Finite State Automata Semantics in Communicating Sequential Processes

Vladimir Dimitrov

Finite State Automata Semantics in Communicating Sequential Processes: *Traditionally, distributed systems and protocols are described with finite state automata. Later on, other more powerful mathematical tools for specification and analyses of distributed systems have been developed, such as Petri nets, CSP etc. Modern tools and notations for specification, development and implementation of distributed systems are based on them. In commercial tools, that use finite state automata, as a base for business process specification, the problem is the need to convert older specifications into new one without losing the semantics. Newly developed tools are based on Petri nets or CSP. They are more powerful in specification and analyses, but they have to support continuity. Intention of this paper is formally to specify finite state automata in CSP. Finite state automata semantics is clear, but there are needs for conversion of business processes specified in them to new form without losing the semantics.*

Key words: *Finite State Automata, Communicating Sequential Processes, Semantics.*

MOTIVATION

Traditionally, distributed systems and protocols are described with finite state automata (finite state machines). As result of that, many tools based on finite state automata have been developed and used. Such an example is business state machines used in IBM WebSphere Integration Developer [1]. Later on, other more powerful mathematical tools for specification and analyses of distributed systems have been developed, such as Petri nets [2], CSP [3], and so on. Modern tools and notations for specification, development and implementation of distributed systems are based on them. For example, Petri nets concepts are broadly used for specification of business processes in notations like UML (activity diagrams) [4], WS-BPEL [5], BPMN [6], etc. The newer mathematical tools are more powerful that the older ones. For example, Petri nets have more expressive power than finite state automata, but are less expressive than CSP. Our intention, here in this paper, is not to compare them. In commercial tools, that use finite state automata, as a base for business process specification, the problem is the need to convert older specifications into new one without losing the semantics. Newly developed tools are usually based on Petri nets or CSP. They are more powerful in specification and analyses, but they have to support continuity with the older developments. Such an example is IBM WebSphere Integration Developer that nowadays is based on WS-BPEL, but has to support backward compatibility with the business state machines. Intention of this paper is formally to specify finite state automata in CSP. Finite state automata semantics is clear, but there are needs for conversion of business processes specified in them to new form without losing the semantics.

DEFINITIONS

There are many kinds of finite state automata: deterministic, non-deterministic, Mealy machines, Moor machines etc. Some extensions like Turing machine get outside the expressive power of finite state automata, but they are not subject of this paper. We will use the next definition of finite state automata: Finite state automata A with alphabet V is the 5-tuple A= <K, V, δ, q0, F>, where: K is non empty finite set of automata states; V is non empty finite set of input symbols - the alphabet; δ is transition function with domain K x V and range K; $q0 \in K$ is the initial state; $F \subseteq K$ is the set of final states. It is possible F to be empty, in this case the machine is executed forever or to stop not in final state. When automata stop in finite state it has finished normally its work, but if it stops in non final state – this means that machine is broken in some way. A finite state automata is deterministic if its transition function is defined in every state for all input symbols, i.e. domain δ is equal to K x V. If the finite state automata has at least one state for which the transition function is

not defined for all input symbols, then this automata is non deterministic. From the theory, we know that every non deterministic machine can be modeled deterministic one. This means that they are equivalent in expressiveness. In some definitions of finite state automata output alphabet O and output function ω are included. When a transition is executed, it is possible to be generated some output. Domain of ω is subset of the domain of δ, but its range is O.

Some examples follow. First example: There are no final states (F =∅). This machine is non-deterministic with: K = {q0, patients, fields, setup, ready, beam_on}; V = {select_patient, select_field, enter, ok, start, stop, intlk}; δ = {(q0, enter) ↦ fields, (patients, enter) ↦ fields, (fields, select_patient) ↦ patients, (fields, enter) ↦ setup, (setup, select_patient) ↦ patients, (setup, select_field) ↦ fields, (setup, ok) ↦ ready, (ready, select_patient) ↦ patients, (ready, select_field) ↦ fields, (ready, start) ↦ beam_on, (ready, intlk) ↦ setup, (beam_on, stop) ↦ ready, (beam_on, intlk) ↦ setup}

In the second example, the machine has output: K = {q0, P0, P1}; V = {init, 00, 01, 10, 11}; δ = {(q0, init) ↦ P0, (P0, 00) ↦ P0, (P0, 01) ↦ P0, (P0, 10) ↦ P0, (P0, 11) ↦ P1, (P1, 00) ↦ P0, (P1, 01) ↦ P1, (P1, 10) ↦ P1, (P1, 11) ↦ P1} O = {NULL, 0, 1}; ω = {(q0, init) ↦ NULL, (P0, 00) ↦ 0, (P0, 01) ↦ 1, (P0, 10) ↦ 1, (P0, 11) ↦ 0, (P1, 00) ↦ 1, (P1, 01) ↦ 0, (P1, 10) ↦ 0, (P1, 11) ↦ 1}

SPECIFICATION IN Z-NOTATION

Here, we will be more strict specifying finite state automata in Z-notation [7]. Basic sets are:

$$[STATES, INPUTS, OUTPUTS]$$

where STATES is non empty final sets of automata states, INPUTS is the set of input symbols (events), OUTPUTS is the set of all output symbols.

$$q0 : STATES;\ NULL : OUTPUTS;\ FINALS : \mathbb{F}\ STATES$$

$$STATES \neq \varnothing \land (\exists n : \mathbb{N} \bullet \#STATES \leqslant n) \land INPUTS \neq \varnothing \land q0 \notin FINALS$$

where q0 is the initial state, NULL is a special output symbol – nothing is outputted, FINALS is possible empty subset of STATES – final states.

FSM

$$transition : STATES \times INPUTS \rightarrow\!\!\!\!\!\!+\ \ STATES;\ output : STATES \times INPUTS \rightarrow\!\!\!\!\!\!+\ \ OUTPUTS$$
$$current : STATES$$

$$\mathrm{dom}\ output = dom\ transition \land q0 \in \mathrm{dom}\ (\mathrm{dom}\ transition) \land FINALS \cap \mathrm{dom}\ (\mathrm{dom}\ transition) = \varnothing \land$$
$$FINALS \subseteq \mathrm{ran}\ transition \land STATES \setminus \{q0\} = \mathrm{ran}\ transition \land$$
$$STATES \setminus FINALS = \mathrm{dom}\ (\mathrm{dom}\ transition)$$

Finite states machine consists of transition function, output function and current state. Transition and output functions have the same domain Cartesian product of STATES and INPUTS. The initial state q0 is part of the domain of transition function. FINALS states have only input arcs, but no output arcs. Only q0 has no input arcs. All states, except final ones, have to have input and output arcs.

FSMInit

$$FSM$$

$$current = q0$$

Finite states automate initially starts with current state q0.

Execute

$\Delta FSM;\ i?: INPUTS;\ o!: OUTPUTS$

$(current, i?) \in \text{dom}\ transition \land current' = transition(current, i?) \land o! = output(current, i?) \land transition' = transition \land output' = output$

Execution of finite state automata consists of application of functions transition and output to the input and the current state. Current state is modified in successful transition. Here is not defined what the automata will do is an unexpected input in this state is accepted. There are to possible actions: the first one is not to react and the state to remain the same; the second is to indicate an error. What approach will be used depend of automata nature. If the machine is grammar recognition it must react with error. If the automata is not of that that type it is possible simply to ignore the input and to remain in the same state. In any case, it is possible the automata to be represented with deterministic one and then transition function will be total and no such a problem will arise.

Finally, we will do some comments about finite state automata and business processes. There are two kinds of processes: such that are executed one time and finish and such that are started and then execute endless. In any case, they have to be initialized in some way that is why initial state is obligatory. But in the first case final states are obligatory. The business process cannot stop in another state (except in final states) – this is design error and has to be checked. Now, let's see examples. The first example:

STATES ::=q0 | patients | fields | setup | ready | beam_on

INPUTS ::= *select_patient* | *select_field* | *enter* | *ok* | *start* | *stop* | *intlk*

$FINALS: \mathbb{F}\ STATES$

$STATES \neq \varnothing \land (\exists n: \mathbb{N} \bullet \#STATES \leqslant n) \land INPUTS \neq \varnothing \land q0 \notin FINALS \land FINALS = \varnothing$

FSM

$transition: STATES \times INPUTS \nrightarrow STATES;\ current: STATES$

$transition = \{(q0, enter) \mapsto fields, (patients, enter) \mapsto fields, (fields, select\_patient) \mapsto patients,$
$(fields, enter) \mapsto setup, (setup, select\_patient) \mapsto patients, (setup, select\_field) \mapsto fields,$
$(setup, ok) \mapsto ready, \quad (ready, select\_patient) \mapsto patients, (ready, select\_field) \mapsto fields,$
$(ready, intlk) \mapsto setup, (ready, start) \mapsto beam\_on, (beam\_on, stop) \mapsto ready,$
$(beam\_on, intlk) \mapsto setup\ \} \land$
$q0 \in \text{dom}\ (\text{dom}\ transition) \land FINALS \cap \text{dom}\ (\text{dom}\ transition) = \varnothing \land FINALS \subseteq \text{ran}\ transition \land$
$STATES \setminus \{q0\} = \text{ran}\ transition \land STATES \setminus FINALS = \text{dom}\ (\text{dom}\ transition)$

FSMInit

FSM

$current = q0$

Execute

$\Delta FSM;\ i?: INPUTS$

$(current, i?) \in \text{dom}\ transition \land current' = transition(current, i?) \land transition' = transition$

OUTPUTS and output function are eliminated in this specification, but STATES, INPUTS and transition are in full details. No output parameter during execution.

The second example is:

$STATES ::= q0 \mid P0 \mid P1$

$INPUTS ::= init \mid i00 \mid i01 \mid i10 \mid i11$

$OUTPUTS ::= NULL \mid o0 \mid o1$

$FINALS: \mathbb{F}\ STATES$

$STATES \neq \varnothing \land (\exists n: \mathbb{N} \bullet \#STATES \leqslant n) \land INPUTS \neq \varnothing \land q0 \notin FINALS \land FINALS = \varnothing$

FSM

$transition: STATES \times INPUTS \rightarrow\!\!\!\!\!+\ STATES;\ output: STATES \times INPUTS \rightarrow\!\!\!\!\!+\ OUTPUTS$

$current: STATES$

$transition = \{(q0, init) \mapsto P0, (P0, i00) \mapsto P0, (P0, i01) \mapsto P0, (P0, i10) \mapsto P0,$
$(P0, i11) \mapsto P1, (P1, i00) \mapsto P0, (P1, i01) \mapsto P1, (P1, i10) \mapsto P1, (P1, i11) \mapsto P1\} \land$
$output = \{(q0, init) \mapsto NULL, (P0, i00) \mapsto o0, (P0, i01) \mapsto o1, (P0, i10) \mapsto o1, (P0, i11) \mapsto o1,$
$(P1, i00) \mapsto o1, (P1, i01) \mapsto o0, (P1, i10) \mapsto o0, (P1, i11) \mapsto o1\} \land$
$\text{dom}\ output = dom\ transition \land q0 \in \text{dom}\ (\text{dom}\ transition) \land FINALS \cap \text{dom}\ (\text{dom}\ transition) = \varnothing \land$
$FINALS \subseteq \text{ran}\ transition \land STATES \setminus \{q0\} = \text{ran}\ transition \land$
$STATES \setminus FINALS = \text{dom}\ (\text{dom}\ transition)$

FSMInit

FSM

$current = q0$

Execute

$\Delta FSM;\ i?: INPUTS;\ o!: OUTPUTS$

$(current, i?) \in \text{dom}\ transition \land current' = transition(current, i?) \land o! = output(current, i?) \land$
$transition' = transition \land output' = output$

Here, only final states are not defined.

INTO THE CSP

Let finite state machine is defined as follows: STATES = $\{q_0, q_1, \dots, q_n\}$ is the set of states; INPUTS = $\{i_1, i_2, \dots, i_m\}$ is the set of input symbols; OUTPUTS = $\{o_1, o_2, \dots, o_p\}$ is the set of output symbols; FINALS = $\{f_1, f_2, \dots, f_q\}$ is the set of final states. The communicating sequential process P modeling finite state machine is represented as a choice P = {x: B -> P(i)}, where B is the set of indexes of the states B = 0..n, and then P = {i: 0..n -> P(i)}. This process communicates with the environment via two channels *in* and *out*. The channels and process alphabets are α(in) = $\{ i_1, i_2, \dots, i_m\}$, α(out) = $\{o_1, o_2, \dots, o_p\}$, α(P) = α(in) U α(out). Every expression P(i) is represented by a process modeling finite state automata behavior in state qi: P(i) = Pi, i = 0, … , n. Let's see now what is Pi. If qi is a final state then Pi = SKIP. If q_i is not a final state then the transition function is defined for q_i and some subset of input events $\{i_{i1}, i_{i2}, \dots, i_{is}\}$. Let these transitions be: $(q_i, i_{ij}) \mapsto q_{ij}$ for j = 1, … , s. The subexpression for this transition is: in?i_{ij} -> Pij for j = 1, … , s. If the output is defined for this transition, i.e. $(qi, i_{ij}) \mapsto o_{ij}$, then this subexpression will be: in?i_{ij} -> out!o_{ij} -> Pij. The whole Pi is Pi = in? i_{i1} -> out!o_{i1} -> Pi1 | in?i_{i2} -> out!o_{i2} -> Pi2 | … | in?i_{is} -> out!o_{is} ->Pis Note: output communications are not defined for all transitions.

Now, let's see how this looks for the examples.

EXAMPLE 1 – HOSPITAL:

Hospital = {s: {q0, patients, fields, setup, ready, beam_on } -> Q(s)}

Qstart = in?enter -> Qfields; Qpatients = in?enter -> Qfields

```
Qfields = in?select_patient -> Qpatients | in?enter -> Qsetup
Qsetup = in?select_patient -> Qpatients | in?select_field -> Qfileds | in?on -> Qready
Qready = in?select_patient -> Qpatients | in?select_field -> Qfileds |
in?intkl -> Qsetup | in?start -> Qbeam_on
Qbeam_on = in?intkl -> Qsetup | in?stop -> Qready
EXAMPLE 2 – SUMMATOR:
Summator = {s: {q0, P0, P1} -> P(s)}; Pq0 = in?init -> Pp0
Pp0 = in?00 -> out!0 -> Pp0 | in?01 -> out!1 -> Pp0 | in?10 -> out!1 -> Pp0 |
in?11 -> out!0 -> Pp1
Pp1 = in?00 -> out!1 -> Pp0 | in?01 -> out!0 -> Pp1 | in?10 -> out!0 -> Pp1 |
in?11 -> out!1 -> Pp1
```

IMPLEMENTATION IN PAT 3

Process Analysis Toolkit [8] is an enhanced simulator, model checker and refinement checker for concurrent and real-time systems. It implements a version of CSP. Let's see our examples implemented in PAT 3.

EXAMPLE 1 – HOSPITAL:

```
enum {q0, patients, fields, setup, ready, beam_on};
enum {select_patient, select_field, enter, ok, start, stop, intlk, end};
#define error -1; channel in 0; channel out 0;
#alphabet Q{in.enter, in.select_patient, in.select_field, in.ok, in.intlk, in.start, in.stop,
in.end};
Q(state) = case {state == q0: in?enter -> Q(fields) [] in?end -> Skip
  state == patients: in?enter -> Q(fields) [] in?end -> Skip
  state == fields: in?select_patient -> Q(patients) [] in?enter -> Q(setup)
    [] in?end -> Skip
  state == setup: in?select_patient -> Q(patients) [] in?select_field -> Q(fields)
    [] in?ok -> Q(ready) [] in?end -> Skip
  state == ready: in?select_patient -> Q(patients) [] in?select_field -> Q(fields)
    [] in?intlk -> Q(setup) [] in?start -> Q(beam_on) [] in?end -> Skip
  state == beam_on: in?intlk -> Q(setup) [] in?stop -> Q(ready) [] in?end -> Skip
  default: out!error -> Stop};
System() = in!enter -> in!enter -> in!ok -> in!start -> in!end -> Skip ||| Q(q0);
#assert System() deadlockfree; #assert System() deterministic;
```

Here, events and states are defined as constants (named numbers) with enum. An event error is defined, because there is no other way of control on process parameters. If an error parameter is accepted by the process, then on out channel error event is sent. Input and output channels have to be defined not buffered. The alphabet of the process is restricted to the given one. The main difference is in the implementation of the process; instead for every choice alternative to be delivered as different process, process expressions are included directly in the choice operator. The choice is CSP is simply an operator defined on a set of events, but here in this implementation choice can be defined on process parameters. This idea processes to have parameters, is used in some versions of CSP, but not in the original representation. Finally, the system can be checked for many properties like deadlock free, determinism etc. These checks are put at the end of the specification and can be verified, but only processes without parameters can be verified, that is why such a process communicating with the machine is defined and checked.

EXAMPLE 2 – SUMMATOR:

```
enum {q0, P0, P1}; enum {initialize, i00, i01, i10, i11, end}; enum {o0, o1, error};
channel in 0; channel out 0;
#alphabet P{in.initialize, in.i00, in.i01, in.i10, in.i11, o.0, o.1};
P(state) = case {state == q0: in?initialize -> P(P0) [] in?end -> Skip
  state == P0: in?i00 -> out!o0 -> P(P0) [] in?i01 -> out!o1 -> P(P0)
```

```
   [] in?i10 -> out!1 -> P(P0) [] in?i11 -> out!o0 -> P(P1) [] in?end -> Skip
  state == P1: in?i00 -> out!o1 -> P(P0) [] in?i01 -> out!o0 -> P(P1)
   [] in?i10 -> out!0 -> P(P1) [] in?i11 -> out!o1 -> P(P1) [] in?end -> Skip
  default: out!error -> Stop};
System() = in!initialize -> in!i00 -> out?x -> in!end -> Skip ||| P(q0);
#assert System() deadlockfree; #assert System() deterministic;
```

In this example, output function is included. One more addition is that end event is added to stop the machine in every state. This process can be implemented with a global variable instead of process parameters, like that:

```
enum {q0, P0, P1}; enum {initialize, i00, i01, i10, i11, end}; enum {o0, o1, error};
channel in 0; channel out 0; var state = q0;
#alphabet P{in.initialize, in.i00, in.i01, in.i10, in.i11, o.0, o.1};
P() = case {state == q0: in?initialize -> {state = P0} -> P [] in?end -> Skip
  state == P0: in?i00 -> out!o0 -> {state = P0} -> P [] in?i01 -> out!o1 -> {state = P0} -> P
   [] in?i10 -> out!1 -> {state = P0} -> P [] in?i11 -> out!o0 -> {state = P1} -> P
   [] in?end -> Skip
  state == P1: in?i00 -> out!o1 -> {state = P0} -> P [] in?i01 -> out!o0 -> {state = P1} -> P
   [] in?i10 -> out!0 -> {state = P1} -> P
   [] in?i11 -> out!o1 -> {state = P1} -> P [] in?end -> Skip
  default: out!error -> Stop};
System() = in!initialize -> in!i00 -> out?x -> in!end -> Skip ||| P();
#assert P() deadlockfree; #assert P() deterministic;
```

But this implementation is not so clear and diverges from CSP. In above example, some properties are impossible to be checked, because the verifier is not sure about the contents of the global variable state. It finds out that the process is deterministic, but thinks that it is not deadlock free. Here, we will stop and will not go in further details what is possible and what is not to do with the tool.

ACKNOWLEDGMENTS

This research is supported by Project 240/2010 “Development of Grid infrastructure for research and education” funded by the Scientific Research Fund of University of Sofia.

REFERENCES

[1] IBM WebSphere Process Server and WebSphere Integration Developer, Version: V6.0.2, Business state machines, http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/index.jsp?topic=/com.ibm.iea.wpi_v6/wpswid/6.0.2/BusinessStateMachine.html
[2] Petri net, http://en.wikipedia.org/wiki/Petri_net
[3] Communicating sequential processes, http://en.wikipedia.org/wiki/Communicating_sequential_processes
[4] UML, http://www.uml.org
[5] OASIS Web Services Business Process Execution Language (WSBPEL) TC, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
[6] BPMN, http://www.bpmn.org/
[7] Z-notation, http://en.wikipedia.org/wiki/Z_notation
[8] Process Analysis Toolkit, http://www.comp.nus.edu.sg/~pat/

За контакти:

Доц. д-р Владимир Димитров, Катедра “Компютърна информатика”, Факултет по математика и информатика, СУ “Св. Климент Охридски”, тел.: 082-888 212, e-mail: cht@fmi.uni-sofia.bg

Докладът е рецензиран.