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

Chryssis Georgiou[†]
Dept. of Computer Science
University of Cyprus
chryssis@cs.ucy.ac.cy

Nancy Lynch
MIT CSAIL
lynch@csail.mit.edu

Panayiotis Mavrommatis
Google
mavrommatis@gmail.com

Joshua A. Tauber
MIT CSAIL
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 [13]. 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, ranging from simple ones such as the Le Lann, Chang and Roberts (LCR) leader election in a ring algorithm to that of Gallager, Humblet and Spira (GHS) for minimum-weight spanning tree formation in an arbitrary graph [29]. Our IOA code for all the algorithms is derived from their Input/Output automaton descriptions that have already been formally proved correct.

The successful implementation of these algorithms 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, these are the first complex, distributed algorithms implemented in an automated way that have 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.

Keywords: Input/Output Automata, Automated Code Generator, Verifiable Distributed Code, IOA Toolkit, Formal Methods.

Contact Author: Chryssis Georgiou, `chryssis@cs.ucy.ac.cy`

1 Introduction

IOA is a formal language for describing distributed computation that serves both as a formal specification language and as a programming language [13]. The IOA toolkit supports the design, development, testing, and formal verification of programs based on the Input/Output automaton model of interacting state machines [26, 27]. 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 [22, 6, 30, 10, 39, 36, 32]) and heavyweight (theorem provers [14, 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

[*]This work is supported in part by USAF, AFRL award #FA9550-04-1-0121 and MURI AFOSR award #SA2796PO 1-0000243658. A preliminary version of this paper has appeared in [16].

[†]The work of this author is supported in part by research funds at the University of Cyprus.

Machines and communicating through the Message Passing Interface [33, 36, 35]. The code produced preserves the safety properties of the IOA program in the generated Java code. This guarantee is conditioned on the assumptions 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 an overview of the challenges that were addressed in realizing the code generator and then we provide a general description of the process of preparing and running any distributed algorithm. We next 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 spanning tree [23, 4, 5, 31]. Our IOA code for these algorithms was derived from the I/O automaton description given for these algorithms in [24].

Finally, we have successfully implemented the algorithm of Gallager, Humblet and Spira (GHS) for finding the minimum-weight spanning tree in an arbitrary connected graph [29]. GHS is a sufficiently complicated algorithm to 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 [38]. 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 8. In the process of realizing this task (implementing these algorithms) several complier implementation difficulties arose and were successfully addressed.

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.

Paper Structure. In Section 2 we overview the challenges faced in realizing the IOA code generator. In Section 3 we provide necessary background on Input/Output automata and present related work. In Section 4 we present the compilation procedure and necessary technical issues. Sections 5–8 describe the implemented algorithms and provide IOA code used for their automated implementation. In Section 9 we present experimental results we obtained from the implementation of our algorithms. We conclude in Section 10.

2 Challenges in Realizing the IOA Compiler

The design and implementation of the IOA compiler required overcoming a number of key challenges. Many of these challenges arise from the differences between characteristics of specifications that are easiest to prove correct and characteristics of programs that are easiest to run. In this section, we overview these challenges and the approach taken to addressing them. *Note that several of the presented implementation difficulties arose (and were addressed) while attempting to use the code generator in implementing the presented distributed algorithms.* For a deeper analysis of the challenges and a thorough presentation of the actions taken in addressing them we refer the reader to Tauber's Thesis [33].

2.1 Program Structuring

The first major challenge needed to be addressed was how to create a system with the correct externally visible behavior of the system without using any synchronization between processes running on different machines. This goal was achieved by matching the formal specification of the distributed system to the target architecture of running systems. That is, the form of the IOA programs admissible for compilation had to be restricted. In particular, the programmer is required (rather than the compiler) to decide on the distribution of computation; the programs submitted for compilation must be structured in a *node-channel* form that reflects the message-passing architecture of the collection of networked workstations that is the target of the compiler. Compilation then proceeds on a node-by-node basis (see Section 4.2 for specific details). By requiring the programmer to match the system design to the target language, hardware, and system services before attempting to compile the program, we are able to generate *verifiably correct code* without any synchronization between processes running on different machines.

2.2 IOA Programs and External Services

IOA programs use external services such as communication networks and console inputs. The IOA compiler generates only the specialized code necessary to implement an algorithm at each node in the system. At runtime, each node connects to the external system services it uses. In the current version the compilation target is a system in which each host runs a Java interpreter and communicates via a small subset of the Message Passing Interface (MPI) [12]. Hence a second major challenge needed to be addressed was to create both correctness proofs about algorithms that connect to such services and to produce correct code that uses such external, preexisting services. The general approach we take for verifying access to an external service consists of four phases: First model the external service as an IOA automaton; for example a subset of MPI was modeled as an automaton (see [33, Chapter 4]). Second, identify the desired abstract service programmers would like to use and specify that abstract service as an IOA automaton; e.g., a specification of an abstract, point-to-point, reliable, FIFO channel was developed (see [33]). Third, write mediator automata such that the composition of the mediator automata and the external service automaton implements the abstract service automaton; for example see the SendMediator and ReceiveMediator automata used in this work (in Appendix A). Fourth, prove that implementation relationship. (Specific details on the MPI-IOA connectivity are given in Section 4.2.)

2.3 Modeling Procedure Calls

The above design for connecting to system services raises new challenges. One particularly tricky aspect of such proofs is modeling the interfaces to services correctly. IOA itself has no notion of procedure call per se. The Java interface to an external service is defined in terms of method invocations (procedure calls). The approach taken to address this issue, when modeling these services, was to carefully treat method invocations and method returns as distinct behaviors of the external service; when procedure calls may block, handshake protocols were developed to model such blocking.

2.4 Composing Automata

The auxiliary mediator automata created to implement abstract system services must be combined with the source automaton prior to compilation. That is, we need to compose these automata to form a single automaton that describes all the computation local to a single node in the system. (Specific details are given in Section 4). A tool, called *composer*, was designed and implemented to compose automata automatically (see [33, Part II]).

2.5 Nondeterminism

The IOA language is inherently nondeterministic. Translating programs written in IOA into an imperative language like Java requires resolving all nondeterministic choices. This process of resolving choices is called *scheduling* an automaton. Developing a method to schedule automata was the largest conceptual challenge in the initial design of an IOA compiler. In general, it is computationally infeasible to schedule IOA programs automatically. Instead, IOA was augmented with nondeterminism resolution (NDR) constructs that allow programmers to schedule automata directly and safely. (More details are given in Section 4.5).

2.6 Implementing Datatypes

Datatypes used in IOA programs are described formally by axiomatic descriptions in first-order logic. While such specifications provide sound bases for proofs, it is not easy to translate them automatically into an imperative language such as Java. However, the IOA framework focuses on correctness of the concurrent, interactive aspects of programs rather than of the sequential aspects. Therefore we were not especially concerned with establishing the correctness of datatype implementations. (Standard techniques of sequential program verification may be applied to attempt such correctness proofs.) Hence, each IOA datatype is implemented by a hand-coded Java class. A library of such classes for the standard IOA datatypes is included in the compiler. Each IOA datatype (e.g., `Tree[]`) and operator (e.g., `Tree[]` $\rightarrow$ `Nat`) is matched with its Java implementation class using a *datatype registry* [30, 36, 39], which we extended in this work (by creating mainly new operators).

3 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.

3.1 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 non-deterministic. In any given state of an automaton (or collection of automata), one, none, or many (possibly 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 [24].

3.2 IOA Language

The *IOA language* [13] 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

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

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 non-deterministic 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 non-determinism*. The **choose** operator provides *explicit non-determinism* in selecting values from (possibly infinite) sets. These two types of non-determinism 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.

3.3 Related Work

Goldman's Spectrum System introduced a formally-defined, purely operational programming language for describing I/O automata [18]. 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 [19].

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 [20]. 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. Formal Systems, Ltd., developed a machine-readable language for CSP.

The CCS process algebra [28] and I/O automata frameworks share several similarities as well as many differences. The work in [37] presents a semantic-based comparison of the two frameworks; the work in [15] evaluates and compares the applicability and usability of a value-passing version of CCS with I/O automata on specifying and verifying distributed algorithms. Cleaveland *et al.* have developed a series of tools based on the CCS process algebra. The Concurrency Workbench [9] and its successor the Concurrency Factory (CF) [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. To the best of our knowledge, no complex distributed algorithm such as GHS has been specified, verified and implemented (in an automated way) using the "CCS-CF-Facile" framework.

4 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 non-determinism 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 [33, 35], 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 [33] 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.

4.1 Imperative IOA Syntax

As mentioned in Section 3.2, 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.

4.2 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.) We note that we use only four of the (myriad) methods provided by MPI:

Isend sends a message to a specified destination and returns a handle to name the particular send.

test tests a handle to see if the particular send has completed.

Iprobe polls to see if an incoming message is available.

recv returns a message when available.

As already mentioned in Section 2, 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*.

As discussed before, 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. The `recvMediator` automaton (Appendix A) mediates between the algorithm automaton and an incoming channel, while the `sendMediator` automaton (Appendix A) handles messages to outgoing channels. Each of the n node programs connects to up to $2n$ mediator automata (one for each of its channels). Figure 1 illustrates how a mediator automaton is composed with MPI to create a reliable, FIFO channel.

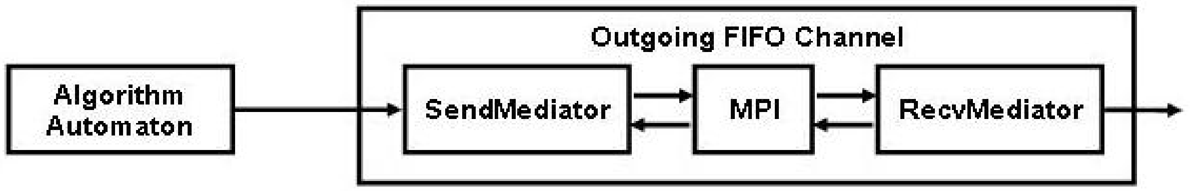

Figure 1: Auxiliary automata mediate between MPI and algorithm automata to yield a reliable FIFO channel.

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.)

4.3 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 [34] expands this composition into a new, equivalent IOA program in primitive form where each piece of the automaton is explicitly instantiated. The resulting *node automaton* describes all computation to 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.

The node automaton combining the `GHSProcess` and standard mediator automata is shown in the Appendix. In that automaton, the SEND and RECEIVE actions are **hidden** so that interfaces between algorithm and mediator automata are not externally visible.

4.4 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. For this purpose, our code generator deploys input buffers; a producer/consumer style is used where the act of consuming inputs from the buffer falsifies the precondition of the consumer so that the consuming transition never tries to read an empty buffer. The producer can always safely append new inputs to the buffer but there is no guarantee as to when the input will be consumed.

4.5 Resolving Non-determinism

As mentioned in Section 2.5, before compiling a node automaton, a programmer must resolve both the implicit non-determinism inherent in any IOA program and any explicit non-determinism introduced by **choose** statements.

4.5.1 Scheduling

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 is 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 *non-determinism 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 [30, 10, 36].

Conceptually, adding an NDR schedule to an IOA program changes it in three ways. The NDR schedule adds new variables, modifies each transition to use the new variables, and provides a computable *next-action* function of the augmented state. The new state variables consist of a program counter and whatever variables the programmer uses in the NDR schedule program. Each locally controlled action is modified in two ways. First, the precondition is strengthened so that the action is enabled only if the program counter names the action. Second, at the end of the effects the program counter is assigned the next action as computed by applying the next-action function to the automaton state. The schedule annotation used to run GHS is included in Appendix C.2.

4.5.2 Choosing

The **choose** statement introduces explicit non-determinism 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 parametrized 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 non-determinism. In this case, the programmer annotates the **choose** statement with an NDR *determinator block*. The **yield** statement specifies the value to resolve a non-deterministic choice. Determinator blocks may reference, but not modify, automaton state variables.

4.5.3 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. The **initially det** block for GHS is included in Appendix C.2.

4.6 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 parametrization and input. For example, the nodes in the GHS algorithm are distinguished only by 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 parametrized, 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 is a unique non-negative 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 [36].

Figure 2 provides a graphical outline of the compilation procedure.

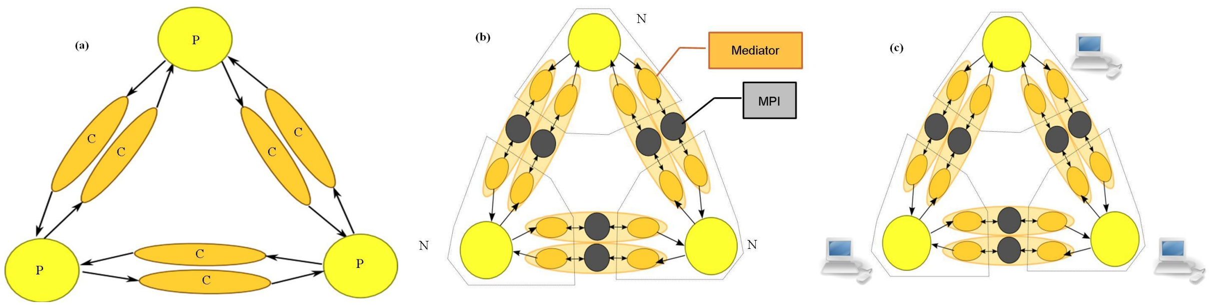

Figure 2: Given an Input/Output automaton specification of a distributed algorithm: (a) Write the *process automaton* in IOA; one automaton per node, connected by simple send and receive interfaces to channels.(b) Compose the process automaton with the *mediator automata* (standard IOA library programs that provide simple FIFO-channel semantics to the algorithm) to form the *composite node automaton*. Using the composer tool, expand the composite node automaton to obtain the *node automaton*. Annotate the node automaton with a non-determinism resolving schedule block, to produce the *scheduled node automaton*. (c) Compile, using the IOA compiler, the scheduled node automaton to a Java class. By compiling the Java class we obtain an executable program in which communication is performed via MPI.

5 Implementing LCR Leader Election

The first algorithm to be automatically compiled with the IOA Compiler was the asynchronous version of the algorithm of Le Lann, Chang and Roberts (LCR) [23, 4] for leader election in a ring network. In LCR, 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 clockwise 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. Informally, it can be seen that only the largest identifier completes a full circuit around the ring and the node that sent it is elected leader. A formal specification of the algorithm as an I/O automaton and a proof of its correctness can be found in [24] (Section 15.1.1).

LCR Leader Election process automaton

```
type Status = enumeration of idle, voting,
              elected, announced

automaton LCRProcess(rank: Int, size: Int)
signature
  input vote
```

```
input RECEIVE(m: Int, const mod(rank - 1,
              size), const rank: Int)
output SEND(m: Int, const rank: Int,
            const mod(rank+1, size))
output leader(const rank)
```

```
states
  pending: Mset[Int] := {rank},
  status: Status := idle
transitions
  input vote
    eff status := voting
  input RECEIVE(m, j, i) where m > i
    eff pending := insert(m, pending)
  input RECEIVE(m, j, i) where m < i
input RECEIVE(i, j, i)
  eff status := elected
output SEND(m, i, j)
  pre status ≠ idle ∧ m ∈ pending
  eff pending := delete(m, pending)
output leader(rank)
  pre status = elected
  eff status := announced
```

The automaton definition that appears in [24](Section 15.1) was used, with some minor modifications. (The **const** keyword means that the transition parameter is going to have the same value every time the transition is fired.) For all the algorithms that follow, the nodes are automatically numbered from 0 to (size - 1). The automata `LCRProcess`, `LCRNode`, `SendMediator` and `ReceiveMediator` were written. The mediator automata implement the channel automata integrated with MPI functionality, and can be found in Appendix A. The `LCRNode` automaton, included in Appendix B.1 simply composes the mediator automata with the process automaton. This automaton was automatically expanded and a schedule was written for the composition, which appears in Appendix B.2. The implementation was tested on a number of different configurations, and ran correctly in all cases.

6 Implementing Spanning Tree and Broadcast/Convergecast

6.1 Asynchronous Spanning Tree

The next algorithm we implemented was an Asynchronous Spanning Tree algorithm for finding a rooted spanning tree in an arbitrary connected graph based on the work of Segal [31] and Chang [5]. This was the first test of the Toolkit on arbitrary graphs, where each node had more than one incoming and outgoing communication channels. In this algorithm all nodes are initially "unmarked" except for 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. It then 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 AsynchSpanningTree automaton, as defined in [24] (Section 15.3) was used. The process automaton is listed below[2]. The schedule for the composition of this automaton with the mediator ones appears in Appendix B.3.

Asynchronous spanning tree process automaton

```
type Message = enumeration of search, null
automaton sTreeProcess(i: Int, nbrs: Set[Int])
signature
  input RECEIVE(m: Message,
                const i: Int, j: Int)
  output SEND(m: Message,
              const i: Int, j: Int)
  output PARENT(j: Int)
states
  parent: Null[Int] := nil,
  reported: Bool := false,
  send: Map[Int, Message] := empty
transitions
  input RECEIVE(m, i, j)
    eff
      if i ≠ 0 ∧ parent = nil then
         parent := embed(j);
         for k: Int in nbrs - {j} do
           send[k] := search
         od
      fi
  output SEND(m, i, j)
    pre send[j] = search
    eff send[j] := null
  output PARENT(j)
    pre parent.val = j ∧ ¬reported
    eff reported := true
```

[2]For an explanation of the constructs appearing in the the code (such as `Null[]` or `embed()`) we refer the reader to [13, Appendix A.11]

6.2 Asynchronous Broadcast Convergecast

The successful implementation of the spanning tree 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). A formal specification and a proof of correctness is given in [24](Section 15.3). In our tests, the root was node 0, and the value $v1$ (a dummy value) was broadcast on the network. The process automaton is shown below. The schedule for the composition of this automaton with the mediator ones appears in Appendix B.4.

Broadcast-convergecast process automaton

```
type Kind = enumeration of bcast, ack
type Val = enumeration of null, v1
type BCastMsg = tuple of kind: Kind, v: Val
type Message = union of msg: BCastMsg,
                        kind: Kind

automaton bcastProcess(rank: Int,
                       nbrs: Set[Int])
signature
  input RECEIVE(m: Message, const rank,
                j: Int)
  output SEND(m: Message, const rank,
              j: Int)
  internal report(const rank)
states
  val: Val := null,
  parent: Null[Int] := nil,
  reported: Bool := false,
  acked: Set[Int] := {},
  send: Map[Int, Seq[Message]]
initially
  rank = 0 ⇒
    (val = v1 ∧
      (∀ j: Int j ∈ nbrs ⇒
         send[j] = {msg([bcast, val])} ))
transitions
  output SEND(m, rank, j)
    pre m = head(send[j])
    eff send[j] := tail(send[j])
  input RECEIVE(m, rank, j)
    eff
      if m = kind(ack) then
        acked := acked ∪ {j}
      else
        if val = null then
          val := m.msg.v;
          parent := embed(j);
          for k:Int in nbrs - {j} do
            send[k] := send[k] ⊢ m
          od
        else
          send[j] := send[j] ⊢ kind(ack)
        fi
      fi
  internal report(rank) where rank = 0
    pre acked = nbrs;
        reported = false
    eff reported := true
  internal report(rank) where rank ≠ 0
    pre parent ≠ nil;
        acked = nbrs - {parent.val};
        reported = false
    eff send[parent.val] :=
          send[parent.val] ⊢ kind(ack);
        reported := true;
```

7 Implementing General Leader Election Algorithms

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 [24]. 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.

7.1 Leader Election Using Broadcast Convergecast

The main idea of the Leader Election Using Broadcast Convergecast algorithm [24, page 500] is to have every node act as a source node and create its own spanning tree, broadcast its UID using this spanning tree and hear from all the other nodes via a convergecast. During this convergecast, along with the acknowledge message, the children also send what they consider as the maximum UID in the network. The parents gather the maximum UIDs from the children, compare it to their own UID and send the maximum to their own parents. Thus, each source node learns the maximum UID in the network and the node whose UID equals the maximum one announces itself as a leader. The process automaton is given below, and the schedule for its composition with the mediator automata in Appendix B.5. The implementation was tested on various logical network topologies, terminating correctly every time.

Leader Election with Broadcast-convergecast process automaton

```
type Kind = enumeration of bcast, ack
type Val = enumeration of null, v1
type BCastMsg = tuple of kind: Kind, v: Val
type AckMsg = tuple of kind:Kind, mx: Int
type MSG = union of bmsg: BCastMsg, amsg:
           AckMsg, kind: Kind
type Message = tuple of msg: MSG, source: Int

automaton bcastLeaderProcess(rank: Int, size: Int)
signature
  input RECEIVE(m: Message, i: Int, j: Int)
  output SEND(m: Message, i: Int, j: Int)
  internal report(i: Int, source: Int)
  internal finished
  output LEADER
states
  nbrs: Set[Int],
  val: Map[Int, Int],
  parent: Map[Int, Null[Int]],
  reported: Map[Int, Bool],
  acked: Map[Int, Set[Int]],
  send: Map[Int, Int, Seq[Message]],
  max: Map[Int, Int],
  elected: Bool := false,
  announced: Bool := false
initially
  val[j] = rank ∧
  (∀ j: Int
    ((0 ≤ j ∧ j < size) ⇒
      rank ≠ j ⇒ val[j] = nil ∧
      parent[j] = -1 ∧
      acked[j] = {} ∧
      max[j] = rank ∧
      (∀ k:Int
        ((0 ≤ k ∧ k < size) ⇒
          send[j,k] = {}) ∧
        (k ∈ nbrs ∧ rank = j) ⇒
          send[j,k] =
            {[bmsg([bcast, v1]), j]})))))
transitions
 output SEND(m, i, j)
   pre m = head(send[m.source, j])
   eff send[m.source, j] :=
         tail(send[m.source, j])
input RECEIVE(m, i, j)
  eff
    if m.msg = kind(ack) then
      acked[m.source] := acked[m.source]
                         ∪ {j}
    elseif tag(m.msg) = amsg then
      if max[m.source] < m.msg.amsg.mx then
        max[m.source] := m.msg.amsg.mx;
      fi;
      acked[m.source] := acked[m.source]
                         ∪ {j}
    else %BcastMsg
      if val[m.source] = -1 then
        val[m.source] := m.msg.bmsg.w;
        parent[m.source] := j;
        for k:Int in nbrs - {j} do
          send[m.source, k] :=
            send[m.source, k] ⊢ m
        od
      else
        send[m.source,j] := send[m.source,j]
           ⊢ [kind(ack), m.source]
      fi
    fi
internal finished
  pre acked[rank] = nbrs ∧ ¬reported[rank]
  eff reported[rank] := true;
      if (max[rank] = rank) then
        elected := true
      fi;
output LEADER
  pre elected ∧ ¬announced
  eff announced := true
internal report(i, source) where i ≠ source
  pre parent[source] ≠ -1 ∧
      acked[source] = nbrs - {parent[source]} ∧
      ¬reported[source]
  eff send[source, parent[source]] :=
        send[source, parent[source]]
        ⊢ [amsg([ack, max[source]]), source];
      reported[source] := true;
```

7.2 Unrooted Spanning Tree to Leader Election

The algorithm **STtoLeader** of [24](page 501) was implemented as the next test for the Toolkit. The algorithm takes as input an unrooted spanning tree and returns a leader. The automaton listed below was written, according to the description of the algorithm in [24]. The schedule for its composition with the mediator automata appears in Appendix B.6.

Unrooted Spanning Tree to Leader Election process automaton

```
type Status = enumeration of idle, elected,
              announced
type Message = enumeration of elect

automaton sTreeLeaderProcess(rank: Int,
                              nbrs:Set[Int])
signature
  input RECEIVE(m: Message, const
               rank: Int, j: Int)
  output SEND(m: Message, const
              rank: Int, j: Int)
  output leader
states
  receivedElect: Set[Int] := {},
  sentElect: Set[Int] := {},
  status: Status := idle,
  send: Map[Int, Seq[Message]]
initially
  size(nbrs) = 1 ⇒
    send[chooseRandom(nbrs)] = {elect}
transitions
  input RECEIVE(m, i, j; local t: Int)
  eff
   receivedElect := receivedElect ∪ {j};
   if size(receivedElect) =
        size(nbrs)-1 then
      t := chooseRandom(nbrs -
                        receivedElect);
      send[t] := send[t] ⊢ elect;
      sentElect := sentElect ∪ {t};
   elseif receivedElect = nbrs then
     if j ∈ sentElect then
       if i > j then status := elected fi
     else
       status := elected
     fi
   fi
output SEND(m, i, j)
  pre m = head(send[j])
  eff send[j] := tail(send[j])
output leader
  pre status = elected
  eff status := announced
```

8 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 [29] 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 a 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 [38] described the GHS algorithm using I/O automata and formally proved its correctness. We derived our IOA implementation of the algorithm from that description. Our IOA code of the GHS automaton (due its length) is given in Appendix C.1. Only technical modifications were necessary to convert the I/O automata description from [38] 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 [38] 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 Appendix C.2. 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 parametrization. As formulated in [38], 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.

Other than the schedule block, the changes necessary to derive compilable IOA code from the description in [38] can be described as syntactic. It follows that our IOA specification preserves the correctness of the GHS algorithm, as was formally proved in [38]. It follows from the correctness of the compiler as proved in [33] that the running implementation also preserves the safety properties proved by Welch *et al.* provided certain conditions are met (see Section 4).

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 every experiment, the algorithm terminated and reported the minimum spanning tree correctly.

9 Performance

We have run our algorithms' implementations (described in Sections 5–8) to demonstrate the functionality of the generated code, measure some of its basic properties, and make some observations about the compilation process. Measuring the performance (runtime and message complexity) of the running algorithms establishes some quantitative benchmarks for comparing the current version of the compiler to future optimizations or any alternative implementations[3].

Our experiments exercise many features of the compiler. First and foremost, we show that distributed message-passing algorithms run and pass messages as expected. In doing so, we employ most of the catalog of IOA datatypes, datatype constructors, and record types. The basic datatypes are booleans, integers, natural numbers, characters, and strings. The type constructors are arrays, sets, multisets, sequences, and mappings. The record types are enumerations, tuples, and unions. Of these, we use all but naturals, characters, and strings. In addition, we introduce new and enhanced datatypes not in the basic IOA language. For example, we enhance the `Set` and `Mset` datatype constructors with choice operators and introduce a `Null` type constructor. We demonstrate the use of all the supported control structures in the language including loops that iterate over the elements of finite sets.

For our experiments we used up to 24 machines from the MIT CSAIL Theory of Computation local area network. The machine processors ranged from 900MHz Pentium IIIs to 3GHz Pentium IVs, and all the machines were running Linux, Redhat 9.0 to Fedora Core 2. We note that the network connectivity was not optimized at all (the machines were essentially people's desktops, interconnected via lots of routers and switches throughout a building). The implementations were tested on a number of logical network topologies. All the tests we report here were performed with each node running on a different machine.

[3]Provided that the same platform and network configuration is used.

9.1 Performance of Simple Algorithm Implementations

Figure 3 displays the runtime performance of our automated implementations for the algorithms LCR, Broadcast/Convergecast, Spanning Tree to Leader Election and Broadcast to Leader Election. The runtime values are averaged over 10 runs of the algorithm.

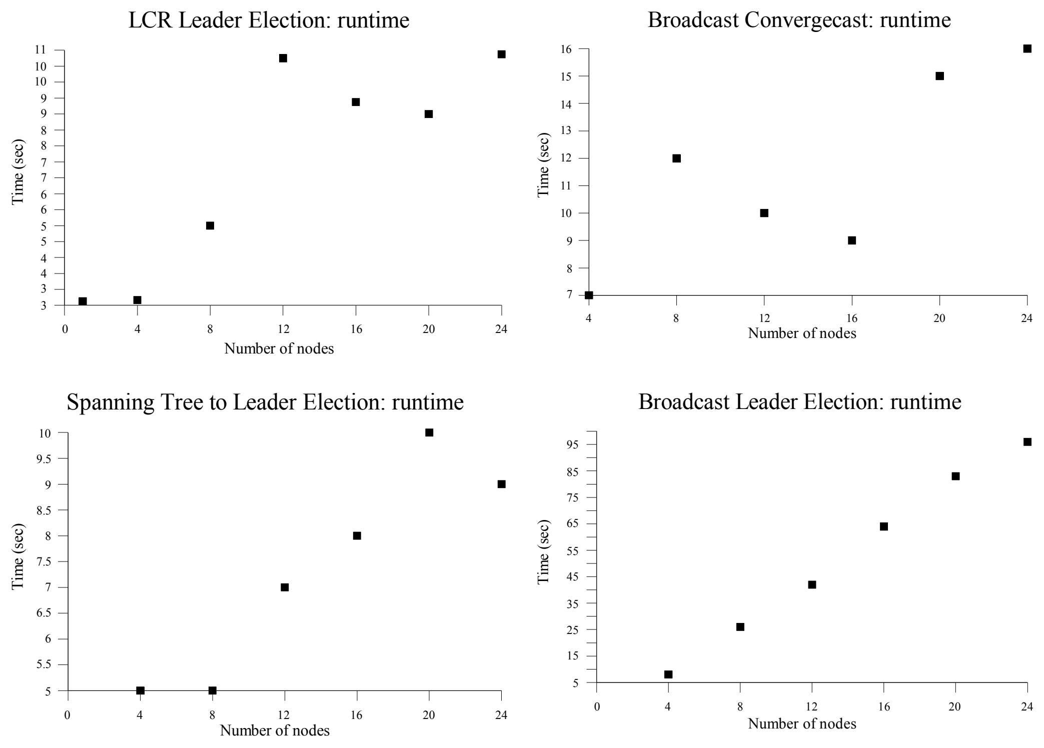

Figure 3: Runtime Performance of the automated implementations of LCR, Broadcast/convergecast, Spanning Tree to Leader Election and Broadcast to Leader Election algorithms.

LCR Runtime The theoretical message complexity of LCR depends on the order of the node identifiers in the ring, and ranges from $O(n)$ to $O(n^2)$, where n is the number of nodes in the network. In all our configurations, the node identifiers were ordered in the most optimal way (in increasing order), thus around $2n$ messages were exchanged. The first n messages can be sent simultaneously, while the last n messages must be linearized. These n linearized messages, where nodes receive the message of the largest node and forward it to their clockwise neighbor, result in a linear runtime for the algorithm overall, because message delay is much larger than local computation. We therefore expect LCR to perform linearly with the number of nodes in these optimal configurations. As Figure 3 indicates, with the exception of a "spike" at 12 nodes, the experimental runtime tends to be linear[4].

Broadcast/Convergecast Runtime The theoretical time complexity for the asynchronous broadcast/convergecast algorithm is $O(n)$ [24]. Our experimental results (Figure 3) once again agree with the theoretical complexity (not considering the spikes).

[4]The spikes are caused by the very random underlying physical topology of the nodes in different cluster sizes, and because the number of messages exchanged was small, the runtime had high deviation. In the case of BLE, the huge number of messages reduced the deviation, and hence no spikes appear, as it can be observed in Figure 3.

Spanning Tree to Leader Election Runtime The time complexity for the leader election algorithm with a given spanning tree is once again $O(n)$ [24]. As Figure 3 indicates, the experimental runtime agrees with $O(n)$ (not considering the spikes).

Broadcast Leader Election Runtime The leader election algorithm that uses simultaneous broadcast/-convergecast should also run within $O(n)$ time, however the message complexity is much larger. The experimental results of Figure 3 agree with the time complexity. The absolute values of the running times, however were much larger compared to the previous algorithms. This is expected since a much larger number of messages are exchanged (on the order of n^2).

9.2 Performance of GHS Implementation

Several runtime measurements were made which can be summarized in Figure 4. The graphs plot 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 $c \cdot n \log n$ [24], is also shown (for $c = 0.25$). 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. The theoretical upper bound on the number of messages is $5n \log n + 2|E|$, and is also plotted in the right graph. As expected, the actual number of messages exchanged was on average lower than this upper bound.

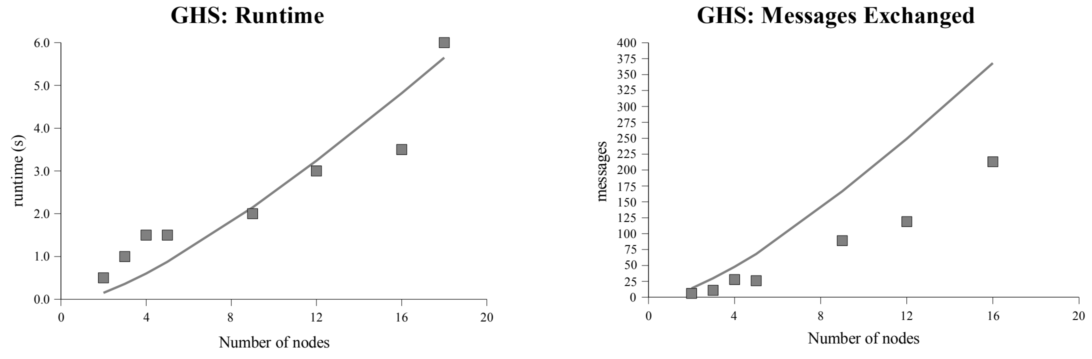

Figure 4: Performance of the automated GHS implementation. The theoretical complexities are also plotted.

We believe that the experimental results imply that the performance of the implementation (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 (and the network connectivity was not optimized – factors like DNS resolution, for example, add a lot of latency to the connections). 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.

9.3 Observations

Programming algorithms from the literature with IOA was generally a smooth process. Writing schedules was both easier and harder than expected: For the algorithms in our case studies, schedules followed fairly predictable patterns. The arrival of a message or an input action triggers the execution of a cascade of transitions. The schedules for our case studies essentially loop over these possible sources of input and when an input arrives the schedule performs the entire resulting cascade of transitions before checking for the next input. Thus, the basic structure of a schedule turned out to be very easy to outline. On the other hand, our experience was that most programming time was spent debugging NDR schedules. In this regard, runtime checks on NDR generated values (e.g., precondition checks) proved valuable. Unfortunately, livelock was an all too frequent result of a buggy schedule. Writing the conditional guards for **fire** statements was particularly tricky when polling for items from queues. In particular, it was a frequent bug that a schedule never executed any transitions.

Finally it is worth mentioning that the time required for MPI to set up all the connections and enable nodes to initialize was not measured in the runtime results. However, when the number of nodes was large, the time was also quite significant (around 5-10 minutes). A possible explanation for this is the following: MPI sets up a connection between all pairs of nodes, even if these connections are not necessary. For example, an n-node LCR needs only n connections, while MPI sets up $\Theta(n^2)$ connections. Perhaps another communication interface, which gives more control over these issues (e.g., Java RMI or Java Sockets with TCP) could be used instead of MPI. As discussed in the next section, ongoing work is heading toward this direction.

10 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 non-deterministic 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.

There are several future research directions that emanate from the presented work. One direction is to investigating whether the automated translation can be optimized to improve efficiency. As mentioned in the previous section, one can use our experiments as benchmarks for comparison with future versions of the code generator. Furthermore, an important research exercise worth pursuing is to compare the performance of our automated algorithm implementations with the ones obtained using a different methodology; for example, specify and verify algorithm GHS using the CCS process algebra and then use the Concurrency Factory approach [8] together with facile to obtain experimental data for comparison.

Recall that the code produced from the IOA code generator preserves the *safety* properties of the IOA specification. An important topic for future investigation is to enable the code generator to also provide some kind of *liveness* guarantees. The preservation of liveness properties depends on the NDR schedules written to resolve non-determinism. Unfortunately no formal semantics have been given for NDR. Therefore, in order to make any claims about the liveness of the generated code, one would first need a formal model for NDR and then investigate how NDR would preserve the liveness properties proved for the IOA specification (before it is fed into the compiler).

Another future research direction is to enable the automated implementation of IOA-specified algorithms on WANs with dynamic node participation. Currently the compiler is limited to static participation and use in LANs due to the use of MPI for communication. As explained in Section 2.2, the compiler design is general enough to enable the use of other communication paradigms. In [17] an alternative communication paradigm is suggested (Java Sockets with TCP) that enables the automated implementation of algorithms that have dynamic participation (nodes may join and leave the computation at any time). Ongoing work is attempting to incorporate this alternative paradigm into the IOA compiler.

The TIOA language (an extension of the IOA language) models distributed systems with timing constraints as collections of interacting state machines, called Timed Input/Output Automata (an extension of Input/Output Automata) [21]. A TIOA toolkit is underway [25] which (so far) includes a TIOA syntax and type checker, a TIOA simulator (with limited functionality), a model checker and a theorem prover. A very challenging research direction is to develop a TIOA code generator, as several issues need to be addressed in order for one to be able to incorporate time into the IOA complier.

Acknowledgments. We thank the anonymous reviewers for their constructive comments that helped us to significantly improve the presentation of this paper.

References

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

[2] 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.

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

[4] 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.

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

[6] 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, May 1998.

[7] 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.

[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. American Mathematical Society, 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] Laura G. Dean. Improved simulation of Input/Output automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 2001.

[11] 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.

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

[13] Stephen Garland, Nancy Lynch, Joshua Tauber, and Mandana Vaziri. IOA user guide and reference manual. Technical Report MIT/LCS/TR-961, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 2004. URL `http://theory.lcs.mit.edu/tds/ioa/manual.ps`.

[14] 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`.

[15] Marina Gelastou, Chryssis Georgiou, and Anna Philippou. On the application of formal methods for specifying and verifying distributed protocols. In *Proceedings of the 7th IEEE International Symposium on Network Computing and Applications (NCA 2008)*, pages 195–204, 2008.

[16] Chryssis Georgiou, Nancy Lynch, Panayiotis Mavrommatis, and Joshua A. Tauber. Automated implementation of complex distributed algorithms specified in the IOA language. In *Proceedings of 18th International Conference on Parallel and Distributed Computing Systems (PDCS 2005)*, pages 128–134, 2005.

[17] Chryssis Georgiou, Peter M. Musial, Alexander A. Shvartsman, and Elaine L. Sonderegger. An abstract channel specification and an algorithm implementing it using java sockets. In *Proceedings of the 7th IEEE International Symposium on Network Computing and Applications (NCA 2008)*, pages 211–219, 2008.

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

[19] 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.

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

[21] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata*. Morgan & Claypool Publishers, 2006.

[22] Dilsun Kırlı Kaynar, Anna Chefter, Laura Dean, Stephen Garland, Nancy Lynch, Toh Ne Win, and Antonio Ramırez-Robredo. The IOA simulator. Technical Report MIT-LCS-TR-843, MIT Laboratory for Computer Science, Cambridge, MA, July 2002.

[23] 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.

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

[25] Nancy Lynch, Laurent Michel, and Alexander Shvartsman. Tempo: A toolkit for the timed input/output automata formalism. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks, and Systems (SIMUTools 2008) – Industrial Track: Simulation Works*, 2008.

[26] 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.

[27] 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, November 1988.

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

[29] P. A. Humblet R. G. Gallager 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.

[30] 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.

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

[32] Edward Solovey. Simulation of composite I/O automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 2003.

[33] Joshua A. Tauber. *Verifiable Compilation of I/O Automata without Global Synchronization*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 2004.

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

[35] Joshua A. Tauber, Nancy A. Lynch, and Michael 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.

[36] Michael J. Tsai. Code generation for the IOA language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, June 2002.

[37] Frits W. Vaandrager. On the relationship between process algebra and input/output automata. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science (LICS 1991)*, pages 387–398, 1991.

[38] Lampoft L. Welch J. and Lynch N. 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.

[39] Toh Ne Win. Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 2003.

APPENDIX

A Mediator automata

SendMediator Automaton

```
type sCall = enumeration of idle, Isend, test
automaton SendMediator(Msg, Node:Type, i:Node, j:Node)
  assumes Infinite(Handle)
  signature
    input  SEND(m: Msg, const i, const j)
    output Isend(m: Msg, const i, const j)
    input  resp_Isend(handle:Handle, const i, const j)
    output test(handle:Handle, const i, const j)
    input  resp_test(flag:Bool, const i, const j)
  states
    status: sCall := idle,
    toSend: Seq[Msg] := {},
    sent: Seq[Msg] := {},
    handles: Seq[Handle] := {}
  transitions
    input SEND(m, i, j)
      eff toSend := toSend ⊢ m
    output Isend(m,i,j)
      pre head(toSend) = m;
          status = idle
      eff toSend := tail(toSend);
          sent := sent ⊢ m;
          status := Isend
    input resp_Isend(handle, i, j)
      eff handles := handles ⊢ handle;
          status := idle
    output test(handle, i, j)
      pre status = idle;
          handle = head(handles)
      eff status := test
    input resp_test(flag, i, j)
      eff if (flag = true) then
             handles := tail(handles);
             sent := tail(sent)
          fi;
          status := idle
```

ReceiveMediator Automaton

```
type rCall = enumeration of idle, receive, Iprobe
automaton ReceiveMediator(Msg, Node: Type,
     i: Node, j:Node)
  assumes Infinite(Handle)
  signature
    output RECEIVE(m:Msg, const i, const j)
    output Iprobe(const i, const j)
    input resp_Iprobe(flag:Bool, const i, const j)
    output receive(const i, const j)
    input resp_receive(m: Msg, const i, const j)
  states
    status: rCall := idle,
    toRecv: Seq[Msg] := {},
    ready: Bool := false
  transitions
    output RECEIVE(m, i, j)
      pre m = head(toRecv)
      eff toRecv := tail(toRecv)
    output Iprobe(i, j)
      pre status = idle;
          ready = false
      eff status := Iprobe
    input resp_Iprobe(flag, i, j)
      eff ready := flag;
          status := idle
    output receive(i, j)
      pre ready = true;
          status = idle
      eff status := receive
    input resp_receive(m, i, j)
      eff toRecv := toRecv ⊢ m;
          ready := false;
          status := idle
```

B Schedule blocks

B.1 Composition automaton for LCR Leader Election

```
automaton LCRNode(rank: Int, size: Int)
  components
    P: LCRProcess(rank, size);
    RM[j:Int]: ReceiveMediator(Int, Int, j, rank)
                  where j = mod(rank-1, size);
    SM[j:Int]: SendMediator(Int, Int, rank, j)
                  where j = mod(rank+1, size)
```

B.2 Schedule block for LCR Leader Election

```
schedule
states
  left : Int := mod((rank+size) -1,size),
  right: Int := mod(rank+1,size)
do
  fire input vote;
  while (true) do
    if P.pending ≠ {} then
      fire output SEND(
        chooseRandom(P.pending),
        rank, right)
    fi;
    if SM[right].status = idle ∧
       SM[right].toSend ≠ {} then
      fire output Isend(
        head(SM[right].toSend),rank,right)
    fi;
    if SM[right].status = idle ∧
       SM[right].handles ≠ {} then
      fire output test(
        head(SM[right].handles),rank,right)
    fi;
    if RM[left].status = idle ∧
       ¬RM[left].ready then
      fire output Iprobe(rank, left) fi;
    if RM[left].status = idle ∧
       RM[left].ready then
      fire output receive(rank, left) fi;
    if RM[left].toRecv ≠ {} then
      fire output RECEIVE(
        head(RM[left].toRecv), left, rank)
    fi;
    if P.status = elected then
      fire output leader(rank)
    fi
  od
od
```

B.3 Schedule block for Spanning Tree formation

```
schedule
states
  nb: Set[Int],
  k: Int
do
  while (true) do
    nb := nbrs;
    while (¬isEmpty(nb)) do
      k := chooseRandom(nb);
      nb := delete(k, nb);
      if P.send[k] = search then
        fire output SEND(search, rank, k)
      fi;
      if SM[k].status = idle ∧
         SM[k].toSend ≠ {} then
        fire output Isend(
          head(SM[k].toSend), rank, k)
      fi;
      if SM[k].status = idle ∧
         SM[k].handles ≠ {} then
        fire output test(
          head(SM[k].handles), rank, k)
      fi;
      if RM[k].status = idle ∧
         RM[k].ready = false then
        fire output Iprobe(rank, k)
      fi;
      if RM[k].status = idle ∧
         RM[k].ready = true then
        fire output receive(rank, k)
      fi;
      if RM[k].toRecv ≠ {} then
        fire output RECEIVE(
          head(RM[k].toRecv), rank, k)
      fi;
      if P.parent = k ∧ ¬P.reported then
        fire output PARENT(k)
      fi
    od
  od
od
```

B.4 Schedule block for Broadcast/Convergecast

```
schedule
states
  tempNbrs: Set[Int],
  k: Int
do
  while(true) do
    tempNbrs := nbrs;
    while (¬isEmpty(tempNbrs)) do
      k := chooseRandom(tempNbrs);
      tempNbrs := delete(k, tempNbrs);
      if P.send[k] ≠ {} then
        fire output SEND(
          head(P.send[k]), rank, k) fi;
      if SM[k].status = idle ∧
         SM[k].toSend ≠ {} then
        fire output Isend(
          head(SM[k].toSend), rank, k) fi;
      if SM[k].status = idle ∧
         SM[k].handles ≠ {} then
        fire output test(
          head(SM[k].handles), rank, k) fi;
      if RM[k].status = idle ∧
         ¬RM[k].ready then
        fire output Iprobe(rank, k) fi;
      if RM[k].status = idle ∧
         RM[k].ready then
        fire output receive(rank, k) fi;
      if RM[k].toRecv ≠ {} then
        fire output RECEIVE(
          head(RM[k].toRecv), rank, k) fi
    od;
    if rank = 0 ∧ P.acked = nbrs ∧
       ¬P.reported then
      fire internal report(rank) fi;
    if rank ≠ 0 ∧ P.parent ≠ nil ∧
       P.acked = nbrs - {P.parent.val} ∧
       ¬P.reported then
      fire internal report(rank) fi
  od
od
```

B.5 Schedule block for Leader Election with Broadcast/Convergecast

```
schedule
states
  c: Int, % source
  tempNbrs: Set[Int],
  k: Int
do
  while(true) do
    c := size;
    while (c > 0) do
      c := c - 1;
      tempNbrs := nbrs;
      while (¬isEmpty(tempNbrs)) do
        k := chooseRandom(tempNbrs);
        tempNbrs := delete(k, tempNbrs);
        if P.send[c, k] ≠ {} then
          fire output SEND(
            head(P.send[c, k]), rank, k) fi;
        if SM[k].status = idle ∧
           SM[k].toSend ≠ {} then
          fire output Isend(
            head(SM[k].toSend), rank, k) fi;
        if SM[k].status = idle ∧
           SM[k].handles ≠ {} then
          fire output test(
            head(SM[k].handles), rank, k) fi;
        if RM[k].status = idle ∧
           ¬RM[k].ready then
          fire output Iprobe(rank, k) fi;
        if RM[k].status = idle ∧
           RM[k].ready then
          fire output receive(rank, k) fi;
        if RM[k].toRecv ≠ {} then
          fire output RECEIVE(
          head(RM[k].toRecv), rank, k) fi
      od;
      if c ≠ rank ∧ P.parent[c] ≠ -1 ∧
         P.acked[c] = nbrs - {P.parent[c]} ∧
         ¬P.reported[c]  then
        fire internal report(rank, c) fi;
      if c = rank ∧  P.acked[rank] = nbrs ∧
         ¬P.reported[rank] then
        fire internal finished fi;
      if P.elected ∧ ¬P.announced  then
        fire output LEADER
      fi
    od
  od
od
```

B.6 Schedule block for Spanning Tree to Leader Election

```
schedule
states
  tempNbrs: Set[Int],
  k: Int
do
  while(true) do
    tempNbrs := nbrs;
    while (¬isEmpty(tempNbrs)) do
      k := chooseRandom(tempNbrs);
      tempNbrs := delete(k, tempNbrs);
      if P.send[k] ≠ {} then
        fire output SEND(
          head(P.send[k]), rank, k) fi;
      if SM[k].status = idle ∧
         SM[k].toSend ≠ {} then
        fire output Isend(
          head(SM[k].toSend), rank, k) fi;
      if SM[k].status = idle ∧
         SM[k].handles ≠ {} then
        fire output test(
         head(SM[k].handles), rank, k) fi;
      if RM[k].status = idle ∧
         ¬RM[k].ready then
        fire output Iprobe(rank, k) fi;
      if RM[k].status = idle ∧
         RM[k].ready then
        fire output receive(rank, k) fi;
      if RM[k].toRecv ≠ {} then
        fire output RECEIVE(
         head(RM[k].toRecv), rank, k) fi
    od;
    if P.status = elected then
      fire output leader
    fi
  od
od
```

C GHS IOA Code

C.1 GHS algorithm automaton

```
type Nstatus = enumeration of sleeping, find, found
type Edge = tuple of s: Int, t: Int
type Link = tuple of s: Int, t: Int
type Lstatus = enumeration of unknown, branch,
        rejected
type Msg = enumeration of CONNECT, INITIATE, TEST,
                          REPORT, ACCEPT, REJECT,
                          CHANGEROOT
type ConnMsg = tuple of msg: Msg, l: Int
type Status = enumeration of find, found
type InitMsg = tuple of msg: Msg, l: Int,
                        c: Null[Edge], st: Status
type TestMsg = tuple of msg: Msg, l: Int,
                        c: Null[Edge]
type ReportMsg = tuple of msg: Msg, w: Int
type Message = union of connMsg: ConnMsg,
                        initMsg: InitMsg,
                        testMsg: TestMsg,
                        reportMsg: ReportMsg,
                        msg: Msg


%%
%  automaton GHSProcess: Process of GHS Algorithm
%                        for min. spanning tree
%  rank: The UID of the automaton
%  size: The number of nodes in the network
%  links: Set of Links with source = rank (L_p(G))
%  weight: Maps the Links ∈ links to their weight
%%
automaton GHSProcess(rank: Int, size: Int,
       links: Set[Link],
       weight: Map[Link, Int])
 signature
  input startP
  input RECEIVE(m: Message, const rank, i: Int)
  output InTree(l:Link)
  output NotInTree(l: Link)
  output SEND(m: Message, const rank, j: Int)
  internal ReceiveConnect(qp: Link, l:Int)
  internal ReceiveInitiate(qp: Link, l:Int,
            c: Null[Edge], st: Status)
  internal ReceiveTest(qp: Link, l:Int,
            c: Null[Edge])
  internal ReceiveAccept(qp: Link)
  internal ReceiveReject(qp: Link)
  internal ReceiveReport(qp: Link, w: Int)
  internal ReceiveChangeRoot(qp: Link)

 states
  nstatus: Nstatus,
  nfrag: Null[Edge],
  nlevel: Int,
  bestlink: Null[Link],
  bestwt: Int,
  testlink: Null[Link],
  inbranch: Link,
  findcount: Int,
  lstatus: Map[Link, Lstatus],
  queueOut: Map[Link, Seq[Message]],
  queueIn: Map[Link, Seq[Message]],
  answered: Map[Link, Bool]
 initially
       nstatus = sleeping
       ∧ nfrag = nil
       ∧ nlevel = 0
       ∧ bestlink.val ∈ links
       ∧ bestwt = weight[bestlink.val]
       ∧ testlink = nil
       ∧ inbranch = bestlink.val
       ∧ findcount = 0
       ∧ ∀ l: Link
         (l ∈ links ⇒
           lstatus[l] = unknown
           ∧ answered[l] = false
           ∧ queueOut[l] = {}
           ∧ queueIn[l] = {})
 transitions
  input startP(local minL: Null[Link], min: Int)
   eff if nstatus = sleeping then
        %WakeUp
        minL := choose l where l.val ∈ links;
        min := weight[minL.val];
        for tempL:Link in links do
         if weight[tempL] < min then
          minL := embed(tempL);
          min := weight[tempL]  fi;
        od;
        lstatus[minL.val] := branch;
        nstatus := found;
        queueOut[minL.val] := queueOut[minL.val]
                              ⊢ connMsg([CONNECT, 0]);  fi
  input RECEIVE(m: Message, i:Int, j:Int)
   eff queueIn[[i,j]] := queueIn[[i,j]] ⊢ m
  output InTree(l: Link)
   pre answered[l] = false ∧ lstatus[l] = branch
   eff answered[l] := true
  output NotInTree(l: Link)
   pre answered[l] = false ∧ lstatus[l] = rejected
   eff answered[l] := true
  output SEND(m: Message, i: Int, j: Int)
   pre m = head(queueOut[[i,j]])
   eff queueOut[[i,j]] := tail(queueOut[[i,j]])
  internal ReceiveConnect(qp: Link, l: Int;
                          local minL: Null[Link], min: Int)
   pre head(queueIn[qp]) = connMsg([CONNECT, l])
   eff queueIn[qp] := tail(queueIn[qp]);
       if nstatus = sleeping then
       %WakeUp
        minL := choose l where l.val ∈ links;
        min := weight[minL.val];
        for tempL:Link in links do
         if weight[tempL] < min then
          minL := embed(tempL);
          min := weight[tempL] fi
        od;
        lstatus[minL.val] := branch;
        nstatus := found;
        queueOut[minL.val] := queueOut[minL.val]
                              ⊢ connMsg([CONNECT, 0]);  fi
       if l < nlevel then
        lstatus[[qp.t,qp.s]] := branch;
        if testlink ≠ nil then
         queueOut[[qp.t,qp.s]] :=
         queueOut[[qp.t,qp.s]] ⊢
         initMsg([INITIATE,nlevel, nfrag, find]);
         findcount := findcount + 1
        else queueOut[[qp.t,qp.s]] := queueOut[[qp.t,qp.s]] ⊢
              initMsg([INITIATE,nlevel, nfrag, found]) fi;
       else if lstatus[[qp.t,qp.s]] = unknown then
         queueIn[qp] := queueIn[qp] ⊢ connMsg([CONNECT, l]
       else queueOut[[qp.t,qp.s]] := queueOut[[qp.t,qp.s]] ⊢
             initMsg([INITIATE, nlevel+1,
          embed([qp.t, qp.s]), find]) fi  fi
  internal ReceiveInitiate(qp: Link, l:Int, c: Null[Edge], st: Status;
            local minL: Null[Link], min: Int, S : Set[Link])
   pre head(queueIn[qp])=initMsg([INITIATE,l,c,st])
   eff queueIn[qp] := tail(queueIn[qp]);
       nlevel := l;
       nfrag := c;
       if st = find then nstatus := find
       else nstatus := found fi;
       %Let S = {[p,q]:lstatus[[p,r]]=branch,r≠q}
       S := {};
       for pr: Link in links do
        if pr.t ≠ qp.s ∧ lstatus[pr] = branch then
           S := S ∪ {pr}
        fi
       od;
       for k: Link in S do
        queueOut[k] := queueOut[k] ⊢
        initMsg([INITIATE, l, c, st])
       od;
       if st = find then
        inbranch := [qp.t, qp.s];
        bestlink := nil;
        bestwt := 10000000; % Infinity
        %Test
        minL := nil; min := 10000000; % Infinity
        for tempL:Link in links do
         if weight[tempL] < min ∧
           lstatus[tempL] = unknown then
           minL := embed(tempL);
         min := weight[tempL]  fi;
        od;
        if minL ≠ nil then
         testlink := minL;
        queueOut[minL.val] := queueOut[minL.val]
                              ⊢ testMsg([TEST, nlevel, nfrag]);
       else testlink := nil;
        if findcount = 0 ∧ testlink = nil then
          nstatus := found;
          queueOut[inbranch] :=
                 queueOut[inbranch] ⊢
                 reportMsg([REPORT, bestwt])  fi  fi;
       %EndTest
       findcount := size(S) fi
 internal ReceiveTest(qp: Link, l: Int, c: Null[Edge];
                      local minL: Null[Link], min: Int)
  pre head(queueIn[qp]) = testMsg([TEST, l, c])
  eff queueIn[qp] := tail(queueIn[qp]);
     if nstatus = sleeping then
      %WakeUp
      minL := choose l where l.val ∈ links;
      min :=  weight[minL.val];
      for tempL:Link in links do
       if weight[tempL] < min then
        minL := embed(tempL); min:= weight[tempL] fi;
      od;
      lstatus[minL.val] := branch;
      nstatus := found;
      queueOut[minL.val] := queueOut[minL.val]
                            ⊢ connMsg([CONNECT, 0]); fi;
     if l > nlevel then
      queueIn[qp] := queueIn[qp] ⊢ testMsg([TEST, l, c]);
     else if c ≠ nfrag then
      queueOut[[qp.t, qp.s]] :=
                            queueOut[[qp.t, qp.s]] ⊢ msg(ACCEPT)
     else if lstatus[[qp.t, qp.s]] = unknown then
       lstatus[[qp.t, qp.s]] := rejected fi;
       if testlink ≠ embed([qp.t, qp.s]) then
        queueOut[[qp.t, qp.s]] :=
                            queueOut[[qp.t, qp.s]] ⊢ msg(REJECT)
       else %Test
       minL := nil;
       min := 10000000; % Infinity
       for tempL:Link in links do
        if weight[tempL] < min ∧ lstatus[tempL] = unknown then
         minL := embed(tempL);
         min := weight[tempL] fi;
       od;
       if minL ≠ nil then
        testlink := minL;
        queueOut[minL.val] := queueOut[minL.val]
               ⊢ testMsg([TEST, nlevel, nfrag]);
       else testlink := nil;
        if findcount = 0 ∧ testlink = nil then
         nstatus := found;
         queueOut[inbranch] := queueOut[inbranch]
                               ⊢ reportMsg([REPORT, bestwt])
        fi  fi; fi; fi; fi;
internal ReceiveAccept(qp: Link)
 pre head(queueIn[qp]) = msg(ACCEPT)
 eff queueIn[qp] := tail(queueIn[qp]);
     testlink := nil;
     if weight[[qp.t, qp.s]] < bestwt then
      bestlink := embed([qp.t, qp.s]);
      bestwt := weight[[qp.t, qp.s]]; fi;
     if findcount = 0 ∧ testlink = nil then
      nstatus := found;
      queueOut[inbranch] := queueOut[inbranch]
                            ⊢ reportMsg([REPORT, bestwt]) fi
internal ReceiveReject(qp: Link;
          local minL: Null[Link], min: Int)
 pre head(queueIn[qp]) = msg(REJECT)
 eff queueIn[qp] := tail(queueIn[qp]);
     if lstatus[[qp.t, qp.s]] = unknown then
      lstatus[[qp.t, qp.s]] := rejected  fi;
     %Test
     minL := nil; min := 10000000; % Infinity
     for tempL:Link in links do
      if weight[tempL] < min ∧
       lstatus[tempL] = unknown then
        minL := embed(tempL); min:= weight[tempL] fi;
     od;
     if minL ≠ nil then
      testlink := minL;
      queueOut[minL.val] := queueOut[minL.val]
                           ⊢ testMsg([TEST, nlevel, nfrag]);
     else testlink := nil;
      if findcount = 0 ∧ testlink = nil then
       nstatus := found;
       queueOut[inbranch] := queueOut[inbranch] ⊢
                            reportMsg([REPORT, bestwt]) fi  fi
internal ReceiveReport(qp: Link, w: Int)
 pre head(queueIn[qp]) = reportMsg([REPORT, w])
 eff queueIn[qp] := tail(queueIn[qp]);
     if [qp.t, qp.s] ≠ inbranch then
       findcount := findcount -1;
       if w < bestwt then
        bestwt := w;
        bestlink := embed([qp.t, qp.s]) fi;
       if findcount = 0 ∧ testlink = nil then
        nstatus := found;
        queueOut[inbranch] := queueOut[inbranch]
                              ⊢ reportMsg([REPORT, bestwt]) fi
     else if nstatus = find then
       queueIn[qp] := queueIn[qp] ⊢  reportMsg([REPORT, w])
      else if w > bestwt then  %ChangeRoot
       if lstatus[bestlink.val] = branch then
        queueOut[bestlink.val] := queueOut[bestlink.val] ⊢
                                  msg(CHANGEROOT)
       else queueOut[bestlink.val] :=  queueOut[bestlink.val] ⊢
                                       connMsg([CONNECT, nlevel]) ;
        lstatus[bestlink.val] := branch fi fi fi
internal ReceiveChangeRoot(qp: Link)
 pre head(queueIn[qp]) = msg(CHANGEROOT)
 eff queueIn[qp] := tail(queueIn[qp]);
     %ChangeRoot
     if lstatus[bestlink.val] = branch then
      queueOut[bestlink.val] := queueOut[bestlink.val] ⊢
msg(CHANGEROOT)
     else queueOut[bestlink.val] :=  queueOut[bestlink.val] ⊢
                                     connMsg([CONNECT, nlevel]) ;
      lstatus[bestlink.val] := branch  fi
```

C.2 Schedule and initialization block for GHS

```
states
  ...
  det do
    P.nstatus := sleeping;
    P.nfrag := nil;
    P.nlevel := 0;
    P.bestlink := embed(chooseRandom(links));
    P.bestwt := weight[chooseRandom(links)];
    P.testlink := nil;
    P.inbranch := chooseRandom(links);
    P.findcount := 0;
    tempLinks := links;
    while (¬isEmpty(tempLinks)) do
      tempL := chooseRandom(tempLinks);
      tempLinks := delete(tempL, tempLinks);
      P.lstatus[tempL] := unknown;
      P.answered[tempL] := false;
      P.queueOut[tempL] := {};
      P.queueIn[[tempL.t, tempL.s]] := {};
      RM[tempL.t] := [idle, {}, false];
      SM[tempL.t] := [idle, {}, {}, {}]
    od
  od
...
schedule
states
  lnks: Set[Link],
  lnk : Link
do
  fire input startP;
  while(true) do
    lnks := links;
    while (¬isEmpty(lnks)) do
      lnk := chooseRandom(lnks);
      lnks := delete(lnk, lnks);
      if P.queueOut[lnk] ≠ {} then
        fire internal SEND(head(P.queueOut[lnk]),
          rank, lnk.t) fi;
      if SM[lnk.t].status = idle ∧
         SM[lnk.t].toSend ≠ {} then
        fire output Isend(head(SM[lnk.t].toSend),
          rank, lnk.t) fi;
      if SM[lnk.t].status = idle ∧
         SM[lnk.t].handles ≠ {} then
            fire output test(head(SM[lnk.t].handles),
         rank, lnk.t) fi;
      if RM[lnk.t].status = idle ∧
         RM[lnk.t].ready = false then
        fire output Iprobe(rank, lnk.t) fi;
      if RM[lnk.t].status = idle ∧
         RM[lnk.t].ready = true then
        fire output receive(rank, lnk.t) fi;
      if RM[lnk.t].toRecv ≠ {} then
        fire internal RECEIVE(
          head(RM[lnk.t].toRecv),
          rank, lnk.t) fi;
      if P.queueIn[[lnk.t, lnk.s]] ≠ {} ∧
         tag(head(P.queueIn[[lnk.t, lnk.s]])) =
           connMsg then
           fire internal ReceiveConnect(
             [lnk.t, lnk.s],
             (head(P.queueIn[[lnk.t, lnk.s]])).
               connMsg.l) fi;
      if P.queueIn[[lnk.t, lnk.s]] ≠ {} ∧
         tag(head(P.queueIn[[lnk.t, lnk.s]])) =
             initMsg then
           fire internal ReceiveInitiate(
             [lnk.t, lnk.s],
             (head(P.queueIn[[lnk.t, lnk.s]])).
                initMsg.l,
             (head(P.queueIn[[lnk.t, lnk.s]])).
                initMsg.c,
             (head(P.queueIn[[lnk.t, lnk.s]])).
                initMsg.st) fi;
      if P.queueIn[[lnk.t, lnk.s]] ≠ {} ∧
         tag(head(P.queueIn[[lnk.t, lnk.s]])) =
            testMsg then
         fire internal ReceiveTest([lnk.t, lnk.s],
            (head(P.queueIn[[lnk.t, lnk.s]])).
              testMsg.l,
            (head(P.queueIn[[lnk.t, lnk.s]])).
              testMsg.c) fi;
      if P.queueIn[[lnk.t, lnk.s]] ≠ {} ∧
         head(P.queueIn[[lnk.t, lnk.s]]) = msg(ACCEPT) then
           fire internal ReceiveAccept([lnk.t, lnk.s])  fi;
      if P.queueIn[[lnk.t, lnk.s]] ≠ {} ∧
         head(P.queueIn[[lnk.t, lnk.s]]) = msg(REJECT) then
           fire internal ReceiveReject([lnk.t, lnk.s])  fi;
      if P.queueIn[[lnk.t, lnk.s]] ≠ {} ∧
         tag(head(P.queueIn[[lnk.t, lnk.s]])) = reportMsg
          then fire internal ReceiveReport([lnk.t, lnk.s],
           (head(P.queueIn[[lnk.t, lnk.s]])).reportMsg.w)
      fi;
      if P.queueIn[[lnk.t, lnk.s]] ≠ {} ∧
         head(P.queueIn[[lnk.t, lnk.s]]) = msg(CHANGEROOT)
           then fire internal ReceiveChangeRoot(
             [lnk.t, lnk.s]) fi;
      if P.answered[lnk] = false ∧ P.lstatus[lnk] = branch
         then fire output InTree(lnk) fi;
      if P.answered[lnk] = false ∧ P.lstatus[lnk] = rejected
         then fire output NotInTree(lnk) fi
    od
  od
od
```