Automated Implementation of Complex Distributed Algorithms Specified in the IOA Language*

Chryssis Georgiou
Dept. of Computer Science
University of Cyprus
Nicosia, Cyprus
chryssis@ucy.ac.cy

Nancy Lynch
MIT CSAIL
Cambridge, MA 02139
lynch@csail.mit.edu

Panayiotis Mavrommatis
MIT CSAIL
Cambridge, MA 02139
pmavrom@mit.edu

Joshua A. Tauber
MIT CSAIL
Cambridge, MA 02139
josh@csail.mit.edu

Abstract

IOA is a formal language for describing Input/Output automata that serves both as a formal specification language and as a programming language [14]. The IOA compiler automatically translates IOA specifications into Java code that runs on a set of workstations communicating via the Message Passing Interface. This paper describes the process of compiling IOA specifications and our experiences running several distributed algorithms. We focus on our implementation of the algorithm of Gallager, Humblet and Spira (GHS) for minimum-weight spanning tree formation in an arbitrary graph [13]. Our IOA code for GHS is derived from the Input/Output automaton description of the algorithm proved correct by Welch, Lamport, and Lynch [33].

The successful implementation of such a complicated algorithm is significant for two reasons: (a) it is an indication of the capabilities of the IOA compiler and of its advanced state of development, and (b) to the best of our knowledge, this is the first complex, distributed algorithm implemented in an automated way that has been formally and rigorously proved correct. Thus, this work shows that it is possible to formally specify, prove correct, and *implement complex distributed algorithms using a common formal methodology.*

1 Introduction

IOA is a formal language for describing distributed computation that serves both as a formal specification language and as a programming language [14]. The IOA toolkit supports the design, development, testing, and formal verification of programs based on the Input/Output automaton model of interacting state machines [23, 24]. I/O automata have been used to verify a wide variety of distributed systems and algorithms and to express and prove several impossibility results. The toolkit connects I/O automata with both lightweight (syntax checkers, simulators, model checkers [20, 6, 26, 10, 34, 32, 28]) and heavyweight (theorem provers [15, 3]) formal verification tools.

The IOA compiler has recently been added to the toolkit to enable programmers to write a specification in IOA, validate it using the toolkit, and then automatically translate the design into Java code. As a result, an algorithm specified in IOA can be implemented on a collection of workstations running Java Virtual Machines and communicating through the Message Passing Interface [29, 32, 31]. The code produced preserves the safety properties of the IOA program in the generated Java code. This guarantee is conditioned on the assumption that our model of network behavior is accurate, that a hand-coded datatype library correctly implements its semantic specification, and that programmer annotations yield specified values. We require a further technical constraint that the algorithm must be correct even when console inputs are delayed.

This paper describes our experiences compiling and running algorithms specified in IOA. We begin with a general description of the process of preparing and running any distributed algorithm. We then highlight important aspects of the process by describing our experiments with algorithms from the literature. Initially, we implemented LCR leader election in a ring, computation of a spanning tree in an arbitrary connected graph, and repeated broadcast/convergecast over a computed spanningtree [21, 4, 5, 27]. Our IOA code for these algorithms was derived from the I/O automaton description given for these algorithms in [22].

We focus primarily on our implementation of the algorithm of Gallager, Humblet and Spira (GHS) for finding the minimum-weight spanning tree in an arbitrary connected graph [13]. GHS is a sufficiently complicated algorithm to

*This work is supported in part by the IST grant 33116 (FLAGS), USAF, AFRL award #FA9550-04-1-0121 and MURI AFOSR award #SA2796PO 1-0000243658

constitute a "challenge problem" for the application of formal methods to distributed computing. Welch, Lamport, and Lynch formulated the algorithm using I/O automata and gave a formal proof of correctness of that specification [33]. Our IOA implementation of GHS is derived from the I/O automaton description by Welch, *et al.* by performing some technical modifications described in Section 5. The complex nature of the GHS algorithm revealed some technical implementation difficulties that we had to overcome in order to successfully implement the algorithm. These difficulties did not appear when dealing with simpler algorithms.

The successful implementation of such a complicated algorithm is significant for two reasons: (a) it indicates the capabilities of the IOA compiler and its advanced state of development, and (b) to the best of our knowledge, this is the first complex, distributed algorithm implemented in an automated way, that has been formally and rigorously proved correct. Thus, this work shows that it is possible to formally specify, prove correct *and* implement complex distributed algorithms using a common formal methodology.

2 Background

In this section, we briefly introduce the I/O automaton model and the IOA language and set the current work in the context of other research.

2.1 Input/Output Automata

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*); and a *transition relation*, consisting of a set of (state, action, state) triples (*transitions* specifying the effects of the automaton's actions).[1] An action π is *enabled* in state s if there is some triple (s, π, s') in the transition relation of the automaton. Input actions are required to be enabled in all states. 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. The I/O automaton model is inherently nondeterministic. In any given state of an automaton (or collection of automata), one, none, or many (possible infinitely many) actions may be enabled. As a result, there may be many valid executions of an automaton. A succinct explanation of the model appears in Chapter 8 of [22].

2.2 IOA Language

The *IOA language* [14] is a formal language for describing I/O automata and their properties. IOA code may be considered either a specification or a program. In either case, IOA yields precise, direct descriptions. States are represented by the values of variables rather than just by members of an unstructured set. IOA transitions are described in precondition-effect (or guarded-command) style, rather than as state-action-state triples. A precondition is a predicate on the the automaton state and the parameters of a transition that must hold whenever that transition executes. An effects clause specifies the result of a transition.

Due to its dual role, the language 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 post-state to the pre-state.

The language directly reflects the nondeterministic nature of the I/O automaton model. 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 a source of *implicit nondeterminism*. The `choose` operator provides *explicit nondeterminism* in selecting values from (possibly infinite) sets. These two types of nondeterminism are derived directly from the underlying model. The first reflects the fact that many actions may be enabled in any state. The second reflects the fact that a state-action pair (s, π) may not uniquely determine the following state s' in a transition relation.

2.3 Related Work

Goldman's Spectrum System introduced a formally-defined, purely operational programming language for describing I/O automata [17]. 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 [18].

Cheiner and Shvartsman experimented with methods for generating code by hand from I/O automaton descriptions [7]. They demonstrated their method by hand translating the Eventually Serializable Data Service of Luchangco *et al.* [11] into an executable, distributed implementation in C++ communicating via MPI. Unfortunately, their general implementation strategy uses costly reservation-based synchronization methods to avoid deadlock.

To our knowledge, no system has yet combined a language with formally specified semantics, automated proof assistants, simulators, and compilers. Several tools have been based on the CSP model [19]. The semantics of the Occam parallel computation language is defined in CSP [1]. While there are Occam compilers, we have found no evidence of verification tools for Occam programs. For-

[1] We omit discussion of *tasks*, which are sets of non-input actions.

mal Systems, Ltd., developed a machine-readable language for CSP.

Cleaveland *et al.* have developed a series of tools based on the CCS process algebra [25]. The Concurrency Workbench [9] and its successor the Concurrency Factory [8] 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.

3 Compiling and Running IOA

IOA can describe many systems architectures, including centralized designs, shared memory implementations, or message passing arrangements. Not every IOA specification may be compiled. An IOA program admissible for compilation must satisfy several constraints on its syntax, structure, and semantics. Programmers must perform two preprocessing steps before compilation. First, the programmer must combine the original "algorithm automaton" with several auxiliary automata. Second, the programmer must provide additional annotations to this combined program to resolve the nondeterminism inherent in the underlying I/O automaton denoted by the IOA program. The program can then be compiled into Java and thence into an executable. At runtime the user must provide information about the programs environment as well as the actual input to the program.

As proved elsewhere [29, 31], the system generated preserves the safety properties of the original IOA specification provided certain conditions are met. Those conditions are that the model of the MPI communication service behavior given in [29] is accurate, that the hand-coded datatype library used by the compiler correctly implements its semantic specification, and that programmer annotations correctly initialize the automaton.

3.1 Imperative IOA syntax

As mentioned in Section 2.2, IOA supports both operational and axiomatic descriptions of programming constructs. The IOA compiler translates only imperative IOA constructs. Therefore, IOA programs submitted for compilation cannot include certain IOA language constructs. Effects clauses cannot include **`ensuring`** clauses that relate pre-states to post-states declaratively. Throughout the program, predicates must be quantifier free. Currently, the compiler handles only restricted forms of loops that explicitly specify the set of values over which to iterate.

3.2 Node-channel form

The IOA compiler targets only message passing systems. The goal is to create a running system consisting of the compiled code and the existing MPI service that faithfully emulates the original distributed algorithm written in IOA. Each node in the target system runs a Java interpreter with its own console interface and communicates with other hosts via (a subset of) the Message Passing Interface (MPI) [12, 2]. (By "console" we mean any local source of input to the automaton. In particular, we call any input that Java treats as a data stream — other than the MPI connection — the console.)

The IOA compiler is able to preserve the externally visible behavior of the system without adding any synchronization overhead because we require the programmer to explicitly model the various sources of concurrency in the system: the multiple machines in the system and the communication channels. Thus, we require that systems submitted to the IOA compiler be described in *node-channel* form. The IOA programs to be compiled are the nodes. We call these programs *algorithm automata*. For the algorithm automaton `GHSProcess` for GHS, see [16].

All communication between nodes in the system uses asynchronous, reliable, one-way, FIFO channels. These channels are implemented by a combination of the underlying MPI communication service and *mediator automata* that are composed with the algorithm automata before compilation. Thus, algorithm automata may assume channels with very simple semantics and a very simple `SEND`/`RECEIVE` interface even though the underlying network implementation is more complex. In the distributed graph algorithms we implement, the network is the graph. That is, usually, nodes map to machines and edges to networks. (The exceptions are experiments in which we run multiple nodes on a single machine.)

3.3 Composition

The completed design is called the *composite node automaton* and is described as the composition of the algorithm automaton with its associated mediator automata. A *composer* tool [30] expands this composition into a new, equivalent IOA program in primitive where each piece of the automaton is explicitly instantiated. The resulting *node automaton* describes all computation to be be performed on one machine. This expanded node automaton (annotated as described below) is the final input program to the IOA compiler. The compiler translates each node automaton into its own Java program suitable to run on the target host.

3.4 Input-delay insensitivity

The I/O automaton model requires that input actions are always enabled. However, our Java implementation is not input enabled, it receives input only when the program asks for it by invoking a method. Therefore, each IOA system submitted for compilation must satisfy a semantic constraint. The system as a whole must behave correctly (as

defined by the programmer) even if inputs to any node from its local console are delayed. This is a technical constraint that most interesting distributed algorithms can be altered to meet.

3.5 Resolving Nondeterminism

Before compiling a node automaton, a programmer must resolve both the implicit nondeterminism inherent in any IOA program and any explicit nondeterminism introduced by **`choose`** statements. Execution of an automaton proceeds in a loop that selects an enabled transition to execute and then performing the effects of that transition. Picking a transition to execute includes picking a transition definition and the values of its parameters. It is possible and, in fact, common that the set of enabled actions in any state is infinite. In general, deciding membership in the set of enabled actions to be undecidable because transition preconditions may be arbitrary predicates in first-order logic. Thus, there is no simple and general search method for finding an enabled action. Even it when it is possible to find an enabled action, finding an action that makes actual progress may be difficult.

Therefore, before compilation, we require the programmer to write a schedule. A schedule is a function of the state of the local node that picks the next action to execute at that node. In format, a schedule is written at the IOA level in an auxiliary *nondeterminism resolution language* (NDR) consisting of imperative programming constructs similar to those used in IOA effects clauses. The NDR **`fire`** statement causes a transition to run and selects the values of its parameters. Schedules may reference, but not modify, automaton state variables. However, schedules may declare and modify additional variables local to the schedule [26, 10, 32].

3.6 Choosing

The **`choose`** statement introduces explicit nondeterminism in IOA. When a **`choose`** statement is executed, an IOA program selects an arbitrary value from a specified set. For example, the statement

```
num := choose n:Int where 0 <= n /\ n < 3
```

assigns either 0, 1, or 2 to `num`. As with finding parameterized transitions to schedule, finding values to satisfy the **`where`** predicates of **`choose`** statements is hard. So, again, we require the IOA programmer to resolve the nondeterminism. In this case, the programmer annotates the **`choose`** statement with an NDR *determinator block*. The **`yield`** statement specifies the value to resolve a nondeterministic choice. Determinator blocks may reference, but not modify, automaton state variables.

3.7 Initialization

The execution of an I/O automaton may start in any of a set of states. In an IOA program, there are two ways to denote its start states. First, each state variable may be assigned an initial value. That initial value may be a simple term or an explicit choice. In the latter case, the choice must be annotated with a choice determinator block to select the initial value before code generation. Second, the initial values of state variables may be collectively constrained by an **`initially`** clause. As with preconditions, an **`initially`** clause may be an arbitrary predicate in first order logic. Thus, there is no simple search method for finding an assignment of values to state variables to satisfy an **`initially`** clause. Therefore, we require the IOA programmer to annotate the **`initially`** predicate with an NDR determinator block. However, unlike NDR programs for automaton schedules **`initially`** determinator blocks may assign values directly to state variables. We omit the **`initially det`** block for GHS due to lack of space.

3.8 Runtime preparation

As mentioned above a system admissible for compilation must be described as a collection of nodes and channels. While each node in the system may run distinct code, often the nodes are symmetric. That is, each node in the system is identical up to parameterization and input. For example, the nodes in the GHS algorithm are distinguished only a unique integer parameter. Automaton parameters can also be used to give every node in the system some common information that is not known until runtime. For example, the precise topology of the network on which the system is running. If a compiled automaton is parameterized, the runtime system reads that information from a local file during initialization. In our testbed, certain special automaton parameters are automatically initialized at runtime. The `rank` of a node each node is a unique nonnegative integer provided by MPI. Similarly, the `size` of the system is the number of nodes connected by MPI. Input action invocations are also read from files (or file descriptors) at runtime. A description of the format for such invocations is given in [32].

4 Implementing Simple Algorithms

Our experimentation with the Toolkit began with the asynchronous version of the algorithm of Le Lann, Chang and Roberts (LCR) [21, 4] for leader election in a ring network. Each node sends its identifier around the ring. When a node receives an incoming identifier, it compares that identifier to its own. It propagates the identifier to its right neighbor only if the incoming identifier is greater than its own. The node that receives an incoming identifier equal to its own is elected as the leader. (In other words, only the

largest identifier completes a full circuit around the ring and the node that sent it is elected leader.) formal specification of the algorithm as an I/O automaton and a formal proof of its correctness can be found in [22] (Section 15.1.1). Our experimentation with LCR revealed some of the features of the Toolkit that were not yet implemented, such as initialization of formal parameters. After implementing these parts directly in Java, LCR became our first algorithm to be compiled from IOA and run successfully on a collection of workstations.

As soon as the Toolkit became fully automated, we worked on implementing an Asynchronous Spanning Tree algorithm for finding a rooted spanning tree in an arbitrary connected graph based on the work of Segal [27] and Chang [5]. Initially all nodes are "unmarked" except from a "source node" (the root of the resulting spanning tree). The source node sends a *search* message to its neighbors. When an unmarked node receives a search message, it marks itself and chooses the node from which the search message has arrived as its parent. Then it propagates the search message to its neighbors. If the node is already marked, it just propagates the message to its neighbors (in other words, a parent of a node i is the node from which i has received a search message for the *first* time). The spanning tree is formed by the edges between the parent nodes with their children. The successful implementation of this algorithm led to the implementation of an Asynchronous Broadcast/Convergecast algorithm, which is essentially an extension of the previous algorithm: Along with the construction of a spanning tree, a broadcast and convergecast takes place (the root node broadcasts a message down the tree and acknowledgments are passed up the tree from the leaves with each parent sending an acknowledgment up the tree only after receiving one from each of its children). We continued with two Leader Election algorithms on arbitrary connected graphs. The first one is an extension of the Asynchronous Broadcast/Convergecast algorithm, where each node performs its own broadcast to find out whether it is the leader (each node broadcasts its identifier, and it receives the identifiers of all other nodes – the one with the largest identifier is elected as the leader). The second one computes the leader based on a given spanning tree of the graph. Our code for each of these algorithms was based on the formal specification and a proof of correctness given in Chapter 15 of [22]. In each case, we were able, using the IOA compiler, to automatically produce an implementation of the algorithm in Java code and run it successfully on a network of workstations and run several experiments. For the details of the implementation including IOA code and runtime results of all these algorithms we refer the reader to [16].

5 Implementing the GHS Algorithm

The successful implementation of the (simple) algorithms above made us confident that it would be possible, using the Toolkit, to implement more complex distributed algorithms. Our algorithm of choice to test the Toolkit's capabilities was the seminal algorithm of Gallager, Humblet and Spira [13] for finding the minimum-weight spanning tree in an arbitrary connected graph with unique edge weights.

In the GHS algorithm, the nodes form themselves into components, which combine to form larger components. Initially each node forms a singleton component. Each component has a leader and a spanning tree that is a subgraph of the eventually formed minimum spanning tree. The identifier of the leader is used as the identifier of the component. Within each component, the nodes cooperatively compute the minimum-weight outgoing edge for the entire component. This is done as follows: The leader broadcasts search request along tree edges. Each node finds, among its incident edges, the one of minimum weight that is outgoing from the component (if any) and it reports it to the leader. The leader then determines the minimum-weight outgoing edge (which will be included in the minimum spanning tree) of the entire component and a message is sent out over that edge to the component on the other side. The two components combine into a new larger component and a procedure is carried out to elect the leader of the newly formed component. After enough combinations have occurred, all connected nodes in the given graph are included in a single connected component. The spanning tree of the final single component is the minimum spanning tree of the graph.

Welch, Lamport and Lynch [33] described the GHS algorithm using I/O automata and formally proved its correctness. We derived our IOA implementation of the algorithm (see [16]) from that description. Only technical modifications were necessary to convert the I/O automata description from [33] into IOA code recognizable by the IOA compiler. First, we introduced some variables that were not defined in the I/O automaton description as formal parameters of the automaton in the IOA code. For example, in our implementation, information about the edges of the graph is encoded in `links` and `weights` automaton parameters. In [33] that information is assumed to be available in a global variable. Second, the I/O automaton description uses the notion of a "procedure" to avoid code repetition. The IOA language does not support procedure calls with side-effects because call stacks and procedure parameters complicate many proofs. Thus, we had to write the body of the procedures several times in our code. Third, statements like "$let\ S = \langle p,r\rangle : lstatus(\langle p,r\rangle) = branch, r \neq q$" were converted into **`for`** loops that computed S.

The schedule block we used to run GHS can be found in [16]. In that block, each variable reference is qualified by the component automaton (`P`, `SM[*]`, or `RM[*]`) in which the variable appears. We also introduce new variables to track the progress of the schedule. The schedule block is structured as a loop that iterates over the neighbors of the

node. For each neighbor, the schedule checks if each action is enabled and, if so, fires it with appropriate parameterization. (The first six conditionals fire actions derived from the omitted mediator automata.) As formulated in [33], individual nodes do not know when the algorithm completes. Therefore, we terminated the algorithm manually after all nodes had output their status. The effect of the schedule is to select a legal execution of the automaton. When an action is fired at runtime, the precondition of the action is automatically checked. Thus, a schedule provides liveness but not safety.

Other than the schedule block, the changes necessary to derive compilable IOA code from the description in [33] can be described as syntactic. It follows that our IOA specification preserves the correctness of the GHS algorithm, as was formally proved in [33]. It follows from the correctness of the compiler as proved in [29] that the running implementation also preserves the safety properties proved by Welch, *et al.* provided certain conditions are met (see Section 3).

From our IOA specification, the compiler produced the Java code to implement the algorithm, enabling us to run the algorithm on a network of workstations. In particular we used up to 18 machines from the MIT Computer Science and Artificial Intelligence Laboratory local area network. The machine processors from 900MHz Pentium IIIs to 3GHz Pentium IVs, and all the machines were running Linux. The implementation was tested on a number of network topologies. Most often we used a grid, while sometimes we used an arbitrary connected graph. The number of nodes in the network ranged from 2 to 18. Tests were performed mostly with one node running on a single machine. In every experiment, the algorithm terminated and reported the minimum spanning tree correctly.

Several runtime measurements were made which can be summarized in Figure 1. The graph plots the execution time (left Y axis) and the total number of messages sent by all nodes (right Y axis) against the number of participating nodes. The theoretical runtime of the algorithm $O(n \log n)$ [22], is also shown. The actual runtime seems to correspond well with the theoretical one, and an important observation is that the execution time does not "explode" as the number of machines used increases, which gives some indication of the possible scalable nature of the implementation. We believe that the experimental results imply that the performance of the algorithm (mainly in terms of execution time) is "reasonable", considering that the implementation code was obtained by an automatic translation and not by an optimized, manual implementation of the original algorithm. Therefore, we have demonstrated that it is possible to obtain automated implementations (that perform reasonably well) of complex distributed algorithms (such as GHS) using the IOA toolkit.

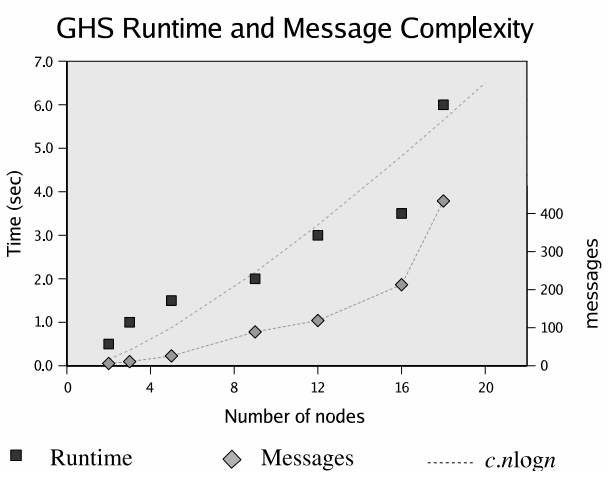

Figure 1. Performance of algorithm GHS

6 Conclusions

Direct compilation of formal models can enhance the application of formal methods to the development of distributed algorithms. Distributed systems specified as message-passing IOA programs can be automatically compiled when the programmer supplies annotations to resolve nondeterministic choices. As shown elsewhere, the resulting implementations are guaranteed to maintain the safety properties of the original program under reasonable assumptions. To the best of our knowledge, our implementation of GHS (using the IOA Toolkit) is the first example of a complex, distributed algorithm that has been formally specified, proved correct, and automatically implemented using a common formal methodology. Hence, this work has demonstrated that it is feasible to use formal methods, not only to specify and verify complex distributed algorithms, but also to automatically implement them (with reasonable performance) in a message passing environment.

References

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

[2] M. Baker, B. Carpenter, S. H. Ko, and X. Li. mpiJava: A Java interface to MPI. Submitted to First UK Workshop on Java for High Performance Network Computing, Europar 1998.

[3] A. Bogdanov. Formal verification of simulations between I/O automata. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, September 2001.

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

[5] E. J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, SE-8(4):391–401, July 1982.

[6] A. E. Chefter. A simulator for the IOA language. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, May 1998.

[7] O. Cheiner and A. 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. AMS, 1999.

[8] 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. AMS, 1994.

[9] 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.

[10] L. G. Dean. Improved simulation of Input/Output automata. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, September 2001.

[11] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. 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.

[12] M. P. I. Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[13] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. In *ACM Transactions on Programming Languages and Systems*, volume 5(1), pages 66–77, January 1983.

[14] S. Garland, N. Lynch, J. Tauber, and M. Vaziri. IOA user guide and reference manual. Technical Report MIT/LCS/TR-961, Laboratory for Computer Science, MIT, Cambridge, MA, July 2004. URL `http://theory.lcs.mit.edu/tds/ioa/manual.ps`.

[15] S. J. Garland and N. 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, MIT, Cambridge, MA, August 1998. URL `http://theory.lcs.mit.edu/tds/papers/Lynch/IOA-TR-762.ps`.

[16] C. Georgiou, P. Mavrommatis, and J. A. Tauber. Implementing asynchronous distributed systems using the IOA toolkit. Technical Report MIT/LCS/TR-966, Laboratory for Computer Science, MIT, Cambridge, MA, September 2004.

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

[18] K. J. Goldman, B. Swaminathan, T. P. McCartney, M. D. Anderson, and R. Sethuraman. The Programmers' Playground: I/O abstraction for user-configurable distributed applications. *IEEE Transactions on Software Engineering*, 21(9):735–746, September 1995.

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

[20] D. K. Kaynar, A. Chefter, L. Dean, S. Garland, N. Lynch, T. N. Win, and A. Ramırez-Robredo. The IOA simulator. Technical Report MIT-LCS-TR-843, MIT Laboratory for Computer Science, Cambridge, MA, July 2002.

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

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

[23] N. A. Lynch and M. 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.

[24] N. A. Lynch and M. 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, MIT, Cambridge, MA, November 1988.

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

[26] J. A. Ramırez-Robredo. Paired simulation of I/O automata. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, September 2000.

[27] A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(1):23–35, January 1983.

[28] E. Solovey. Simulation of composite I/O automata. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, September 2003.

[29] J. A. Tauber. *Verifiable Compilation of I/O Automata without Global Synchronization*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, September 2004.

[30] J. A. Tauber and S. J. Garland. Definition and expansion of composite automata in IOA. Technical Report MIT/LCS/TR-959, Laboratory for Computer Science, MIT, Cambridge, MA, July 2004. URL `http://theory.lcs.mit.edu/tds/papers/Tauber/MIT-LCS-TR-959.pdf`.

[31] J. A. Tauber, N. A. Lynch, and M. J. Tsai. Compiling IOA without global synchronization. In *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (IEEE NCA04)*, pages 121–130, Cambridge, MA, September 2004.

[32] M. J. Tsai. Code generation for the IOA language. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, June 2002.

[33] J. Welch, L. Lamport, and N. Lynch. A lattice-structured proof of a minimum spanning tree algorithm. In *Proceedings of 7th ACM Symposium on Principles of Distributed Computing*, pages 28–43, August 1988.

[34] T. N. Win. Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, May 2003.