Systematic Removal of Nondeterminism for Code Generation in I/O Automata

Mandana Vaziri, Joshua A. Tauber, Michael Tsai, and Nancy Lynch

MIT Laboratory for Computer Science, Cambridge, MA 02155, USA
vaziri,josh,mjt,lynch@lcs.mit.edu,
http://theory.lcs.mit.edu/tds *

Abstract. The Input/Output (I/O) automaton model, developed by Lynch and Tuttle[11], models components in asynchronous concurrent systems as labeled transition systems. IOA is a precise language for describing I/O automata and for stating their properties. A toolset is being developed for IOA, to support distributed software design and implementation. One of the tools consists of a user-assisted code generator from IOA into an imperative programming language such as C or Java.
One aspect that distinguishes IOA programs from programs written in imperative languages, is the presence of nondeterminism, which comes in the form of explicit nondeterministic statements and implicit scheduling choices made during execution. Code generation therefore consists partially of systematically removing all forms of nondeterminism.
In this paper, we describe our approach and design for code generation. We focus on the issue of removing implicit nondeterminism, and specify a transformation on IOA programs that makes all nondeterminism explicit. The programmer can then replace all explicit nondeterminism with deterministic statements, prior to code generation. We also describe this transformation at a semantic level, i.e., at the level of the I/O automaton mathematical model. We show that the transformation defined at the IOA level conforms to the one at the semantic level.

1 Introduction

The Input/Output (I/O) automaton model, developed by Lynch and Tuttle [11], models components in asynchronous concurrent systems as labeled transition systems. It has been used to model and verify many distributed algorithms and distributed system designs, and also to express impossibility results. Lynch's book *Distributed Algorithms* [9] describes many algorithms in terms of I/O automata and contains proofs of various properties of these algorithms. In [10], the authors describe atomic transactions in terms of I/O automata. Examples of work done using this model include among others, implementation of sequentially

* This work supported in part by the Defense Advanced Research Projects Agency under contract F19628-95-C-0118, by the Air Force Office of Scientific Research under contract F49620-97-1-0337, and the National Science Foundation under contracts CCR-9804665 and ACI-9876931

consistent shared objects [3], group communication systems [4, 2], verification of communication protocols [14].

IOA is a precise language for describing I/O automata and for stating their properties. It uses Larch [7] specifications to define the semantics of abstract data types and I/O automata. A toolset is being developed for IOA, to support distributed software design and implementation. These tools range from light weight tools, which check the syntax of automaton descriptions, to medium weight tools, which simulate the action of an automaton, and to heavier weight tools (e.g., theorem provers), which provide support for proving properties of automata.

The toolset design includes a user-assisted code generator from IOA into an imperative programming language such as C or Java. The goal of the process is to produce a collection of programs that runs in a physically-distributed setting and whose correctness has been proved, subject to stated assumptions about the behavior of externally provided system services (e.g., communication services), and subject to assumptions about the correctness of hand-coded data type implementations.

One aspect that distinguishes IOA programs from programs written in an imperative language is the presence of nondeterminism, which comes in the form of explicit nondeterministic statements and implicit scheduling choices made during execution. Code generation therefore consists partially of systematically removing all forms of nondeterminism. This consists of removing all instances of implicit nondeterminism, and making them explicit statements, and having the user replace all explicit nondeterminism with deterministic statements.

In this paper, we review our approach and design for code generation, and the various transformations involved in it. We then carefully focus on one such transformation which makes the implicit nondeterminism explicit. We describe this at the level of IOA programs, and call it the *syntactic NAD*[1] *transformation*, as well as at the level of the I/O automaton model which we call the *semantic NAD transformation*. The motivation for the latter is to provide a more general framework for describing the transformation and present it in its full generality, unconstrained by program syntax. We also use the semantic NAD transformation to prove that our syntactic transformation is "correct," and conforms to it, in a sense we define in a later section.

The outline of the paper is as follows. Section 2 briefly reviews the I/O automaton model and the IOA language. Section 3 describes the code generation process within the IOA toolset. Section 4 presents the syntactic NAD transformation on IOA programs, and Section 5 the semantic NAD transformation on I/O automata. Finally, Section 6 gives a summary and explores directions for future work.

---

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

2 Background

2.1 I/O Automata

An I/O Automaton $\mathcal{A}$ is a tuple consisting of the following components[2].

- $\textsc{Sig}(\mathcal{A})$: a signature, consisting of three disjoint sets of input actions $\textsc{In}_{\mathcal{A}}$, internal actions $\textsc{Int}_{\mathcal{A}}$, and output actions $\textsc{Out}_{\mathcal{A}}$. We use $\textsc{All}_{\mathcal{A}}$ to denote the set of all actions of $\mathcal{A}$, and $\textsc{Loc}_{\mathcal{A}}$ denotes $\textsc{Int}_{\mathcal{A}} \cup \textsc{Out}_{\mathcal{A}}$.
- $\textsc{States}(\mathcal{A})$: a set of *states*.
- $\textsc{States}_0(\mathcal{A})$: a nonempty subset of $\textsc{States}(\mathcal{A})$ known as the *initial states*.
- $\textsc{Trans}(\mathcal{A})$: a *state-transition relation*, where $\textsc{Trans}(\mathcal{A}) \subseteq \textsc{States}(\mathcal{A}) \times \textsc{All}_{\mathcal{A}} \times \textsc{States}(\mathcal{A})$, and for every state $\mathcal{S}$ and every input action π, $(\mathcal{S}, \pi, \mathcal{S}') \in \textsc{Trans}(\mathcal{A})$.

Note that every input action is enabled in every state; we say that I/O automata are *non blocking* on inputs. An *execution fragment* of $\mathcal{A}$ is a sequence of states and actions, $\mathcal{S}_0$, π_0, $\mathcal{S}_1$, π_1, $\cdots$, starting at a state and such that $(\mathcal{S}_i, \pi_i, \mathcal{S}_{i+1}) \in \textsc{Trans}(\mathcal{A})$. An *execution* of $\mathcal{A}$ is an execution fragment starting at a start state. A *trace* of $\mathcal{A}$ is the subsequence of an execution of $\mathcal{A}$ consisting of all input and output actions. We use $\textsc{Traces}(\mathrm{A})$ to denote the set of traces of $\mathcal{A}$.

We now introduce what it means for an automaton to be next-state deterministic, and next-action deterministic.

Definition 1. *An I/O automaton $\mathcal{A}$ is* next-state deterministic *if there is a unique initial state and, for all $\mathcal{S} \in \textsc{States}(\mathcal{A})$ and for all $\pi \in \textsc{All}_{,}$ there is at most one $\mathcal{S}'$ s.t. $(\mathcal{S}, \pi, \mathcal{S}') \in \textsc{Trans}(\mathcal{A})$.*

Definition 2. *An I/O automaton $\mathcal{A}$ is* next-action deterministic *if for all $\mathcal{S} \in \textsc{States}(\mathcal{A})$, there is at most one $\pi \in \textsc{Loc}_{\mathcal{A}}$ enabled in $\mathcal{S}$.*

Most I/O automata that are written are not next-action deterministic. They have a form of *implicit* nondeterminism, which consists of nondeterministically choosing the next enabled action to execute. Transforming an I/O automaton into a next-action deterministic one, is a step in generating code from it.

2.2 IOA Language

IOA is a precise language for describing I/O automata and for stating their properties. It uses Larch [7] specifications to define the semantics of abstract data types. An IOA program A contains the following syntactic components.

- Param_A, contains the parameters to the IOA program A.
- Larch_A, contains the Larch traits used or assumed by A.

[2] We omit *tasks* for the purposes of this paper.

– Act_A, contains the actions of A, where each action a is of the form: $Kind_a\ a(p_1 : P_1, \cdots, p_n : P_n)$ `where` $pred_a$, where $Kind_a$ is either `input`, `output`, or `internal`.
– Var_A, contains the state variables of A together with their types.
– $init_A$, denotes the initial condition of A.
– Td_A, contains the transition definitions of A, where each transition definition d corresponding to an action named a is of the form:

```
Kind_a a(t_1, ..., t_n) where pred_d
        choose ps_d
        pre pre_d
        eff prog_d
            so that soThat_d
```

The transition definition above specifies a transition for action a having parameters $(t_1, \cdots, t_n)$ subject to the predicate $pred_d$. The keyword `choose` introduces parameters used in the body of the transition definition that are chosen non-deterministically. The keyword `pre` introduced the precondition, and `eff` a sequence of statements representing the effect of the action on the state. Finally, the `so that` predicate puts additional constraints on the effect.

An example of an IOA program is given in Figure 1, which represents a process in the LeLann-Chang-Roberts (LCR) leader election algorithm. In this algorithm, a finite set of processes arranged in a ring elect a leader by communicating asynchronously. The algorithm works as follows. Each process sends a unique string representing its name, which need not have any special relation to its index, to its neighbor. When a process receives a name, it compares it to its own. If the received name is greater than its own in lexicographic order, the process transmits the received name to the right; otherwise the process discards it. If a process receives its own name, that name must have travelled all the way around the ring, and the process can declare itself the leader. Here we do not show the channel automata that must be composed with the processes.

Note that in this example, all the parameters of transition definitions are globally unique. This is not a requirement of IOA programs. However, it is an assumption that we make, without loss of generality, on the form of IOA programs that are subjected to the NAD transformation.

3 Code generation using the IOA Toolset

The code generation tool currently under development is designed to translate IOA programs into executable programs written in a standard imperative language. The programmer starts with a distributed algorithm expressed in IOA and uses the tool to produce executable code that runs in a physically distributed computational environment. In our current target environment, the resulting collection of programs runs on a collection of networked workstations. Each host runs a Java interpreter and communicates via (a subset of) the Message Passing Interface (MPI) [5, 1] or TCP/IP [8]. This collection can be proved equivalent

```
automaton Process(I: type, i: I)
  assumes RingIndex(I, String)
  type Status = enumeration of waiting, elected, announced
  signature
    input  receive(m: String, const left(i), const i)
    output send(m: String, const i, const right(i)),
           leader(m: String, const i)
  states
    pending: Mset[String] := {name(i)},
    status:  Status := waiting
  transitions
    input receive(m1, j1, i1)
      eff  if m > name(i) then pending := insert(m, pending)
           elseif m = name(i) then status := elected
           fi
    output send(m2, i2, j2)
      pre m ∈ pending
      eff  pending := delete(m, pending)
    output leader(m3, i3)
      pre status = elected ∧ m = name(i)
      eff  status := announced
```

Fig. 1. IOA specification of election process

to the original algorithm, subject to stated assumptions about the behavior of externally provided system services (e.g., communication services), and subject to assumptions about the correctness of hand-coded data type implementations.

3.1 Approach

To transform the original expression of the algorithm in IOA into an executable, the programmer is guided through a series of successive refinements to create an equivalent form of the program that is suitable for automated translation. Generally, the programmer begins with a simple, global description of the behavior of the system and its interface to the environment. This high-level model tends to be easy to understand and to have important global properties that can be proved. The refined, low-level version of the program is a collection of interacting automata whose form corresponds to the distributed nature of the target environment while preserving the important properties and interface of the system. The programmer can use the validation tools included in the toolset to confirm the correctness of these refinements.

The code generation process starts with an automaton designed to run on a single computational node of the distributed system. To generate code for an entire system, this process is repeated for each algorithmically distinct node. For example, the programs implementing clients and servers are likely to be compiled separately but the code for parameterized clients or peers can be compiled just once.

The process consists of a series of refinements of that "algorithm" automaton. The automaton is connected to (models of) external system services and transformed to eliminate implicit nondeterminism. The resulting IOA program can then be automatically translated into the target imperative language and linked to libraries that implement the external system services.

The code generator is structured as a set of small modules, each of which performs a transformation. Transformations are source-to-source within the IOA language (or in an internal intermediate form) up to the last step. Only in that last step is the IOA program translated into the target language. In our prototype, we have implemented a unified graphical user interface (GUI) to guide the programmer through this process.

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

For each external system service, we design at least two models. First, we write an *abstract model* that describes the interface and the behavior of the service that the IOA programmer wants to use (e.g., point-to-point, reliable, FIFO channels as in [9], Chapter 8). Second, we write a lower-level *concrete model* that corresponds to the interface and behavior of the actual preexisting service (e.g., MPI [15]). We then introduce one or more *auxiliary automata*, written in IOA, that compose with the lower-level model to implement the abstract service. (See Fig. 2). Interfaces to system services may be fixed or algorithmically described. In the former case, the auxiliary automata are fixed and kept as a library. In the latter case, the auxiliary automata must be generated for each program. To create a complete executable for each node, the code generator actually emits code for the *composition* of the algorithm automaton and all the auxiliary IOA automata for all the system services the algorithm accesses.

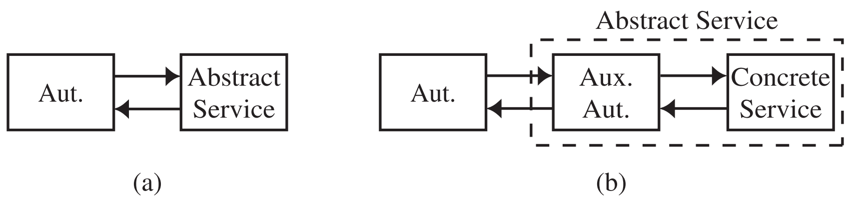

Fig. 2. Auxiliary automata mediate between external system services and algorithm automata

There is a once-per-service proof obligation that the composition of the concrete model and the auxiliary automata implements the abstract service. Once the proof is done, the programmer can use the properties of the abstract model subject to the assumption that concrete model corresponds to the behavior of the service. For example, a proof of correctness for our design that uses MPI to implement reliable FIFO channels appears in [15].

Each external system service introduces constraints on the form of the low-level IOA program that accesses it. For example, since the code generator uses standard workstation networking services, programs submitted for code generation are required to be in *node-channel* form. That is, the algorithm must be described as a collection of algorithm automata (one per node) that communicate via standard, one-way, reliable, FIFO channel automata.

In addition, the model introduces another (technical) constraint. Atomicity requires that the effect part of each transition be done without interruption, even if inputs arrive from the external user or from the communication service during its execution. In our design, such inputs are buffered. In between running non-input actions, the generated program examines buffers for newly arrived inputs, and handles (some of) them by running code for input actions. Since the processing of inputs is delayed (with respect to the performance of these actions by their originators), such delays have the potential to upset precise implementation claims for the algorithm automata. We call this the *input delay insensitivity assumption*.

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

3.2 Design

The code generator consists of a series small program transformers. Each accepts an IOA program or programs as input and produces another IOA program as output (except the last, of course, which translates IOA into the target language). Several of the modules are used by some, if not all, other tools in the IOA toolkit. The granularity of the transformation is driven largely by the usefulness of mixing and matching these tools. Table 1 lists the steps in the process. Below, we describe each step in process. We briefly sketch the function of each module and show how it is used in the code generation process.

The first module to which any IOA program is submitted is the *parser* and *static semantic checker* (or simply *checker*). In addition, to the obvious functions, the checker produces an intermediate representation suitable for use by other tools. This S-expression-based intermediate language (IL) has a simpler parse tree than the more readable IOA source language. [12]. As the checker acts as a front-end to just about all other elements of the toolset, the IL provides a convenient interchange language within the IOA toolset. A checker prototype has been implemented. The IL representation is semantically equivalent to the source representation.

Table 1. Transformers in the IOA code generation process. Numbers in the input column indicate that the transformer takes in the output of the numbered step.

Step	Transformer	Output	Input
1	Checker	internal form	algorithm aut.
2	Interface generator	auxiliary interface aut.	1
3	Composer	primitive aut.	1, 2, auxiliary aut. library
4	NAD	next-action deterministic aut.	3
5	Schedule editor	next-state deterministic aut.	4
6	Code emitter	Java source	5, data type impl. library

The *interface generator* connects an algorithm automaton with the external *console* system service. The console service embodies all user interaction with an IOA program at a particular node. The interface generator creates a customized auxiliary automaton to mediate between the console service and the algorithm automaton. The auxiliary console automaton parses input from the console into input actions for the algorithm automaton. Similarly, output from the algorithm automaton is forwarded to the console. In our initial prototype the abstract model of user input allows the user to nondeterministically invoke any (non-network) input of the algorithm while the concrete model of the console is a simple source and sink for streams of integers [16, 13].

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

In the code generation process, the programmer's original algorithm automaton is composed with the the auxiliary interface automaton (generated in the previous step) with fixed auxiliary network automata. The resulting automaton describes all the behavior of the executable code we actually wish to generate for that node. Unfortunately, it is not obvious how to directly translate this nondeterministic form of the program into a standard imperative language.

The *next-action determinator (NAD)* converts the input IOA program into an equivalent (in the sense of trace inclusion) next-action deterministic program. The NAD form of an IOA program has no implicit nondeterminism. Explicit nondeterminism is grouped in a new *schedule* transition that contains one set of `choose` statements. In this stylized form, each nondeterministic choice is assigned directly to a state variable of the automaton. Section 4 details the actual program transformation. Section 5 shows equivalence of the input and output programs.

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

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

Since all implicit nondeterminism has been removed, the code emitter can start from the unique initial state and perform a loop in which, at each iteration, it executes the unique action enabled in the current state. More specifically, it uses the programmer-provided expressions to determine the next transition and parameter values and then executes that transition with those parameters. Since there is no explicit nondeterminism, this uniquely determines the next state.

4 Syntactic NAD Transformation

In this section, we describe the syntactic NAD transformation on IOA programs, which makes implicit nondeterminism explicit. Then in the next section, we present the transformation in the I/O automaton model and show that the syntactic transformation conforms to the semantic one in a precise sense.

We first introduce some notation. Let t be a term and $\bar{t}$ a sequence of terms. Let $\bar{v}$ and $\bar{x}$ be sequences of variable names, having the same size as $\bar{t}$. We write $t[\bar{v} \mid \bar{t}]$ to denote a term identical to t where every free occurrence of $\bar{v}_i$, $i \geq 1$, if any, has been replaced by $\bar{t}_i$. We write $\bar{v} = \bar{x}$ to denote $\bigwedge_i \bar{v}_i = \bar{x}_i$.

In the program we intend to transform, we assume that the free variables of actual parameters for transition definitions and choose parameters are unique throughout the program[3].

[3] This assumption does not cause loss of generality. It is possible to relax it by first performing some renaming on the program to be transformed.

Given an IOA program A, let B be the following program, with actions Act_B, state variables Var_B, initial condition $Init_B$, and transition definitions Td_B.

- Param_B is identical to Param_A.
- Larch_B is identical to Larch_A.
- Act_B is identical to Act_A, with an additional internal action:
  `internal` $Sched$
- In addition to the state variables of A, Var_B has the following state variables.
  - $w_1 : W_1, \cdots, w_k : W_k$, where w_1 through w_k are the free variables appearing in actual or choose parameters of locally controlled actions of A.
  - $pc : \{d_1, \cdots, d_m, sched\}$, where d_1 through d_m are the locally controlled transition definitions of A in the order that they appear syntactically.
- $Init_B$ is $Init_A \wedge pc = sched$.
- Td_B consists of the following.
  - For each input transition definition d of A, Td_B contains an identical transition definition d' having effect
    $prog_d;\ pc = sched.$
  - For each locally controlled transition definition d of A, corresponding to an action named a, Td_B contains the following transition definition d', where $\bar{w}_d$ denotes the sequence of free variables appearing in actual or choose parameters of d, and $\bar{x}$ is a sequence of fresh variables having the same size as $\bar{w}_d$.
    $Kind_a\ a(t_1[\bar{w}_d \mid \bar{x}], \cdots, t_n[\bar{w}_d \mid \bar{x}])$
        `pre` $pc = d \wedge \bar{w}_d = \bar{x}$
        `eff` $prog_d;$
            $pc := sched$
            `so that` $soThat_d$
  - The following transition definition for the internal action `Sched`, where $a_1, \cdots, a_m$ are the corresponding action names of $d_1, \cdots, d_m$, and $\bar{p}_{a_i}$ and $\bar{t}_{d_i}$, $1 \leq i \leq m$, are the sequence of formal parameters of action a_i, and actual parameters of transition definition d_i, respectively.
    `internal` $Sched$
        `pre` $pc = sched$
        `eff` $w_1 :=$ `choose`;
            $\cdots$
            $w_n :=$ `choose`;
            $pc :=$ `choose` q `where` $q \neq sched;$
            `if` $\neg(pc = d_1 \wedge pre_{d_1} \wedge where_{a_1}[\bar{p}_{a_1} \mid \bar{t}_{d_1}] \wedge where_{d_1}$
            $\vee \cdots$
            $\vee\, pc = d_m \wedge pre_{d_m} \wedge where_{a_m}[\bar{p}_{a_m} \mid \bar{t}_{d_m}] \wedge where_{d_m})$
            `then` $pc := sched$ `fi`

We now consider again the example introduced in Section 2.2, representing a process in the LCR leader election algorithm. Figure 3 presents the result of subjecting that example automaton to the NAD transformation we defined above. The transformation adds an internal action, `sched`, as well as a state variable `pc`. It also adds all parameters of locally controlled actions to the state. The parameters of these actions are renamed, and the names do not have to be unique. The `Sched` action first chooses values for the parameters and for `pc`, and then checks whether an action is enabled given these values. If there is, then that action is chosen to be executed next. The precondition of locally controlled actions have been modified to make this possible.

In this example, all nondeterminism is explicit in the form of `choose` statements which choose a value nondeterministically, subject to a condition specified after the keyword `where`[4]. Moreover, in any state there is at most one action enabled. We show that this automaton satisfies next-action determinism, more precisely in the next section.

5 Semantic NAD Transformationn

In this section, we present the NAD transformation at the I/O automaton level, and refer to it as a semantic transformation. Our motivation is to provide a transformation at the level of the mathematical model as a complement to the syntactic one. This gives a more general characterization of the transformation, and provides a framework in which we can verify that the syntactic transformation is correct.

5.1 Semantic Transformation

We define the semantic transformation by giving a relation NAD($\mathcal{A}$), which denotes the set of all next-action deterministic I/O automata corresponding to $\mathcal{A}$.

The relation NAD($\mathcal{A}$) is defined by giving restrictions on the components of the automata that belong to it. An automaton $\mathcal{B}$ in NAD($\mathcal{A}$) has all the actions of $\mathcal{A}$, and a set of additional internal actions, which we call *scheduling actions*, and denote Π.

Next we define the restrictions on the set of states of $\mathcal{B}$. We first define a partition $\mathcal{P}$ on the states of $\mathcal{B}$, having an equivalence class for each action that is locally controlled in $\mathcal{B}$. We write $\mathcal{P}_\pi$ to denote the equivalence class corresponding to π. Informally, a state $\mathcal{S}$ is in $\mathcal{P}_\pi$, if action π has been chosen for execution in state $\mathcal{S}$. The action π may or may not be enabled in $\mathcal{S}$.

We then define a second partition $\mathcal{Q}$ on the states of $\mathcal{B}$, having an equivalence class for each state of $\mathcal{A}$. We write $\mathcal{Q}_{\mathcal{S}_A}$ to denote the equivalence class corresponding to the state $\mathcal{S}_A$ of $\mathcal{A}$. Informally, a state $\mathcal{S}$ of $\mathcal{B}$ is in $\mathcal{Q}_{\mathcal{S}^+}$, if it "corresponds" to $\mathcal{S}^+$.

Having defined these two partitions we state that they must be restricted such that $\forall \mathcal{S} \in \textsc{States}(\mathcal{A})$, $\pi \in \textsc{Loc}_\mathcal{B}$, $\mathcal{Q}_\mathcal{S} \cap \mathcal{P}_\pi \neq \emptyset$ holds. This condition allows us to introduce the following notation. We write $[\pi, \mathcal{S}]$ to denote $\mathcal{Q}_\mathcal{S} \cap \mathcal{P}_\pi$, and call it a *sector*.

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

```
automaton Process(I: type, i: I)
  assumes RingIndex(I, String)
  type Status = enumeration of waiting, elected, announced
  signature
    input  receive(m: String, const left(i), const i)
    output send(m: String, const i, const right(i)),
           leader(m: String, const i)
    internal Sched
  states
    pending: Mset[String] := {name(i)},
    status:  Status := waiting,
    m2, m3: String,
    i2, i3: I,
    j2: I,
    pc: enumeration of send, leader, sched
    so that pc := sched
  transitions
    input receive(m1, j1, i1)
      eff  if m1 > name(i1) then pending := insert(m1, pending)
           elseif m1 = name(i1) then status := elected
           fi;
           pc := sched
    output send(m, i, j)
      pre  pc = send ∧ m = m2  ∧ i = i2 ∧ j = j2
      eff  pending := delete(m2, pending);
            pc := sched
    output leader(m, i)
      pre  pc = leader ∧ m = m3 ∧ i = i3
      eff  status := announced
    internal Sched
      pre pc = sched
      eff m2 := choose; m3 := choose;
          i2 := choose; i3 := choose; j2 := choose;
          pc = choose q where q ≠ sched;
          if not (pc = send ∧ m2 ∈ pending
                ∨ pc = leader ∧ status = elected ∧ m3 = name(i3))
          then pc := sched fi
```

Fig. 3. IOA specification of election process

Informally, $\mathcal{B}$ is an automaton which behaves like $\mathcal{A}$, but in addition it explicitely chooses which action to execute next. For each state $\mathcal{S}_A$ of $\mathcal{A}$, $\mathcal{B}$ has a set of states corresponding to $\mathcal{S}_A$, and these states encode different choices for the next action to execute. A state in sector $[\pi, \mathcal{S}]$ corresponds to state $\mathcal{S}$ of $\mathcal{A}$ with the choice of executing action π next.

It remains to specify the initial states of $\mathcal{B}$, as well as its transitions. The initial states of $\mathcal{B}$ are all those that belong to $[\pi, \mathcal{S}_0]$, for $\pi \in \Pi$ and $\mathcal{S}_0$ is an initial state of $\mathcal{A}$.

The transitions of $\mathcal{B}$ must be such that if $\mathcal{B}$ goes from state $\mathcal{S}$ to state $\mathcal{S}'$ while executing action π of $\mathcal{A}$, then $\mathcal{S}$ corresponds to state $\mathcal{S}_A$ of $\mathcal{A}$, $\mathcal{S}'$ corresponds to $\mathcal{S}'_A$, and $(\mathcal{S}_A, \pi, \mathcal{S}'_A)$ is a transition of $\mathcal{A}$.

Figure 4 illustrates the execution of some automaton b $\in \mathcal{B}$. In this figure, rows represent partition $\mathcal{P}$ and columns partition $\mathcal{Q}$. The states in equivalence classes of $\mathcal{P}$ corresponding to scheduling actions are labeled *scheduling states*. An execution of $\mathcal{B}$ starts at an initial state, which is a scheduling state corresponding to an initial state of $\mathcal{A}$. Then $\mathcal{B}$ executes a scheduling action, and chooses an action to execute that is enabled in the current state. This causes the automaton to move vertically into a state in which a choice has been made. Next, $\mathcal{B}$ executes the chosen action, and moves to a scheduling state, and this process is repeated.

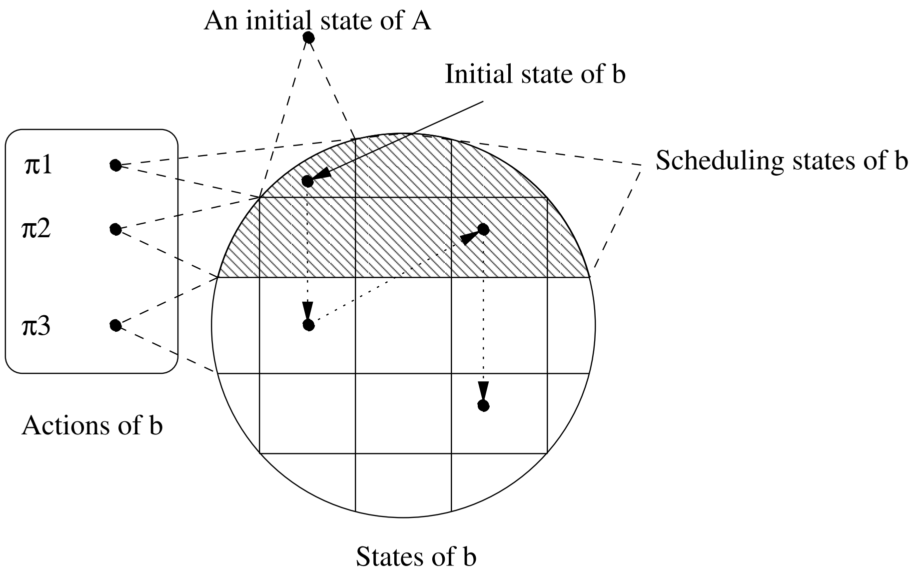

Fig. 4. The exectution of some automaton b $\in \mathcal{B}$.

Formally, the Nad relation is given as follows.

Definition 3. $\mathcal{B} \in \mathrm{NAD}(\mathcal{A})$ *if and only if there exists a set* Π *of actions disjoint from* $\mathrm{ALL}_{\mathcal{A}}$*, and partitions* $\mathcal{P}$ *and* $\mathcal{Q}$ *on* $\mathrm{STATES}(\mathcal{B})$*, having an equivalence class for each action in* $\mathrm{LOC}_{\mathcal{B}}$ *and each state of* $\mathcal{A}$*, respectively, such that:*

1. $\forall \mathcal{S} \in \mathrm{STATES}(\mathcal{A})$, $\pi \in \mathrm{LOC}_{\mathcal{B}}$, $\mathcal{Q}_{\mathcal{S}} \cap \mathcal{P}_{\pi} \neq \emptyset$.
2. $\mathrm{SIG}(\mathcal{B}) = (\mathrm{IN}_{\mathcal{A}}, \mathrm{INT}_{\mathcal{A}} \cup \Pi, \mathrm{OUT}_{\mathcal{A}})$.
3. $\mathrm{STATES}_0(\mathcal{B}) = \{\mathcal{S} \mid \exists \pi \in \Pi, \mathcal{S}_0 \in \mathrm{STATES}_0(\mathcal{A}), \mathcal{S} \in [\pi, \mathcal{S}_0]\}$.
4. *A transition* $(\mathcal{S}, \pi, \mathcal{S}')$ *is in* $\mathrm{TRANS}(\mathcal{B})$ *if and only if:*

$$\begin{aligned}
&\exists \pi_0 \in \Pi, \mathcal{S}_A, \mathcal{S}'_A \in \mathrm{STATES}(\mathcal{A}) \textit{ such that:}\\
&\quad \pi \in \mathrm{IN}_{\mathcal{A}} \wedge (\mathcal{S}_A, \pi, \mathcal{S}'_A) \in \mathrm{TRANS}(\mathcal{A}) \wedge \mathcal{S} \in \mathcal{Q}_{\mathcal{S}_A} \wedge \mathcal{S}' \in [\pi_0, \mathcal{S}'_A]\\
&\quad \vee\\
&\quad \pi \in \mathrm{LOC}_{\mathcal{A}} \wedge (\mathcal{S}_A, \pi, \mathcal{S}'_A) \in \mathrm{TRANS}(\mathcal{A}) \wedge \mathcal{S} \in [\pi, \mathcal{S}_A] \wedge \mathcal{S}' \in [\pi_0, \mathcal{S}'_A]\\
&\quad \vee\\
&\quad \pi \in \Pi \wedge \mathcal{S} \in [\pi, \mathcal{S}_A] \wedge \mathcal{S}' \in [\pi_0, \mathcal{S}_A] \cup [\pi', \mathcal{S}_A],\\
&\quad \textit{for } \pi' \in \mathrm{LOC}_{\mathcal{A}} \textit{ such that } \pi' \textit{ is enabled in } \mathcal{S}_A.
\end{aligned}$$

5.2 Soundness of Semantic Transformation

In this section, we argue that any I/O automaton in $\mathrm{NAD}(\mathcal{A})$ has the same traces as $\mathcal{A}$ and satisfies next-action determinism.

Lemma 1. *Let* $\mathcal{B} \in \mathrm{NAD}(\mathcal{A})$*. Then* $\mathrm{TRACES}(\mathcal{B}) = \mathrm{TRACES}(\mathcal{A})$.

Proof. We do not show the proof here. This can be done by induction on the length of an execution [17].

Lemma 2. *Let* $\mathcal{B} \in \mathrm{NAD}(\mathcal{A})$*.* $\mathcal{B}$ *is next-action deterministic.*

Proof. Let $\mathcal{S}$ be a state of $\mathcal{B}$. In order for any locally controlled action π of $\mathcal{B}$ to be enabled in $\mathcal{S}$, $\mathcal{S}$ must be such that $\mathcal{S} \in \mathcal{P}_{\pi}$, by construction. Since $\mathcal{P}$ is a partition on $\mathrm{STATES}(\mathcal{B})$, then there is at most one locally controlled action enabled in $\mathcal{S}$. Thus $\mathcal{B}$ is next-action deterministic.

5.3 Correspondence Between the Syntactic and the Semantic Transformations

In this section, we argue that the syntactic transformation described previously, conforms to the semantic transformation above. For this we introduce the notion of the semantic of an IOA program *sem*. For an IOA program A, $sem(A)$ denotes the I/O automaton corresponding to A. The semantics of IOA programs have been defined precisely in [6], and we will not reproduce them here.

Let B be the result of transforming IOA program A with the NAD syntactic transformation. Then the following holds.

Theorem 1. *sem(B)* $\in \mathrm{NAD}(sem(A))$

We do not show the formal proof here, since that would require showing as well the precise definition of the semantics of a program, but we give the informal argument behind the proof. The proof is presented in [17].

In order for *sem(B)* to be related to *sem(A)* via the semantic NAD transformation, there must exist a set Π of actions and partitions $\mathcal{P}$ and $\mathcal{Q}$ satisfying the conditions of Definition 3. In this case, Π consists of a single action *Sched*.

The state of B consists of all the states of A and some additional state variables, *pc* and parameter lists, that indicate which action to execute next. We specify the partition $\mathcal{Q}$ as follows. A state $\mathcal{S}$ of B is in $\mathcal{Q}_{\mathcal{S}_A}$, for a state $\mathcal{S}_A$ of A, if the portion of state $\mathcal{S}$ corresponding to the state of A, is identical to $\mathcal{S}_A$.

We specify the partition $\mathcal{P}$ as follows. A state $\mathcal{S}$ of B is in $\mathcal{P}_\pi$, for a locally controlled action of B, if the extra state of $\mathcal{S}$ indicates that π is the next action to run.

Given these values for Π, $\mathcal{Q}$, and $\mathcal{P}$, we can verify that the conditions of Definition 3 are satisfied.

6 Conclusion

In this paper, we presented our approach and design for code generation in I/O automata. We have also presented in detail, one of the transformation involved in this process, the syntactic NAD transformation (synNAD). Further, we described the transformation at the level of the mathematical model, the semantic NAD transformation (semNAD), and showed that the syntactic transformation conforms to it in a precise sense.

The introduction of a semantic transformation illustrates a method for designing program transformations in this context. We use semNAD as a "specification" transformation, that is "implemented" by synNAD. Our method consists of giving a semantic transformation and a syntactic one, and proving that the latter conforms to the former in a precise sense. This method has the following advantages.

- A precise description of the transformation is given, which makes mathematical manipulation possible.
- The definition of the transformation is as general as possible.
- It is not tied down to a particular language syntax.
- Syntactic variations of the same transformation can be easily shown to be equivalent.

As future work, we plan to investigate ways of assisting the user in replacing explicit nondeterministic statements with deterministic ones, and to complete a working prototype of the code generator.

Acknowledgments

We thank Steve Garland, Antonio Ramirez, and Holly Reimers for their contributions to the development of the IOA project.

References

[1] 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.

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

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

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

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

[6] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. *IOA: A Language for Specifying, Programming and Validating Distributed Systems.* Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, December 1997. `URL http://sds.lcs.mit.edu/~garland/ioaLanguage.html`.

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

[8] Idit Keidar and Roger Khazan. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. Technical report, MIT Laboratory for Computer Science, 1999. In preparation, submitted for conference publication.

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

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

[11] 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.

[12] Antonio Ramirez. IOA simulator design and implementation, 1999. Manuscript.

[13] Holly Reimers. An unparser for the IOA intermediate language, 1999. Manuscript.

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

[15] Joshua A. Tauber. IOA code generation - theory and practice. Manuscript.

[16] Michael Tsai. Design and implementation of the IOA GUI, interface generator, and NAD modules, 1999. Manuscript.

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