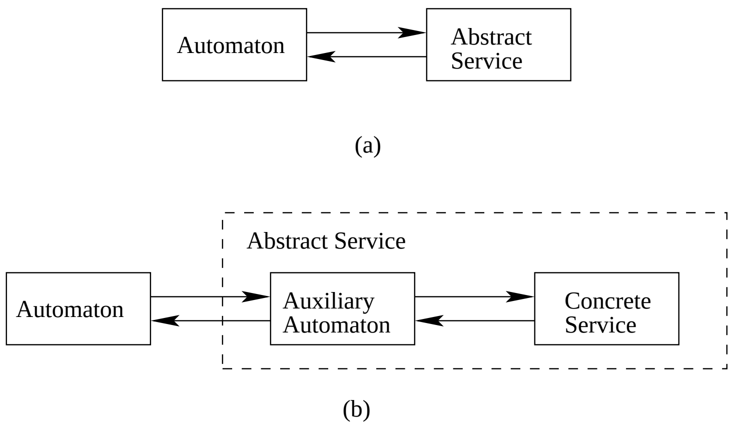

Figure 1: Auxiliary automata mediate between external system services and algorithm automata. (a) An abstract service defines the desired behavior. (b) The composition of the concrete service and the auxiliary automaton implements the abstract service.

4.1.1 External System Services

The compiler does not generate code for the entire distributed system. Rather, it generates the specialized code necessary to implement the algorithm at each node. As with any programming system, newly written programs leverage preexisting, external system services. Programs connect to those services when running.

For each external system service, we design at least two models. First, we write an *abstract model* that describes the interface and the behavior of the service that the IOA programmer wants to use (*e.g.*, point-to-point, reliable, FIFO channels as in [Lyn96], Chapter 8). Second, we write a lower-level *concrete model* that corresponds to the interface and behavior of the actual preexisting service (*e.g.*, MPI [For94, BCKL]). We then introduce one or more *auxiliary automata*, written in IOA, that compose with the lower-level model to implement the abstract service. (See Figure 1.) Interfaces to system services may be fixed or algorithmically described. In the former case, the auxiliary automata are fixed and kept as a hand-coded library. In the latter case, the auxiliary automata must be generated automatically for each program. To create a complete executable for each node, the compiler actually emits code for the *composition* of the algorithm automaton and all the auxiliary IOA automata for all the system services the algorithm accesses.

Each external system service introduces constraints on the form of the low-level IOA program that accesses it. For example, since the compiler uses standard workstation networking services, programs submitted for compilation are required to be in *node-channel* form. That is, the algorithm must be described as a collection of primitive algorithm automata (one per node) that communicate via standard message passing channels without any global synchronization.

We introduce another (technical) constraint. Atomicity requires that the effect part of each transition be done without interruption, even if inputs arrive from the external user or from the communication service during transition execution. In our design, such inputs are buffered. Between running non-input actions, the generated program examines buffers for newly arrived inputs, and handles (some of) them by running code for input actions. Since the processing of inputs is delayed with respect to the performance of these actions by their originators, there is the potential to upset precise implementation claims for the algorithm automata. We require that any algorithm submitted for compilation be able to tolerate this input buffering. Stated in its strongest, simplest form, the restriction is that each algorithm automaton be *input-delay-insensitive*: its external behavior should not change if its input actions are delayed and reordered before processing. It is possible to weaken this requirement slightly.

By insisting that IOA programs from which we generate real code match the available hardware and services, and by requiring algorithm automata to tolerate input delays, we avoid the need for expensive non-local synchronization in achieving a faithful implementation.

4.2 Modeling and Correctness Analysis

Our approach to correctness analysis for the compiler is to apply standard I/O automaton modeling and verification techniques to the compilation process itself. We not only model the submitted distributed program and its components, but we model the emitted Java code too. Similarly, not only do we specify abstract services with which the program is designed to interface (*e.g.*, reliable FIFO channels and the user environment), we model actual existing external system services (*e.g.*, MPI and the I/O console). We will use implementation relation proofs to show that if our models of the real world services, the emitted code, and the code generation process are accurate, then the code emitted actually implements the input IOA program.

4.2.1 Code Generation

Our main theorem concerns the code generation process itself. This meta-theorem gives a correctness condition for the code generator. We will model the emitted code as automata with multiple threads of control, locks, and very fine granularity of atomicity (essentially arbitrarily small except for locking primitives). We will then show how to derive these low-level automata from a simplified model of IOA programs called *structured automata* by the application of threading and locking. We call these low-level automata the *threaded versions* of structured automata. We will also show how to derive a "standard" I/O automaton (or *atomic version*) from the structured automaton. The meta-theorem says that for a restricted class of IOA programs (those acceptable to submit to the code generator) the application of particular threading and locking disciplines (those used by the code generator) always generates threaded versions of structured automata that implement (in the sense of trace inclusion) the atomic version of the given structured automata. We will pay special attention to carefully model the interface between the emitted code and external system services. From this theorem we can conclude that if the code generator applies the described threading and locking discipline, then the emitted code has the desired semantics. (See Figure 2.)

Extensions to the compiler (*e.g.*, fault-tolerance guarantees or support for alternate networking protocols such as TCP) will require additional modeling and verification.

4.2.2 External System Services

As we described in Section 4.1.1, we will specify each abstract service, model its concrete service, and design auxiliary automata to mediate between the two. There is a once-per-service proof obligation that the composition of the concrete model and the auxiliary automata implements the abstract service. Once the proof is done, the programmer can use the properties of the abstract model subject to the assumption that the concrete model corresponds to the behavior of the service.

There are two services of interest for the initial prototype. The MPI network service will provide communication channels while the console will provide the user interface. Since we use a fixed interface between algorithm automata and the communication channels, we need to design and verify only one set of auxiliary automata for networking.

However, we allow programmers to select the user input and output actions for each algorithm compiled. In so doing, the programmer implicitly defines an abstract environment automaton with which the algorithm automaton expects to interact. As a result, custom auxiliary automata must be automatically generated

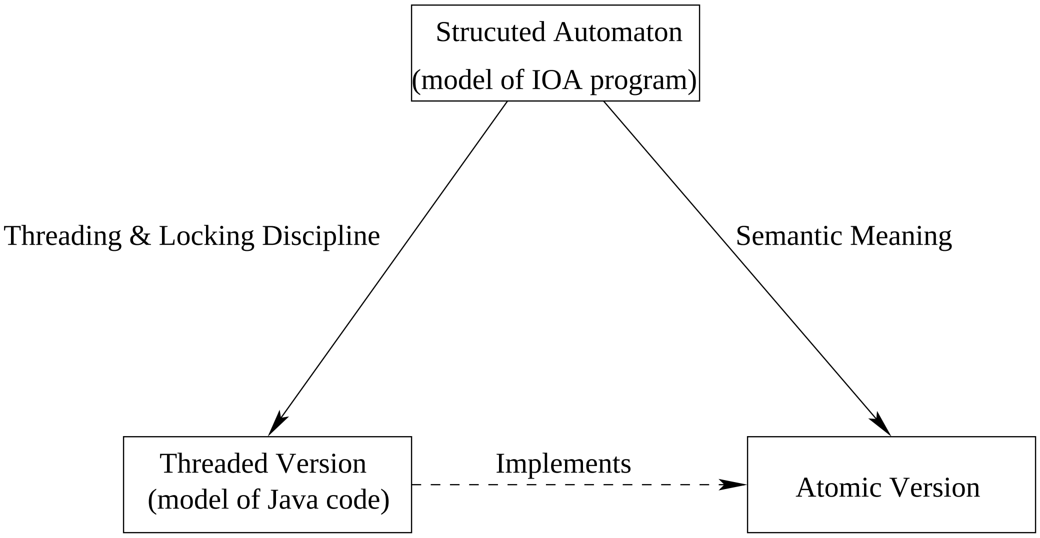

Figure 2: *Structured automata* model IOA programs. We derive both *threaded* and *atomic versions* of a structured automaton. The former models the code generated by the IOA compiler. The later encapsulates the semantics of the program. We will prove a theorem that for the class of automata with which we are concerned and for the threading and locking disciplines used by the code generator, the threaded version implements the atomic version. Thus, the generated Java code implements the desired semantics.

to mediate between the console and the algorithm automaton. To verify the implementation of the user interface we must construct a meta-theorem that the auxiliary automata generated always compose with the model of the console to implement the implicitly defined environment.

4.3 Experimental Evaluation

We will use three types of evaluation to assess the IOA compiler. First, we will measure some basic properties of the emitted code. By measuring the code we hope to lay some quantitative benchmarks for comparing this prototype to any alternative implementations or future optimizations. Second, we will compare emitted code to hand-written code for the same high level algorithms. We will use both quantitative and qualitative assessments for comparison. Using these comparisons, we hope to gain some insight into the quality and performance of the emitted code. Third, we will exercise the expressive range of the compiler by implementing a series of algorithms. We will select a range of algorithms to compile in order to exercise key features of the compiler.

5 Proposed Work

We divide the proposed work into three areas: design and implementation of the compiler, modeling and correctness analysis of the compiler, and experimental evaluation of the compilation environment. In this Section we describe the specific work required in each area.

Table 1: Transformers in the IOA compiler. Numbers in the input column indicate that the transformer takes in the output of the numbered step. The flow of an IOA program through these transformers is depicted in Figure 3

Step	Transformer	Output	Input
1	Checker	internal form	algorithm automaton
2	Interface generator	auxiliary interface automaton	1
3	Composer	primitive automaton	1, 2, auxiliary automaton library
4	NAD	next-action deterministic automaton	3
5	Schedule editor	next-state deterministic automaton	4
6	Code generator	Java source	5, datatype implementation library

5.1 IOA Compiler

The IOA compiler design consists of a series of small program transformers. Each accepts an IOA program or programs as input and produces another IOA program as output (except the last, of course, which translates IOA into the target language). Several of the modules are used by some, if not all, of the other tools in the IOA toolset. Table 1 lists the steps in the process while Figure 3 illustrates the flow of an IOA program through the compiler. The granularity of the transformation is driven largely by the usefulness of mixing and matching these tools. For example, the checker and composer are used by nearly every back-end tool in the toolset and, therefore, remain separate from the interface generator. Programmer input is required between the NAD[3] transformation and code generation, so a separation is maintained between those tools. In addition, the source-to-source nature of these transformers allows us to consider alternative designs for each of these transformations. For example, alternate user interface designs (say, graphically-based) could be supported. Similarly, we could apply a variant of Ramirez's NDR scheme [RR00] to resolve nondeterminism to replace the NAD and schedule editor modules. Below, we describe each step in the process. We briefly sketch the function of each module and show how it is used in the compilation process.

The first module to which any IOA program is submitted is the checker. As described in Section 2.7, the checker parses and statically checks the algorithm automaton, which is designed to perform all local computation for a single node of the distributed system, and produces an intermediate representation suitable for use by other tools.

5.1.1 Connection to External System Services

The *interface generator* or *parser generator* connects an algorithm automaton with the external *console* system service. The console service embodies all user interaction with an IOA program at a particular node. The interface generator creates a customized auxiliary parser automaton to mediate between the console service and the algorithm automaton. The auxiliary automaton parses input from the console into input actions for the algorithm automaton. Similarly, output from the algorithm automaton is forwarded to the console. In our initial prototype the abstract model of user input allows the user to nondeterministically invoke any (non-network) input of the algorithm while the concrete model of the console is a simple source and sink for streams of integers. A prototype interface generator has been implemented by Tsai [Tsa00].

In the prototype programming environment, each computation node is connected to a point-to-point, one-way, reliable FIFO channel to every node in the system as described in Chapter 8 of [Lyn96]. This provides the IOA programmer a simple, fixed, send/receive communication interface. However, the actual compilation target is a group of networked workstations. Each host runs a Java interpreter and communicates via a small subset of the Message Passing Interface (MPI) [For94, BCKL]. To mediate between the actual MPI interface

[3] NAD stands for next-action deterministic, and will be introduced shortly.

and the simple channel interface, we will design auxiliary network automata. As both the MPI interface and the IOA channel interface are fixed, the auxiliary automata can be fixed as well

The selection of target network protocol is orthogonal to the design. We selected MPI partly because of historical accident (Cheiner and Shvartsman used it for their initial design [CS98]) and partly for its suitability for this implementation effort. MPI has simple semantics that are documented relatively well for a networking protocol. Most importantly, MPI has asynchronous communication primitives that allow each node to make progress without reference to progress by any other node. The chief drawback to using MPI is that it has no adequate error model so we cannot use it to implement fault-tolerant algorithms. MPI is available for a huge variety of concurrent and parallel programming platforms. To study fault-tolerant algorithms and the design of the compiler-network interface, we may extend our design to work connect to a TCP/IP-based library [KK00].

To connect the input program to the external system services, we combine the algorithm automaton with the generated parser automaton and the fixed auxiliary network automata using the composer (described in Section 2.7). The resulting primitive automaton describes all the behavior of the executable code we actually wish to generate for that node. Unfortunately, it is not obvious how to directly translate this nondeterministic form of the program into a standard imperative language. The next section discusses our solution to this problem.

5.1.2 Elimination of Nondeterminism

One aspect that distinguishes IOA programs from programs written in an imperative language is the presence of nondeterminism, which comes in the form of explicit nondeterministic statements and implicit scheduling choices made during execution. Compilation, therefore, consists partially of systematically removing all forms of nondeterminism. Our design for the IOA compiler breaks this into two steps. First, we automatically remove all instances of implicit nondeterminism, making them explicit statements. Second, we have the programmer replace all explicit nondeterminism with deterministic statements.

An I/O automaton (and by extension, an IOA program) is *next-action deterministic* (NAD) if, in all reachable states, there is at most one enabled, locally-controlled action. *Next-state determinism* is a stronger constraint on I/O automata in which not only is there a unique enabled action to execute but, for any given state, the (uniquely) enabled action always results in a unique post state. Most IOA programs that are written are not next-action deterministic. They have a form of *implicit* nondeterminism, which consists of nondeterministically choosing the next enabled action to execute.

The syntactic NAD module converts the input IOA program into an equivalent next-action deterministic program. The NAD form of an IOA program has no implicit nondeterminism. Explicit nondeterminism is grouped in a new internal *Scheduler* transition that contains a set of **choose** statements. Values are selected nondeterministically, subject to a condition specified after the keyword **where**[4]. In this stylized form, each nondeterministic choice is assigned to a state variable of the automaton. A prototype NAD module has been implemented by Tsai [Tsa00].

The final step before actually emitting imperative code is to remove the grouped, explicit nondeterminism from the automaton. The resulting program is both next-action and next-state deterministic. In our prototype, the graphical user interface displays the NAD output to the programmer [Tsa00, Rei99]. The programmer edits the schedule transition by replacing each **choose** expression in turn with a deterministic IOA expression that selects a value from the same set as the **choose** expression. (Showing this is a proof obligation for correctness.) The resulting IOA node automaton is completely deterministic and easily translated into an imperative language.

[4]When the condition is omitted, it is simply true.

5.1.3 Code Generation

The code generation module will translate one primitive, deterministic, node automaton into actual code in the target language that implements the node automaton. For each operation, we emit code from datatype class libraries written in a standard programming language (currently, Java). At present, we do not address the problem of establishing the correctness of this sequential code (other than by conventional testing and code inspection). Standard techniques of sequential program verification (based, for example, on Hoare logic) should be capable of handling such correctness proofs. (Note that the IOA framework focuses on correctness of the concurrent, interactive aspects of programs rather than of the sequential aspects.)

Since all nondeterminism has been removed, the emitted code will start from the unique initial state and perform a loop in which, at each iteration, it executes the unique action enabled in the current state. More specifically, it uses the scheduler transition to determine the next enabled transition, then executes that transition. Since there is no nondeterminism, this uniquely determines the next state.

5.2 Modeling and Correctness Analysis

To enhance our confidence in the design of the compiler, we will prove three key theorems (and rely on another by Vaziri [VF99]). The first theorem describes the relation between the input IOA program and the emitted code. The second two theorems show that the interfaces to the network and console system service are correct. We also describe work we did with Vaziri justifying the use of the NAD transformation.

5.2.1 Code generation

To model the code generation process, we describe two classes of I/O automata. The first describes the behavior of the "high level" input algorithms to the code generator. The second describes the behavior of the "low level" code emitted by the code generator. The key theorem for the code generator states that the low-level, generated code implements (in the sense of trace inclusion) the high-level specification.

A key novel aspect of this theorem is how we describe these two classes of automata. We start from a common base and derive the two classes of automata from it. That base we call *structured automata*. Structured automata are state machine models for IOA programs. As the name implies, structured automata encompass the form of an IOA program. Each of the key elements of the program are there: signature, state variables, transition relations, even individual program statements. However, we do not provide a unique execution model for this structure. Instead, we apply *two* mathematical interpretations of execution.

The *atomic version* of a structured automaton yields semantics that coincide with the standard, large granularity, semantics of I/O automata. Atomic versions of structured automata are what IOA programmers typically *mean* for their IOA programs to do. In particular, each transition definition executes as one atomic unit. All program statements of a transition are executed in one step.

The *threaded version* of a structured automaton has finer-grained semantics than the atomic version. The execution corresponds to the important aspects of how the generated code runs on a real machine. In particular, each program statement (not an entire transition) is executed in one step. We model the multiple threads of control used to execute the code. For each thread, we model a program counter that tracks exactly which step of the code is to be executed. In addition, we will model locks to synchronize data sharing between the threads. The specific application of threads and locks used to interpret a structured automaton as its threaded version is intended to correspond directly to how the code generator and Java runtime system introduce threads and locks into the code generated from an IOA program.

The crux of the theorem is to show that given a structured automaton belonging to a restricted class (those

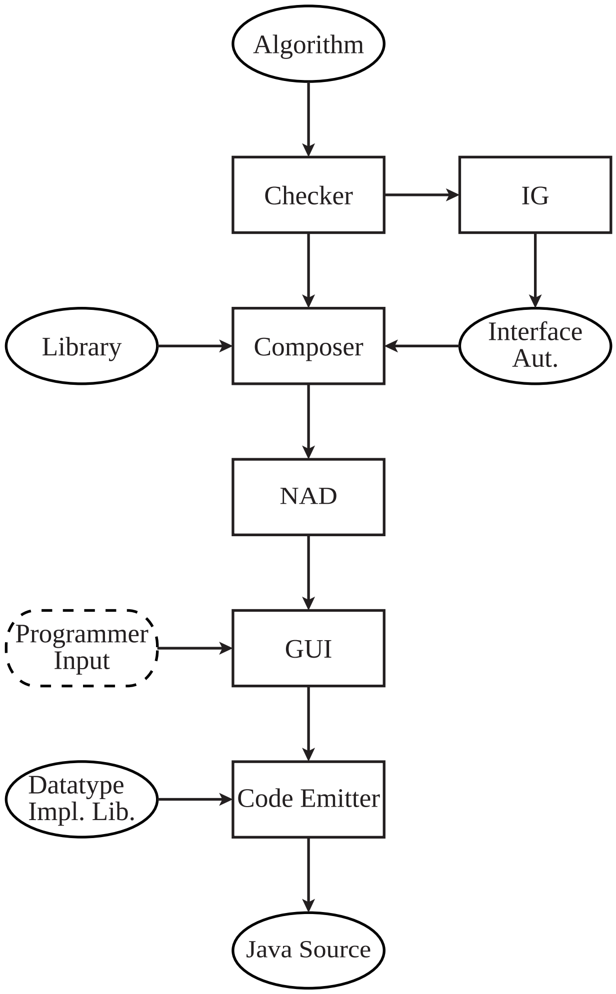

Figure 3: The *checker* parses and performs static semantic checks on algorithm automata. The *interface generator* (IG) produces an auxiliary automaton to connect the algorithm automaton to the console. The *composer* expands the composition of the algorithm automaton, the auxiliary interface automaton, and the library of auxiliary network automata into one primitive scheduler automaton. The *NAD* removes implicit nondeterminism from the input automaton. The *code emitter* translates the fully deterministic scheduled automaton into Java using the (handwritten) datatype implementation library. These transformers in the compilation process are tabulated in Table 1.

that the code generator will accept) and applying a specific thread-and-locking discipline to it (those used by the code generator), we generate a threaded version that implements the atomic version of the same structured automaton. That is, we will show a simulation relation between the states of the threaded and atomic versions of any structured automaton in this class. We will show this by demonstrating a step correspondence between the threaded version and the atomic version. That is, for any step the threaded version takes from a corresponding state, there is a (possibly empty) sequence of steps the atomic version can take that maintains the correspondence.

5.2.2 Connection to External System Services

Following the method described in Sections 4.1.1 and 4.2.2 we will model each abstract service, the concurrent service we use to implement it, and the auxiliary automata that mediate between them. We will prove an implementation relation for each external system service accessed by the generated code.

For the MPI network service, we will use four automata. The first automaton is the abstract service specification. For that we use the point-to-point, one-way, reliable FIFO channel automaton as described in Chapter 8 of [Lyn96]. The second automaton we will develop will model those features of the MPI communication system to which the code generator interfaces. Finally, we will design two auxiliary automata to connect to MPI: a send automaton and a receive automaton. The send automaton receives messages from the algorithm automaton and dispatches them into the MPI communication system. Analogously, the receive automaton receives messages from MPI and delivers them to the algorithm automaton. Since both the algorithm automaton and MPI interfaces are fixed, the automata are fixed too. To show the correctness of our implementation, we will prove an implementation relation from the composition of the MPI automaton and the two auxiliary automata to the specification one-way channel automaton.

Verifying our implementation of the console system service is more involved than verifying the network service. In defining the non-network input and output actions for an algorithm to be compiled, a programmer implicitly defines the interface of an abstract environment automaton. In the most general specification, this environment automaton is a nondeterministic source for those inputs and a ready sink for those outputs. To implement these environments, the generated code connects to the user console. The interface generator creates a custom auxiliary automaton for each submitted algorithm automaton. The parser bundles input from the console into input actions expected by the algorithm automaton and reverses the process to send the outputs from the algorithm automaton to the console. Since we have no fixed abstract environment automaton and no fixed auxiliary automaton, we cannot prove a simple implementation relation. However, we will develop a meta-theorem that the auxiliary automata generated always compose with the model of the console to implement the implicitly defined environment. This theorem will constitute a correctness condition for the algorithm employed by the interface generator.

5.2.3 Elimination of Nondeterminism

[VTL] presents a careful examination of a transformation that makes implicit nondeterminism explicit. That study describes the transformation both at the level of IOA programs and at the level of I/O automata. We call the former the *syntactic NAD transformation*. We call the latter the *semantic NAD transformation*. The motivation for the latter transformation is to provide a more general framework for describing the conceptual transformation (of making implicit nondeterminism explicit) and to present it in its full generality, unconstrained by program syntax. The semantic NAD transformation is used to prove that the syntactic transformation is "correct," and conforms to it, in a formal sense.

[VF99] presents the formal proofs of the claims made in [VTL]. First, Vaziri shows the soundness of the semantic transformation. The semantic NAD transformation is formally defined by a relation on I/O automata. Vaziri then shows that I/O automata in the set resulting from the semantic transformation have the same traces as the input automaton and that the related automata satisfy next-action determinism.

Second, Vaziri shows that the syntactic transformation corresponds to the semantic one. To make this claim of correctness precise, Vaziri uses the formal semantics of IOA programs as defined in [GLV00]. The semantics define an I/O automaton for every IOA program. In a formal sense, the I/O automaton gives meaning to the IOA program. The precise claim proved in [VF99] is that the I/O automaton defining the semantics of the syntactically transformed IOA program is in the set of I/O automata produced by applying the semantic transformation to the I/O automaton defining the semantics of the input IOA program.

5.3 Experimental Evaluation

As discussed in Section 4.3, we propose three types of evaluation to assess the IOA compiler: simple measurement, comparison to hand code, and a range of compilations.

We will extend the compiler to instrument emitted code so we can measure time spent in handling messages, scheduling, and in executing auxiliary automata. We will also instrument the code to verify input/output behavior of simple algorithms. We will then compile a few simple algorithms with instrumentation to measure basic performance and set benchmarks for improvement.

We will both compile and hand translate basic distributed algorithms from the literature to compare the quality of the generated code. We will use some basic data management algorithms from Chapter 17 of [Lyn96] (based on work in [Gif79, GL87, ABND95]) for the comparison. Some bases of comparison for hand-written code and emitted code include:

- code size
- performance
- readability
- programmer time

We will select a range of algorithms to compile in order to exercise key features of the compiler. We have identified the following features of the language and compiler design:

Basic distributed functionality. Show a distributed message-passing algorithm runs and passes messages as expected.

Built-in datatypes. Employ the full catalog of IOA datatypes, datatype constructors, and record types. The basic datatypes are booleans, integers, natural numbers, real numbers, characters, and strings. The type constructors are arrays, sets, multisets, sequences, and mappings. The record types are enumerations, tuples, and unions.

New datatypes. Introduce new or enhanced datatypes not in the basic IOA language. Example datatypes we might use are priority queues, heaps, or trees.

Periodicity. Demonstrate an algorithm that must perform some task periodically (without response to input). This represents a minor challenge as there is no notion of time in the basic I/O automaton model *per se*. However, we can use devices such as a local counter to overcome this. Implementation of a particular algorithm will likely require a certain amount of performance tuning.

Composition. Demonstrate compilation of an algorithm automatically derived from the composition of services at distinct levels of abstraction.

We will exercise basic functionality with built-in datatypes by implementing the same toy message-passing algorithms we use for instrumentation and measuring. To exercise a range of datatypes, we may implement leader election algorithms found in Chapter 15 of [Lyn96] (based on work in [Lan77, CR79, Pet82]). To demonstrate the addition of non-standard datatypes and periodicity we may implement totally ordered broadcast using Lamport Time [Lam78] and priority queues. We can then compose a replicated state machine implementation with totally ordered broadcast to implement an atomic object as described in Chapter 18 of [Lyn96].

5.4 Extensions

We may consider extending the design by retargetting the compiler to replace the MPI-based network service interface with a TCP/IP-based sockets interface [KK00]. One reason to extend the design to work with TCP is to expand the expressive power of the programs we can compile. TCP, unlike MPI, explicitly supports connection establishment and tear down. As a result we could implement a range of algorithms supporting nodes that dynamically join and leave a computation. In addition, we may be able to support algorithms that require failure detection and include fault-tolerance properties. The literature on group membership and group communication provides a variety of candidate algorithms for implementation. For example, we may implement a simple membership algorithm. A survey of group communication algorithms can be found in [VKC01].

Since the most available and best documented and modeled TCP library is in written in C++, emitting TCP based code would likely entail retargetting the compiler to to emit code in C++. These extensions would allow us to evaluate the difficulty in replacing one target language or network service with another. The compiler design is specifically intended to be orthogonal to these choices of target. We could evaluate any extended implementation on the basis of the above methodology.

Retargetting the compiler to TCP would also entail extending the modeling and verification work described in Section 5.2. First, we would need to model the new system service (TCP). We will probably need to update the definition of the abstract network service to consider errors. Similarly, it is unclear whether the code generation meta-theorem would need to be extended to cover behaviors enabled by TCP.

6 Milestones

Following is a list of goals of this research, which corresponds approximately to the planned outline of the thesis.

- Design
  - Design compilation process — Tauber — complete
  - Design parser generator — Tauber — complete
  - Design NAD — Tauber — complete
  - Design composer — Chefter, Garland — 90% complete?
  - Design code generator — Tauber, Tsai — 90% complete
- Implementation
  - Build and test parser generator — Tsai — complete
  - Build and test NAD — Tsai — complete
  - Build and test composer — Garland

- Write and test datatypes — Tsai — complete
- Write and test code generator — Tauber, Tsai — 90% complete

- Modeling and correctness analysis
  - MPI
    - Write MPI_Sys automaton — Tauber — complete
    - Write MPI auxiliary automata — Tauber — complete
    - Define simulation relation from composition of MPI_Sys and MPI auxiliary automata to reliable FIFO channels — Tauber — complete
    - Prove simulation relation — Tauber — complete
  - User interface
    - Describe derivation of environment automaton from algorithm automaton — Tauber
    - Write console automaton — Tauber — complete
    - Describe generation of auxiliary parser automata — Tauber
    - Define meta-simulation relation from composition of console and generated auxiliary automata to environment — Tauber
    - Prove simulation relation — Tauber
  - Code generation
    - Define structured automata — Tauber — 90% complete
    - Define atomic versions of structured automata — Tauber — complete
    - Define threaded versions of structured automata — Tauber — 50% complete
    - Specify locking discipline for threaded version — Tauber
    - Define simulation relation from threaded version with locking to atomic versions of given structured automaton — Tauber
    - Prove simulation relation — Tauber
  - NAD
    - Define syntactic NAD transformation — Tauber, Vaziri — complete
    - Define semantic NAD transformation — Vaziri — complete
    - Prove soundness of semantic transformation — Vaziri — complete
    - Prove correspondence of syntactic and semantic transformation — Vaziri — complete
    - Prod Vaziri to make manuscript a tech report — Tauber
    - cite the tech report — Tauber

- Experimental evaluation
  - Measurement
    - Extend compiler to instrument emitted code — Tauber
    - Write toy message passing example — Tauber
    - Measure example — Tauber
  - Comparison
    - Write examples in IOA and Java — Tauber
    - Measure examples — Tauber
  - Expressiveness
    - Write examples in IOA — Tauber
    - Write new datatypes — Tauber
    - Compile examples — Tauber

        * Test examples — Tauber

- Write-up
  - Agonize — Tauber
  - Write thesis — Tauber
  - Defend thesis — Tauber
  - Sign thesis — Lynch, Ernst, Garland, Rinard
  - Turn in thesis — Tauber
  - Graduate — Tauber

References

[ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.

[AHR00] Myla M. Archer, Constance L. Heitmeyer, and Elvinia Riccobene. Using TAME to prove invariants of automata models: Two case studies. In *Proceedings of Third ACM Workshop on Formal Methods in Software Practice (FMSP'00)*, August 2000.

[BBC+96] Nikolaj Bjørner, Anca Browne, Eddie Chang, Michael Colon, Arjun Kapur, Zohar Manna, Henny Sipma, and Tomás Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *8th International Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418. Springer-Verlag, July 1996.

[BBC+99] Nikolaj Bjørner, Anca Browne, Michael A. Colón, Bernd Finkbeiner, Zohar Manna, Henny Sipma, and Tomás Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design*, pages 1–45, 1999. to appear.

[BCKL] Mark Baker, Bryan Carpenter, Sung Hoon Ko, and Xinying Li. mpiJava: A Java interface to MPI. Submitted to First UK Workshop on Java for High Performance Network Computing, Europar 1998.

[BGLR01] Elizabeth Borowsky, Eli Gafni, Nancy Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm, January 2001. Submitted for journal publication.

[Bie97a] A. Biere. *Effiziente Modellprüfung des μ-Kalküls mit binären Entscheidungsdiagrammen.* PhD thesis, Universität Karlsruhe, 1997.

[Bie97b] A. Biere. μcke — efficient μ-calculus model checking. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 468–471. Springer Verlag, 1997.

[BKT92] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. on Soft Eng.*, 18(3):190–205, March 1992.

[BMSU98] Nikolaj Bjørner, Zohar Manna, Henny Sipma, and Tomás Uribe. Deductive verification of real-time systems using STeP. Technical Report STAN-CS-TR-98-1616, Computer Science Department, Stanford University, December 1998.

[Bra94] R. Braden. Extending TCP for transactions — concepts. Internet RFC-1379, July 1994.

[CGL+94] R. Cleaveland, J. N. Gada, P. M. Lewis, S. A. Smolka, O. Sokolsky, and S. Zhang. The Concurrency Factory - practical tools for specification, simulation, verification and implementation of concurrent systems. In *Specification of Parallel Algorithms. DIMACS Workshop.*, pages 75–89. American Mathematical Society, 1994.

[CHD98] G. Chockler, N. Huleihel, and D. Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *Proceedings of the 17th ACM Annual Symposium on Distributed Computing*, June 1998.

[Che97] Oleg Cheiner. Implementation and evaluation of an eventually-serializable data service. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 1997.

[Che98] Anna E. Chefter. A simulator for the IOA language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, May 1998.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Co., Reading, MA, 1988.

[CPS93] R. Cleaveland, J. Parrow, and B. U. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM TOPLAS*, 15(1), 1993.

[CR79] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, May 1979.

[CR96] S. Chaudhuri and P. Reiners. Understanding the set consensus partial order using the Borowsky-Gafni simulation. In *10th International Workshop on Distributed Algorithms*, October 1996. To appear in *Lecture Notes in Computer Science,* Springer Verlag.

[CS98] Oleg Cheiner and Alex A. Shvartsman. Implementation of an eventually serializable data service. In *Proceedings of the 17th Annual ACM Symposium on the Principles of Distributed Computing*, Puerta Vallarta, Mexico, June 1998. (Short paper).

[CS99] Oleg Cheiner and Alex Shvartsman. Implementing an eventually-serializable data service as a distributed system building block. In M. Mavronicolas, M. Merritt, and N. Shavit, editors, *Networks in Distributed Computing*, volume 45 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 43–72. American Mathematical Society, 1999.

[Dem97] Piotr Dembinski. Semantics of timed concurrent systems. *Fundamenta Informaticae*, 29:27–50, 1997. IOS Press.

[Dev99] M.C.A. Devillers. Translating IOA automata to PVS. Preliminary Research Report CSI-R9903, Computing Science Institute, University of Nijmegen, the Netherlands, feb 1999.

[DFLS98] Roberto DePrisco, Alan Fekete, Nancy Lynch, and Alex Shvartsman. A dynamic view-oriented group communication service. In *Proceedings of the 17th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 227–236, Puerto Vallarta, Mexico, June-July 1998. Also, technical memo in progress.

[DGRV00] M.C.A. Devillers, W.O.D. Griffioen, J.M.T. Romijn, and F.W. Vaandrager. Verification of a leader election protocol — formal methods applied to IEEE 1394. *Formal Methods in System Design*, 16(3):307–320, June 2000.

[dRE98] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[EGL92] Urban Engberg, Peter Grønning, and Leslie Lamport. Mechanical verification of concurrent systems with TLA. In G. V. Bochmann and D. K. Probst, editors, *Proceedings of the Fourth International Conference on Computer Aided Verification CAV'92*, volume 663 of *Lecture Notes in Computer Science*, pages 44–55, Berlin, June 1992. Springer-Verlag.

[FGL+96] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 300–309, Philadelphia, PA, May 1996.

[FGL+99] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data service. *Theoretical Computer Science*, 220(1):113–156, June 1999. Special Issue on Distributed Algorithms.

[FKL98] Alan Fekete, M. Frans Kaashoek, and Nancy Lynch. Implementing sequentially consistent shared objects using broadcast and point-to-point communication. *Journal of the ACM*, 45(1):35–69, January 1998.

[FLS97a] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 53–62, Santa Barbara, CA, August 1997. Expanded version in [FLS97b].

[FLS97b] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. Technical Memo MIT-LCS-TM-570, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 02139, 1997. Conference version in [FLS97a].

[For94] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications"*, 8(3/4), 1994.

[GG91] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Research Report 82, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1991.

[GHG+93] John V. Guttag, James J. Horning, Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing, editors. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag Texts and Monographs in Computer Science, 1993.

[GHW85] J.V. Guttag, J.J Horning, and J.M. Wing. The larch family of specification languages. *IEEE Software*, pages 24 –35, September 1985.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150–162. ACM, 1979.

[GL87] Kenneth J. Goldman and Nancy Lynch. Data replication in nested transaction systems. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, May 1987. Also, [Gol87].

[GL98] Stephen J. Garland and Nancy A. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1998. URL `http://theory.lcs.mit.edu/tds/papers/Lynch/IOA-TR-762.ps`.

[GLV00] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. IOA: A language for specifying, programming and validating distributed systems, December 2000. User and Reference Manual. `http://theory.lcs.mit.edu/tds/papers/Garland/ioaManual.ps.gz`.

[Gol87] Kenneth J. Goldman. Data replication in nested transaction systems. Technical Report MIT/LCS/TR-390, MIT Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1987.

[Gol90a] Kenneth J. Goldman. *Distributed Algorithm Simulation using Input/Output Automata.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 1990. Also,[Gol90b].

[Gol90b] Kenneth J. Goldman. Distributed algorithm simulation using input/output automata. Technical Report MIT/LCS/TR-490, MIT Laboratory for Computer Science, Cambridge, MA, September 1990. Also, PhD Thesis [Gol90a].

[Gol91] Kenneth J. Goldman. Highly concurrent logically synchronous multicast. *Distributed Computing*, 6(4):189–207, 1991.

[GSM+95] Kenneth J. Goldman, Bala Swaminathan, T. Paul McCartney, Michael D. Anderson, and Ram Sethuraman. The Programmers' Playground: I/O abstraction for user-configurable distributed applications. *IEEE Transactions on Software Engineering*, 21(9):735–746, September 1995.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hay96] Mark Hayden. *Ensemble Reference Manual*. Cornell University, 1996.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HLvR99] Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for Ensemble layers. In Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems, Fifth International Conference, (TACAS'99), Amsterdam, the Netherlands, March 1999*, volume 1579 of *Lecture Notes in Computer Science*, pages 119–133. Springer-Verlag, 1999.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, United Kingdom, 1985.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series, New Jersey, 1991.

[HSV94] L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. In H. Barendregt and T. Nipkow, editors, *Proceedings of the International Workshop on TYPES'93*, volume 806 of *Lecture Notes in Computer Science*, pages 127–165, Nijmegen, The Netherlands, 1994. Springer-Verlag. Full version available as Report CS-R9420, CWI, Amsterdam, March 1994.

[HvR96] M. Hayden and R. van Renesse. Optimizing layered communication protocols. Technical Report TR96-1613, Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA, November 1996.

[Int89] Information Processing Systems Open Systems Interconnection. ESTELLE — A formal description technique based on an extended state transition model, 1989.

[Kal95] S. Kalvala. A formulation of TLA in Isabelle. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971, pages 214–228, Aspen Grove, Utah, USA, 1995. Springer-Verlag.

[KK00] Idit Keidar and Roger Khazan. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. In *20thIEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 344–355, Taipei, Taiwan, April 2000. Full version: MIT Lab. for Computer Science Technical Report MIT-LCS-TR-794.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam83] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages Systems*, 5(2):190–222, April 1983.

[Lam93] Leslie Lamport. How to write a proof, December 1993. `http://www.research.compaq.com/SRC/personal/lamport/proofs/proofs.html`.

[Lam94a] Leslie Lamport. How to write a long formula, January 1994. `http://www.research.compaq.com/SRC/personal/lamport/proofs/proofs.html`.

[Lam94b] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam99] Leslie Lamport. *Specifying concurrent systems with TLA*, pages 183–247. Number 173 in Computer and Systems Sciences. IOS Press, 1999.

[Lan77] Gérard Le Lann. Distributed systems - towards a formal approach. In Bruce Gilchrist, editor, *Information Processing 77* (Toronto, August 1977), volume 7 of *Proceedings of IFIP Congress*, pages 155–160. North-Holland Publishing Co., 1977.

[LMWF94] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions.* Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[LSGL95] Victor Luchangco, Ekrem Söylemez, Stephen Garland, and Nancy Lynch. Verifying timing properties of concurrent algorithms. In Dieter Hogrefe and Stefan Leue, editors, *Formal Description Techniques VII: Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques* (FORTE'94, Berne, Switzerland, October 1994), pages 259–273. Chapman and Hall, 1995.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, Vancouver, British Columbia, Canada, August 1987.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, November 1988.

[Luc95] Victor Luchangco. Using simulation techniques to prove timing properties. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, June 1995.

[LV95] Nancy Lynch and Frits Vaandrager. Forward and backward simulations — Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[Lyn96] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.

[Mes92] José Meseguer. Conditional rewriting logic as an unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall International, United Kingdom, 1989.

[MN97] Olaf Müller and Tobias Nipkow. Traces of I/O automata in Isabelle/HOLCF. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 580–594. Springer, 1997.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[Mül98] Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory.* PhD thesis, Technishen Universität München, June 1998.

[NS95] Tobias Nipkow and Konrad Slind. I/O automata in Isabelle/HOL. In P. Dybjer, B. Nordström, and J. Smith, editors, *Proceedings of the International Workshop on Types for Proofs and Programs*, pages 101–119, Båstad, Sweden, 1995. Springer-Verlag LNCS 996.

[OCC84] INMOS Ltd: occam Programming Manual, 1984.

[OCC88] INMOS Ltd: occam 2 Reference Manual, 1988.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs, I. *Acta Informatica*, 6(4):319–340, 1976.

[Ost97] Jonathan S. Ostroff. A visual toolset for the design of real-time discrete event systems. *IEEE Transactions on Control Systems Technology*, 5(3), May 1997.

[Pau93] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1993.

[Pet82] Gary L. Peterson. An $O(n \log n)$ unidirectional distributed algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4(4):758–762, October 1982.

[PPG$^+$96] Tsvetomir P. Petrov, Anna Pogosyants, Stephen J. Garland, Victor Luchangco, and Nancy A. Lynch. Computer-assisted verification of an algorithm for concurrent timestamps. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX: Theory, Applications, and Tools* (FORTE/PSTV'96: Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification, Kaiserslautern, Germany, October 1996), pages 29–44. Chapman & Hall, 1996.

[RAH00] Elvinia Riccobene, Myla M. Archer, and Constance L. Heitmeyer. Applying TAME to I/O automata: A user's perspective. Technical Report NRL.MR.5540–00-8848, Naval Research Laboratory, April 2000.

[Rei99] Holly R. Reimers. An unparser for the IOA intermeiate language. Manuscript, August 1999.

[RGG$^+$95] A.W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS '95), volume 1019 of *Lecture Notes in Computer Science*, pages 133–152. Springer-Verlag, 1995.

[RR00] J. Antonio Ramırez-Robredo. Paired simulation of I/O automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 2000.

[SAGG$^+$93] Jørgen F. Søgaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-assisted simulation proofs. In Costas Courcoubetis, editor, *Computer-Aided Verification* (5th International Conference, CAV'93, Elounda, Greece, June/July 1993), volume 697 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 1993.

[Smi96] Mark Smith. Formal verification of communication protocols. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX: Theory, Applications, and Tools* FORTE/PSTV'96: Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification, Kaiserslautern, Germany, October 1996, pages 129–144. Chapman & Hall, 1996.

[Smi97] Mark Smith. *Formal Verification of TCP and T/TCP*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, September 1997.

[Smi98] Mark Smith. Reliable message delivery and conditionally-fast transactions are not possible without accurate clocks. In *Proceedings of the 17th Annual ACM Symposium on the Principles of Distributed Computing*, pages 163–171, June 1998.

[SOR93] N. Shankar, Sam Owre, and John Rushby. The PVS proof checker: A reference manual. Technical report, Computer Science Lab, SRI Intl., Menlo Park, CA, 1993.

[Söy94] Ekrem Söylemez. Automatic verification of the timing properties of MMT automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, February 1994.

[Tsa00] Michael J. Tsai. Design and implementation of the IOA GUI, interface generator, and NAD modules. Manuscript, January 2000.

[VF99] Mandana Vaziri-Farahani. A transformation of I/O automata removing implicit nondeterminism, 1999. Manuscript available from the author.

[VKC01] Roman Vitenberg, Idit Keidar, and Gregory V. Chockler. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 2001. To appear. Previous version: MIT Technical Report MIT-LCS-TR-790, September 1999.

[VTL] Mandana Vaziri, Joshua A. Tauber, and Nancy A. Lynch. A transformation of I/O automata removing implicit nondeterminism. Manuscript, Dec 12, 1998.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In Laurence Pierre and Thomas Kropf, editors, *Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, 1999.