Implementing I/O-Automaton Specifications on Erlang

Yoshinobu Kawabe and Jun Zhao
Department of Information Science
Aichi Institute of Technology
1247 Yachigusa Yakusa-cho, Toyota-shi, Aichi, 470-0392, Japan
Email: \{kawabe, b09723bb\}@aitech.ac.jp

Abstract—An I/O-automaton is a formal system for describing and analyzing distributed algorithms, and IOA is a specification language based on I/O-automaton theory. The IOA language is not a programming language, so specifications written in IOA cannot be run. If we can run communicating I/O-automata in a distributed environment directly, then IOA can be employed as an executable specification language; this means that we can prototype various communication systems rapidly. In this paper, we describe how to implement IOA specifications in a network programming environment. Specifically, we introduce a transformation from IOA into Erlang, which is a functional programming language that has facilities for TCP/IP network communication. Also, we demonstrate how a distributed system written in IOA is translated into an executable Erlang program code.

I. INTRODUCTION

IOA is a formal specification language based on I/O-automaton theory [5][6], which is a formal system for describing and analyzing distributed algorithms. IOA is usually employed to specify a distributed system formally, and its correctness is proven with a theorem prover. This makes it possible to design highly reliable distributed systems.

The IOA language is not a programming language, so specifications written in IOA are not executable. If we can run communicating I/O-automata in a distributed environment directly, IOA can be employed as an executable specification language. This enables us to prototype various communication systems rapidly. Figure 1 shows a sketch for developing a distributed system with IOA. We first write an I/O-automaton for a distributed system, and we also define functions as a set of equations. These descriptions are combined as an IOA specification, which is translated into a first-order predicate logic formula to prove the correctness with a theorem proving tool such as Larch Prover (LP) [2]. After proving the correctness, we translate the IOA specification into Erlang [9], which is a functional programming language that has facilities for TCP/IP network communication. Finally, the processes are run on Erlang’s run-time systems where they communicate with each other.

This paper’s contribution is to provide a strategy for translating an IOA specification into an Erlang program code (see the bottom of Fig. 1). Erlang is a functional programming language, so it can deal with equations directly. Moreover, Erlang has facilities for TCP/IP network communication, so if we can translate the I/O-automaton portion of a specification, then we can run the IOA specification in a distributed setting.

This paper is organized as follows. Section II describes the basic notions and notations for both the IOA specification language and the Erlang programming language. In Section III, we introduce a technique for translating IOA into Erlang. Section IV describes a method of implementing inter-process communication. Finally, we discuss this work in Section V. In particular, we consider the correctness of the specifications and scheduling issues.
In this section, we first provide a brief overview of the IOA specification language. We then describe the Erlang programming language.

A. IOA Specification Language

An I/O-automaton is a formal model for distributed algorithms, and IOA is a specification language based on I/O-automaton theory. IOA has actions, which are classified as either internal, input or output. The internals correspond to the transitions of a conventional automaton, while the inputs and outputs (also called externals) describe message passing between I/O-automata.

The following is an IOA example that models a communication channel. Here, we assume several pre-defined datatypes or operators:

- datatype Seq for a sequence,
- constant [] (empty sequence),
- operator := (substitution),
- operator ++ (appending two sequences),
- operator last (a sequence’s tail element) and
- operator init (deleting the tail element).

Example 1: A channel IOA from I to J:

automaton ChannelIOA(I, J: ID)

| signature |
| input send(M: MES, const I, const J) |
| output recv(M: MES, const I, const J) |

| states |
| QUEUE: Seq[MES] := [] |

| transitions |
| input send(M, I, J) |
| eff QUEUE := [M] ++ QUEUE; |
| output recv(M, I, J) |
| pre QUEUE /= [] |
| M = last(QUEUE) |
| eff QUEUE := init(QUEUE) |

ChannelIOA(I, J) consists of the following portions:

1) signature: declares actions and their sorts;
2) states: declares variables and their initial values. In this example, a variable QUEUE for a message sequence is declared with its initial value empty ([ ]) and a sort Seq[MES];
3) transitions: defines a body for each action, and the body is described in a precondition-effect style. There are two actions in the above example:

In I/O-automaton theory, an output action of an I/O-automaton can synchronize with an input action of another I/O-automaton, if these actions have the same name. For example, if an output send(M, I, J) appears in IOA I, then this action can synchronize with the channel IOA’s input action send(M, I, J) (also see Fig. 2).

A computer-assisted correctness proof is possible for IOA specifications. IOA-Toolkit [7][10] is a formal verification tool for the IOA specification language, and it translates an IOA specification into first-order logic formulae. After this translation, the correctness of a specification is proven with a theorem proving tool, Larch Prover (LP) [2], which is a theorem prover based on both first-order predicate logic and equational theory that supports IOA-Toolkit.

B. Erlang Programming Language

Erlang is a functional programming language that was originally designed by Ericsson to support various applications including distributed systems. Erlang provides language-level features for creating and managing concurrent processes. For example:

- For concurrency, Erlang follows the Actor model. In this model, processes have no shared memory and they can communicate asynchronously. Also, a process in Erlang is extremely lightweight, and very large numbers of concurrent processes can be run in a single Erlang system;
- Erlang is designed to be run in a distributed environment. An Erlang virtual machine is called a “node”, and processes running on Erlang nodes constitute a distributed system. In Erlang, a destination process at a different node can be handled as a process at the same node. That is, programmers do not need to consider hosts or locations. This enables programmers to concentrate on writing the other essential portions of a distributed system.
These features of Erlang allow us to simplify concurrent programming. The following is an example Erlang program that creates three processes.

**Example 2 (An Erlang program with three processes):**
This is a program for sending “Hello” from process A to process C via process B.

```erlang
-module(netProg).
-export([toplevel/0, procC/0, procB/1, procA/1]).
toplevel() ->
    C = spawn(fun procC/0),
    B = spawn(fun() -> procB(C) end),
    spawn(fun() -> procA(B) end),
    "Done".
procC() ->
    receive
        X -> io:format("C says \"p\"n", [X])
    end.
procB(C) ->
    receive
        X -> C ! X
    end.
procA(B) -> B ! "Hello".
```

We can see that an Erlang program is a series of the form

\[ X \rightarrow Y. \]

This is an oriented equation; in other words, it is a rewrite rule. The left-hand side of the equation is replaced with the right-hand side.

### III. Translating IOA Specifications into Erlang

In this section, we describe how to translate an IOA specification into its corresponding Erlang code. In the rest of this paper, for simplicity, we introduce the following requirements for IOA specifications:

- Automata in a distributed system communicate through a channel I/O-automaton, which is shown as Example 1;
- Each automaton has one (unique) identifier;
- We employ Erlang’s basic functions for list processing etc.

An IOA action consists of a precondition part and an effect part.\(^1\) The precondition part of an action describes a condition for firing the action, and the effect part contains a sequence of substitution commands. For example, the output action **display** in Fig. 3 has precondition part

\[ \text{pre } \text{COUNTER} < 100 \land \text{C} = \text{COUNTER} \]

and effect part

\[ \text{eff } \text{COUNTER} := \text{COUNTER}+1. \]

To implement these portions of an action, we define the following sorts of Erlang functions:

1) **Precondition functions:** are defined for output and internal actions, and each action’s precondition is translated into a precondition function. For example, the precondition function of action **display** is:

```erlang
pre_output_display(_, COUNTER) ->
    Tmp1_output_display = (COUNTER<100),
    if not(Tmp1_output_display) -> { false, [] };
                          true -> C = COUNTER,
                          { true, C }
end.
```

Action **display** tries to find some value for variable \( \text{C} \) with \( \text{COUNTER} < 100 \land \text{C} = \text{COUNTER} \). If such a value is found, then the precondition function returns the value with \( \text{true} \); otherwise, it returns an empty sequence (\([]\)) with \( \text{false} \).

2) **Effect functions:** are defined for all types of actions, and the effect part of each action is translated into an effect function. For example, the effect function of action **display** is:

```erlang
eff_output_display(ID, COUNTER, C) ->
    side_effect_output_display(ID, COUNTER, C),
    NextCOUNTER = COUNTER+1,
    mainloop(ID, NextCOUNTER).
```

\(^1\)Note that input actions do not have a precondition.
mainloop(ID, COUNTER) ->
{ COND_out01, C_out01 } = pre_output_display(ID, COUNTER),
if COND_out01 ->
  eff_output_display(ID, COUNTER, C_out01);
true -> receive
  ( "add", X_in01 ) ->
    eff_input_add(ID, COUNTER, X_in01);
  stop ->
    stop;
  _Other ->
    io:format("Invalid Message Received!\n"),
    mainloop(ID, COUNTER)
end.

end.

Fig. 4. Main-Loop Function for Automaton count

We can see this function has three parameters:
- ID: an identifier of automaton count,
- COUNTER: a global variable of automaton count, and
- C: a local variable for action display.

The action display increments the global variable COUNTER, so this effect function also increments COUNTER and the result is stored in variable NextCOUNTER.

The above effect function requires the following functions:

3) Side effect functions: are employed to describe side effects. IOA actions simply change the value of an automaton's global variable. Hence, to enable an IOA program to interact with the outside world such as I/O-ports, filesystems and networks, a programmer should specify a side effect. For example, if we need to write data to a display when action display is fired, then we should define a side effect function as follows:

\[
\text{side_effect_output_display(\_ID,}\ 
\text{\_COUNT,}\ C) \\
\quad \rightarrow \ 
\text{io:format("Count: \text{\_p\_n", [C]).}
\]

In this example, this side effect function writes data C to a display. For simplicity, in this study side effect functions are defined only for output actions, but it is also possible to introduce a side effect function for input actions.

4) The main-loop function: is the main loop of a program. The main-loop function for automaton count is shown in Fig. 4. We can see this function has two parameters:
- ID: an identifier of automaton count and
- COUNTER: a global variable of automaton count.

This function first evaluates the precondition function of display. If the precondition is satisfied, then the main-loop function calls the display's effect function; otherwise, the main-loop function tries to evaluate other precondition functions. In this case, there are no other precondition functions, so the main-loop function evaluates input actions. If a message

\[
\{ \text{"add", value } \}
\]

is sent to this automaton, the main loop function receives the message and calls

\[
\text{eff_input_add(ID, COUNTER, value),}
\]

which is an effect function for action add(X) and its definition is:

\[
\text{eff_input_add(ID, COUNTER, X) ->}
\text{NextCOUNTER = COUNTER+X,}
\text{mainloop(ID, NextCOUNTER).}
\]

The main-loop function also accepts stop, which halts automaton. Other messages are handled as invalid messages, and they are discarded.

In addition to the above functions, a top-level function is introduced. The top-level function for automaton count is as follows:

\[
\text{toplevel(Name) ->}
\text{Port_mainloop = spawn(count, mainloop0, []),}
\text{register(Name, Port_mainloop),}
\text{Port_mainloop.}
\]

\[
\text{mainloop0()}
\quad \rightarrow \ 
\text{mainloop(self(), 0).}
\]

The function toplevel creates a sub-process that manages automaton count. The sub-process starts the main-loop function mainloop(self(), 0). After creating the sub-process, the toplevel function returns an identifier of the automaton. Instead of the identifier, we can use an alias that is given by variable Name.
IV. WRITING COMMUNICATING AUTOMATA

In this study we employ a channel IOA for inter-process communication. Automaton channelIOA(I, J) in Example 1 represents a communication channel from automaton I to automaton J. In theory, the above channel I/O-automata are provided for any I and J, but it is hard to implement them all.

In this study, we use Erlang’s communication facilities; specifically, we describe a side effect function for automaton I’s output action send(M, I, J) as follows:

```erlang
% precondition function for send(M, I, J)
pre_output_send(ID, DESTID, SENDQ) ->
  Tmp1_output_send = (SENDQ =/= []),
  if not(Tmp1_output_send) ->
    true -> M = head(SENDQ),
    I = ID,
    J = DESTID,
    [true, M, I, J ];
end.

% effect function for send(M, I, J)
side_effect_output_send(ID, DESTID, SENDQ, M, I, J) ->
  side_effect_output_send(ID, DESTID, SENDQ, M, I, J) ->
  mainloop(ID, DESTID, NextSENDQ).
```

Figure 6 shows an example IOA specification that has actions send(M, I, J) and recv(M, I, J) for communication. There are two automata, sender and recipient in this example. Automaton sender can send a number to automaton recipient, while the recipient automaton writes data to a display.

When we model this kind automaton, we usually employ two sorts of variables for message passing:
- SENDQ: an output buffer and
- MESQ: an input buffer.

Actions send and recv rewrite these variables, and these actions of I/O-automata are translated into Erlang programs in Figs. 6 and 7.
V. DISCUSSION

A. Proving Correctness of Specifications

We can see that Section IV’s automata sender and recipient implement Section III’s automaton count.

Let Comp$(\alpha, \beta)$ be the result of composing

- sender$(\alpha, \beta)$,
- recipient$(\beta)$ and
- channelIOA$(i, j)$ for any $i$ and $j$,

and let CompHide$(\alpha, \beta)$ be the result of hiding send$(m, i, j)$ and recv$(m, i, j)$ from Comp$(\alpha, \beta)$ for any $m$, $i$ and $j$. We have

\[
traces(CompHide(\alpha, \beta)) \subseteq traces(count)
\]

where traces$(X)$ is a set of all the traces. That is, the composition of sender and recipient is an implementation of count. This is an example showing that we can prove the correctness of an IOA specification formally. Moreover, with this paper’s transformation, it is possible to run the verified specification over the network directly.

B. On Scheduling

Resolving nondeterminism is an important issue as regards implementing IOA specifications. In this study, we first evaluate the preconditions for output and internal actions and then we evaluate the input actions. If two or more sets of parameters can satisfy a precondition, the first set is chosen. When we employ this strategy, some action might not be executed. A solution to this problem is to evaluate preconditions concurrently. After evaluating all the preconditions, a scheduler selects and enables one action. To avoid unfair executions, a mutual exclusion algorithm, such as Lamport’s bakery algorithm, should be employed when choosing an action. Another solution is to require the programmer to write a schedule; this strategy is employed in [8].

The IOA language provides another sort of nondeterminism. It is the explicit nondeterminism introduced by a programmer. This type of nondeterminism can be avoided if we deal solely with a subclass of I/O-automata called pseudo-deterministic automata [1]. An automaton is called pseudo-deterministic if its every state has at most one successor state for any action. A pseudo-deterministic IOA is useful from an implementation viewpoint, but it is also useful from a verification viewpoint since transitions of a pseudo-deterministic automaton can be formalized by using equations with which a theorem prover can solve problems efficiently. We believe a pseudo-deterministic automaton is powerful enough to describe distributed systems. We have formalized distributed systems, namely a rendezvous-style communication system [3] and an e-voting protocol [4] with pseudo-deterministic automata.

C. Related Work

Tauber et al. [8] presented a compiler for I/O-automaton specifications. They used Java to run automata on a group of networked workstations, whereas Erlang is employed in this paper. Erlang’s language-level features for concurrent processes and communication enable us to implement IOA specifications directly.

VI. CONCLUSION

This paper discussed how to implement IOA specifications. In this study we employed the Erlang functional programming language. We described a strategy for transforming IOA specifications into Erlang programs. In this transformation we introduced four sorts of functions: precondition functions, effect functions, side effect functions and a main-loop function. We also discussed the transformation of send and recv actions, which are used for inter-process communication.

As future work, we are planning to run various distributed algorithms and communication protocols, such as leader election algorithms or a protocol for implementing a shared memory. It is also crucial to develop a compiler that converts an IOA specification into an Erlang program based on the implementation method proposed in this paper. It is especially important to introduce a scheduler.

REFERENCES