Modeling and Simulation of a Line Tracking Robot with RT-DEVS

Ali Salah Eddin

[aseddin@connect.carleton.ca](mailto:aseddin@connect.carleton.ca)

Abstract

Modeling and Simulation techniques are important in the development of complex projects. They help simplify the system development and analyze the system’s behavior before extensively developing and testing it. The DEVS formalism is a popular modeling and simulation technique. This report, presents how a Line Tracking Robot is modeled and simulated using the DEVS formalism. It presents DEVS formal specification of a component model of the whole system. It presents a state behavior of that component and its implementation in a computerized environment. Finally, a simulation of the complete robot’s model is run and the results are presented.

# Introduction

Developing a complex project is hard, exhaustive and takes long. It can be expensive to develop and test and hard to predict its outcome. For such reasons, system modeling and simulation (M&S) techniques were developed to facilitate experimenting on the system before or during its development, reduce high manufacturing costs of building prototypes and testing them, analyze the desired system at early stages to see if it would work properly or not, etc. One of the important M&S techniques developed is called Discrete EVent System specification (DEVS) [1] which aims at decomposing a complex system into small models that interconnect and follow a formal specification to interact between themselves and change states with regards to incoming events. In this report, the DEVS formalism is used to model and simulate a Line Tracking Robot’s behaviour before building it, which is flexible and not costly work.

# Background

## Discrete EVent System specification

DEVS is a formalism developed to model and simulate dynamic systems. It follows a rigorous rule set to define state changes of the modeled systems with regards to input events or time delay triggers. The DEVS formalism also defines a guideline to decompose complex system designs and precisely hierarchize them into basic models called *atomic* and composite models called *coupled* [2]. A coupled model is composed of a group of atomic and/or coupled models with well-defined coupling connections between its components.

A DEVS atomic model is formally defined by:

M = <X, Y, S, δint, δext, λ, ta>,

Where:

**X** = {(p,v) | p ∈ IPorts, v ∈ Xp} is the set of input ports and values;

**Y** = {(p,v) | p ∈ OPorts, v ∈ Yp} is the set of output ports and values;

**S**: is the set of sequential states;

**δint**: S **→** S is the internal state transition function;

**δext**: Q × X **→**S is the external state transition function, where:

**Q** = {(s,e) | s ∈ S, 0 < ∈ < ta(s)} is the total state set, e is the time elapsed since the last state transition;

**λ**: S **→**Y is the output function;

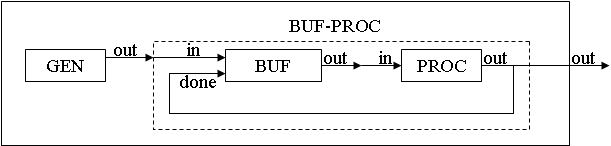
**ta**: S **→** R+0, *∞* is the time advance function.



**Figure 1:** Informal definition of an atomic model [2]

Figure 1 describes the above mentioned specification. An incoming input event x triggers an external transition δext in the atomic model. This transition is a function of the current state s, the input event x (set of ports and values) and elapsed time e since the last transition of the system. Based on the conditions of x and e, the external transition changes the state of the system to s’. Inside the system, there could be a time advance timer set to trigger when its value ta has elapsed since it was reset in the last transition. This trigger simultaneously causes an internal transition δint and output function λ, both functions of the current state s. The output function generates an output event y and the internal transition changes the current system state to a new state s’.

The DEVS specification of a coupled model is different from that of an atomic model as it doesn’t define system state transitions or output functions but instead defines the formal port connections and their directions between the components. The following figure 2 is a simple example of a coupled model hierarchy:



**Figure 2:** Generator-Buffer-Processor hierarchical DEVS model [3]

The Top model is a coupled model made of an atomic model GEN and a coupled model BUF-PROC, connected via the GEN::out 🡪BUF-PROC::in connection. BUF-PROC is connected to the Top model via the BUF-PROC::out 🡪 Top::out connection. BUF-PROC is also composed of two atomic models BUF and PROC who’s ports are interconnected as shown in the figure.

A DEVS coupled model is formally defined by:

CM = <X, Y, D, {Md|d ∈ D}, EIC, EOC, IC, select>

Where

**X** = {(p,v) |p ∈IPorts, v ∈Xp} is the set of input events, where IPorts represents the set of input ports and Xp represents the set of values for the input ports;

**Y** = {(p,v)|p ∈OPorts, v ∈Yp} is the set of output events, where OPorts represents the set of input ports and Yp represents the set of values for the output ports;

**D** is the set of the component names and for each d ∈D;

**Md** is a DEVS basic (i.e., atomic or coupled) model;

**EIC** is the set of external input couplings, EIC ⊆ { ( (Self, inSelf), (j, inj) )|inSelf ∈IPorts, j ∈D, inj ∈Iportsj};

**EOC** is the set of external output couplings, EOC ⊆ { ((i,outi),(Self,outSelf))|outSelf ∈OPorts, i∈D, outi∈OPortsi};

**IC** is the set of internal couplings, IC ⊆ { ((i,outi), (j,inj) ) |i,j ∈D, outi ∈OPortsi , inj ∈IPortsj };

**select** is the tiebreaker function, where **select** ⊆D🡪D, such that, for any nonempty subset E, **select** (E) ∈E.

## Embedded CD++

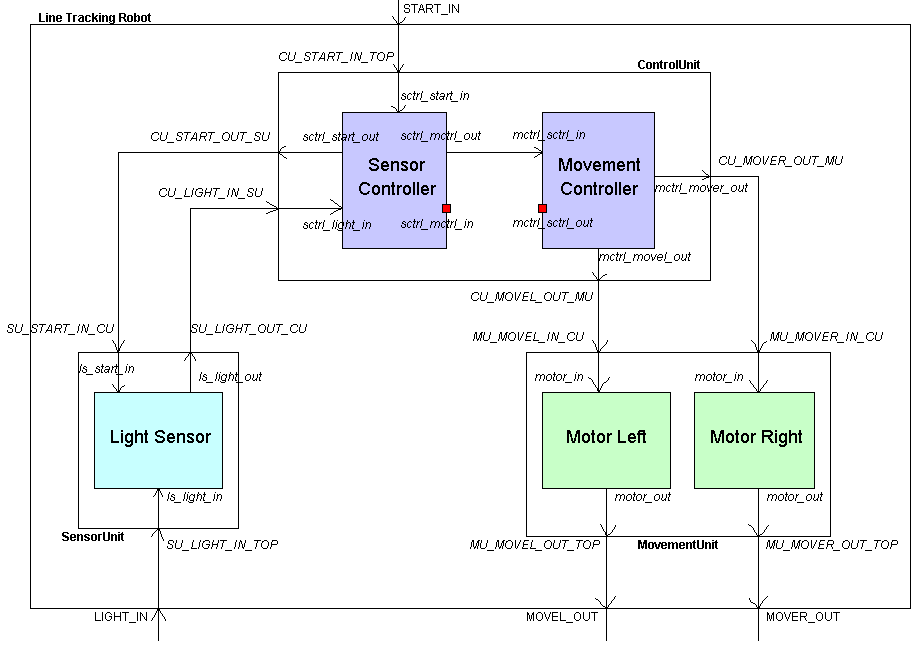
Embedded CD++ (ECD++) [4] is an extension to the CD++ [5] tool developed based on DEVS theory and its extensions [2]. It has converted the virtual time function of CD++ into a real-time function (using a time advance function tied to the real-time clock) and added hardware interaction capability to it. Model implementations are programmed in C++ code. The ECD++ tool is ran in a Linux environment and uses open source tools. It also has an Eclipse IDE plug-in to help with the ease of development and simulation visualization [3].

# Line Tracking Robot

The Line Tracking Robot can follow a line path on the ground from a starting point to a destination. It has a light sensor mounted on it and facing the ground. The light sensor absorbs the amount of light reflected off a small ground surface and translates it into a percentage value. The dark line path on the ground would occupy some of the reflecting bright surface thus decreasing the amount of light reflected into the sensor’s detection range. The robot’s controller considers a medium percentage of reflected light as a detected path and initiates the robot to move forward. In case the robot goes off track and doesn’t pick up a path trail, it stops and turns counter-clockwise slightly then tries to detect a trail again. If it finds a path it will move forward again, otherwise it will continue to turn until it finds a path to follow. The destination is considered as a very wide dark ground surface. At that point the light sensor would detect a small amount or no light reflection and the robot’s controller understands that it has reached a destination and stops moving. The robot can also receive manual signals to start and stop working.

## Top Model Overview

The following figure 3 illustrates the DEVS model hierarchy of the Line Tracking Robot. The input port are named as *To(port)\_*IN\_*From(model)* and the output port are named as *From(port)*\_OUT\_*To(model)* for coupled models, as for atomic models the right part of the name is undefined.



**Figure 3:** Line Tracking Robot model hierarchy diagram

The Line Tracking Robot’s top model is a couple model made of three coupled models: *Sensor Unit*, *Control Unit* and *Movement Unit*.

The *Sensor Unit* contains an atomic model: *Light Sensor*. It’s a collection of the robot’s sensors. It was implemented as a coupled model to enclose other sensors in the future, although it currently just contains one.

The Light Sensor reads the amount of light reflected off of a ground surface and transmits the readings to the *Control Unit* for processing.

The *Control Unit* contains two atomic models: *Sensor Controller* and *Movement Controller*. It is responsible for gathering and processing the information provided by the sensors and initiating the robot’s movement accordingly. It also receives a manual input to run and stop the sensors in the *Sensor Unit*.

The *Sensor Controller* receives the data from the *Light Sensor* and determines if the robot is on the right path or not. It sends its decision as a signal to the *Movement Controller*. The *Sensor Controller* is the component inside the *Controller Unit* that sends the manual run/stop signals to the *Sensor Unit*. It also understands if the robot has reached a destination and sends a signal to stop the *Movement Controller* and the *Sensor Unit* to stop.

The *Movement* *Controller* receives signals from the *Sensor* *Controller* telling it whether the robot is on track or not. Based on the signal received, the *Movement* *Controller* decides whether to signal the *Motor* models to move forward, stop, or turn until the *Sensor* *Controller* tells it that the robot is back on track.

The *Movement Unit* is made of two atomic models: *Motor Left* and *Motor Right*. It’s a collection of the robot’s actuators that move in response to commands received from the *Control Unit*.

The *Motor* models control the functions of the robot treads. They can only move forward, in reverse or stop according to the signals they receive from the *Control Unit*. A combination of a motor moving forward and the other motor moving in reverse makes the robot turn around itself.

## DEVS Model Specification

The DEVS formal specification of the *Sensor Controller* model is, for example, as follows:

M = <X, S, Y, δext, δint, λ, ta>,

Where

**X**: (sctrl\_light\_in, {BRIGHT, DARK, ALL\_DARK}) Connected to *Light Sensor* light detection signal. (sctrl\_start\_in, {START\_PROC, STOP\_PROC}) connected to the global system’s manual start/stop signal. (sctrl\_mctrl\_in, {Ø}) currently not connected, but made available for future possible use.

**S**: ”IDLE”, “PREP\_RX”, “WAIT\_DATA”, “TX\_DATA”, “PREP\_STOP”.

**Y**: (sctrl\_mctrl\_out, {ON\_TRACK, OFF\_TRACK, STOP\_PROC}) Connected to the *Movement Controller* input port.

(sctrl\_start\_out, {START\_PROC, STOP\_PROC}) connected to *Light Sensor* start/stop input port.

**δext**: Receives inputs from the input port and initiates appropriate state transitions.

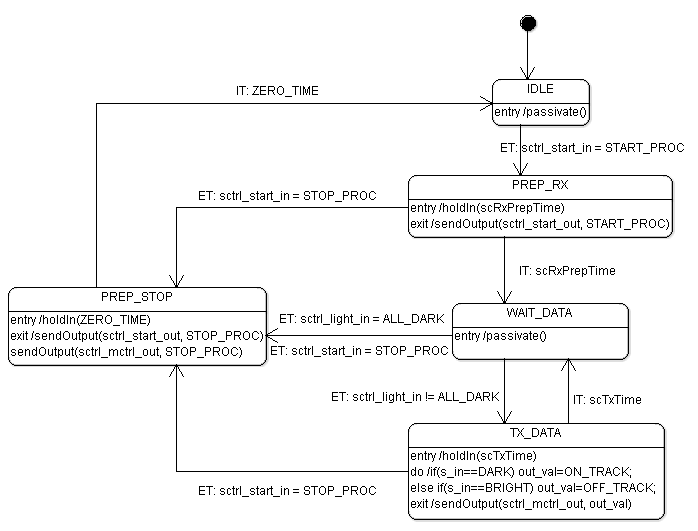
**δint**: defines state changes based on current state.

**λ**: based on the input value and the current state sends the output signals to the output ports.

**ta**: real-time advance function.

DEVS formal specifications of the other models are listed in Appendix A.

The following figure 4 illustrates the state transitions of the *Sensor Controller* model:



**Figure 4:** *Sensor* *Controller* state diagram

The *Sensor* *Controller* starts by being in the IDLE state for an infinite time. If a manual start signal is received, it triggers an external transition that changes its state to PREP\_RX. At this stage, it waits for a time advance ta=scRxPrepTime after which a ‘start’ output signal is sent to the *Light* *Sensor* and an internal transition is triggered changing its state to WAIT\_DATA. The *Sensor* *Controller* waits in this state until it receives a signal from the *Light* *Sensor*. When a signal is received, if the signal indicates that the robot reached a destination (signal value is ALL\_DARK), an external transition moves the *Sensor* *Controller* to the PREP\_STOP state, at which it will immediately send a stop signal to the *Light* *Sensor* and the *Movement* *Controller* and then transition back to the IDLE state. However, if the received signal was different, the *Sensor* *Controller* would go to the TX\_DATA state at which it will wait for a time advance period of ta=scTxTime before it sends an output signal to the *Movement* *Controller* indicating whether the robot is on track or not, and transition back to the WAIT\_DATA state. At any point in time, if the *Sensor* *Controller* receives a manual stop signal (STOP\_PROC), it will execute an external transition to the PREP\_STOP state to stop all activities.

State diagrams of the other models are listed in Appendix B.

## Implementation in ECD++

ECD++ provides a mechanism to program DEVS models’ hierarchical structures. The model definitions and couplings are written in a model file following a specific format and the state transitions and output function are overwritten in C++, as part of each model’s class definition.

The following code describes the *Sensor* and *Movement* *Controllers* specification as components of the *Control* *Unit* in the model file, in accordance with the model diagram in figure 3:

1. [ControlUnit]
2. components : SCtrl@SensorController MCtrl@MovementController
3. in : CU\_START\_IN\_TOP CU\_LIGHT\_IN\_SU
4. out : CU\_START\_OUT\_SU CU\_MOVER\_OUT\_MU CU\_MOVEL\_OUT\_MU
5. %input connections
6. Link : CU\_START\_IN\_TOP sctrl\_start\_in@SCtrl
7. Link : CU\_LIGHT\_IN\_SU sctrl\_light\_in@SCtrl
8. %output connections
9. Link : sctrl\_start\_out@SCtrl CU\_START\_OUT\_SU
10. Link : mctrl\_moveR\_out@MCtrl CU\_MOVER\_OUT\_MU
11. Link : mctrl\_moveL\_out@MCtrl CU\_MOVEL\_OUT\_MU
12. %internal connections
13. Link : sctrl\_mctrl\_out@SCtrl mctrl\_sctrl\_in@MCtrl

The code snippet starts by defining the Control Unit as a coupled model composed of two instances: *SCtrl* and *MCtrl*, of *Sensor* and *Movement* *Controller* respectively. Then, the code defines the input and output ports of the *Control* *Unit*. Finally, the code describes the input and output connections between the ports of the *Control* *Unit* and those of *SCtrl* and *MCtrl*, as well as the internal connections between *SCtrl* and *MCtrl*. The direction of the connection is read as FROM port 🡪 TO port.

The following code describes the transition and output functions of the *Sensor* *Controller*, in accordance with the state diagram in figure 4:

1. Model &SensorController::externalFunction( const ExternalMessage &msg ) {
2. …
3. if (msg.port() == sctrl\_light\_in){
4. if(state == WAIT\_DATA) {
5. sensor\_input= static\_cast<int>(msg.value());
6. if(sensor\_input==ALL\_DARK) {
7. state=PREP\_STOP;
8. holdIn(Atomic::active, ZERO\_TIME );
9. } else {
10. state = TX\_DATA;
11. holdIn(Atomic::active, scTxTime );
12. }
13. }
14. }
15. return \*this;
16. }
17. Model &SensorController::internalFunction( const InternalMessage & ) {
18. switch (state){
19. …
20. case TX\_DATA:
21. state = WAIT\_DATA;
22. passivate();
23. break;
24. }
25. return \*this;
26. }
27. Model &SensorController::outputFunction( const InternalMessage &msg ) {
28. switch (state){
29. …
30. case TX\_DATA: {
31. int output\_val;
33. if(sensor\_input==DARK)
34. output\_val = ON\_TRACK;
35. Else if(sensor\_input==BRIGHT)
36. output\_val = OFF\_TRACK;
37. sendOutput(msg.time(),sctrl\_mctrl\_out, output\_val);
38. break;
39. }
40. };
41. return \*this ;
42. }

The code snippet first shows a portion of the external transition function that describes the transition from state WAIT\_DATA to either TX\_DATA or PREP\_STOP depending on the value (sensor\_input) of the incoming signal from the *Light* *Sensor* received on port sctrl\_light\_in. Lines 18 to 28 show a portion of the internal transition function describing the transition from TX\_DATA to WAIT\_DATA. Finally, lines 29 to 43 show a portion of the output function’s behaviour at state TX\_DATA. The output function sets the output signal (ON\_TRACK or OFF\_TRACK) to send to the *Movement* *Controller* through port sctrl\_mctrl\_out.

## Simulation Results

The implementation of the Line Tracking Robot was simulated in ECD++ using its simulation framework for DEVS models. The following table 1 shows the simulation results accompanied with descriptive comments:

**Table 1:** Results of Line Tracking Robot’s model simulation in ECD++

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Description** |
| 1. 00:00:01:000 START\_IN START | - | System START - no output |
| 1. 00:00:02:000 LIGHT\_IN DARK | 00:00:02:200:000 mover\_out Go\_FWD | Path detected –motors go forward |
|  | 00:00:02:200:000 movel\_out Go\_FWD |  |
| 1. 00:00:02:500 LIGHT\_IN BRIGHT | 00:00:02:600:000 mover\_out STOP | OFF Track signal – motors stop |
|  | 00:00:02:600:000 movel\_out STOP |  |
|  | 00:00:02:700:000 mover\_out GO\_FWD | Robot Turns – motor\_right forward |
|  | 00:00:02:700:000 movel\_out GO\_REV | and motor\_left reverse |
| 1. 00:00:02:700 START\_IN STOP | 00:00:02:700:000 mover\_out STOP | Manual System STOP - Turn interrupted |
|  | 00:00:02:700:000 movel\_out STOP |  |
| 1. 00:00:03:000 LIGHT\_IN DARK | - | Ignored – System STOPPED |
| 1. 00:00:03:500 LIGHT\_IN BRIGHT | - | Ignored – System STOPPED |
| 1. 00:00:05:000 START\_IN START | - | System START - no output |
| 1. 00:00:05:500 LIGHT\_IN BRIGHT | 00:00:05:700:000 mover\_out Go\_FWD | OFF Track signal - Robot Turns |
|  | 00:00:05:700:000 movel\_out Go\_REV |  |
| 1. 00:00:06:000 LIGHT\_IN BRIGHT | - | Ignored - Still Turning |
| 1. 00:00:06:500 LIGHT\_IN DARK | - | Ignored - Still Turning |
| 1. - | 00:00:06:650:000 mover\_out STOP | Turn complete – motors stop |
|  | 00:00:06:650:000 movel\_out STOP |  |
| 1. 00:00:07:000 LIGHT\_IN DARK | 00:00:07:200:000 mover\_out Go\_FWD | Path detected –motors go forward |
|  | 00:00:07:200:000 movel\_out Go\_FWD |  |
| 1. 00:00:07:500 LIGHT\_IN DARK | - | Redundant - motors still moving forward |
| 1. 00:00:08:000 LIGHT\_IN ALL\_DARK | 00:00:08:050:000 mover\_out STOP | Reached Destination - motors stop |
|  | 00:00:08:050:000 movel\_out STOP |  |
| 1. 00:00:08:500 LIGHT\_IN DARK | - | Ignored - STOPPED |
| 1. 00:00:09:000 LIGHT\_IN DARK | - | Ignored - STOPPED |
| 1. 00:00:09:300 START\_IN STOP | 00:00:09:300:000 mover\_out STOP | Manual System STOP |
|  | 00:00:09:300:000 movel\_out STOP |  |

Appendix C contains a sequence diagram describing lines 1 and 2 of the simulation results in table 1 above.

# Conclusion

This report demonstrated a DEVS model of a Line Tracking Robot. It showed how the system was decomposed into several atomic and coupled models connected via a well define hierarchical scheme, where simply the Robot consisted of three main components: Sensor Unit, Control Unit and Movement Unit. The Sensor Unit receives light reflection readings then sends them to the Control Unit. The Control Unit analyses the data received and determines in whether the Robot is on a valid path or not and sends movement signals to the Movement Unit accordingly. The Control Unit tells the Movement Unit to either move forward, turn or stop. The report also showed how the Sensor Controller, as an example, was modeled according to DEVS formal specification. It also presented a state diagram of the Sensor Controller and a matching code implementation snippet written in ECD++. Finally, a simulation of the implemented model was run in ECD++ and the results were presented in a table. The results were satisfactory and were as expected from the model’s behavior. This work didn’t involve building the Line Tracking Robot and deploying it. This will be considered as future work to what was currently presented.

## References

[1] - Zeigler, B. P. 1976. *Theory of modeling and simulation.* New York: Wiley-Interscience.

[2] – Wainer, G. “Discrete-Event Modeling and Simulation, a practitioner’s approach”.

[3] – Moallemi, M; Wainer, G; Awad A.; Tall, D. “Application of RT-DEVS in Military.

[4] - YU, J.; WAINER, G. “E-CD++: a tool for modeling embedded applications”. In Proceedings of the 2007 SCS Summer Computer Simulation Conference. San Diego, CA. 2007.

[5] - Wainer, G. "CD++: a toolkit to define discrete-event models". Software, Practice and Experience. Wiley. Vol. 32, No.3. pp. 1261-1306. November 2002

# Appendix A

The DEVS formal specification of the *Light Sensor* model is as follows:

M = <X, S, Y, δext, δint, λ, ta>,

Where

**X**: (ls\_light\_in, {BRIGHT, DARK, ALL\_DARK}).

(ls\_start\_in, {START\_PROC, STOP\_PROC}).

**S**: ”IDLE”, “DETECTING”, “WAIT\_DATA”, “TX\_DATA”.

**Y**: (ls\_light\_out, { BRIGHT, DARK, ALL\_DARK}).

**δext**: Receives inputs from the input port and initiates appropriate state transitions.

**δint**: defines state changes based on current state.

**λ**: based on the input value and the current state sends the output signals to the output ports.

**ta**: real-time advance function.

The DEVS formal specification of the *Movement Controller* model is as follows:

M = <X, S, Y, δext, δint, λ, ta>,

Where

**X**: (mctrl\_sctrl\_in, {ON\_TRACK, OFF\_TRACK, STOP\_PROC})

**S**: ”IDLE”, “PREP\_MOVE\_FWD”, “MOVE\_FWD”, “MX\_STOP”, “PREP\_TURN”, “TURN\_ALPHA”, “WAIT\_DATA”, “PREP\_STOP”.

**Y**: (mctrl\_mover\_out, {O\_GO\_FWD, O\_GO\_REV, O\_STOP})

(mctrl\_movel\_out, { O\_GO\_FWD, O\_GO\_REV, O\_STOP })

(mctrl\_sctrl\_out, {Ø}) currently not connected, but made available for future possible use.

**δext**: Receives inputs from the input port and initiates appropriate state transitions.

**δint**: defines state changes based on current state.

**λ**: based on the input value and the current state sends the output signals to the output ports.

**ta**: real-time advance function.

The DEVS formal specification of the *Motor* models is as follows:

M = <X, S, Y, δext, δint, λ, ta>,

Where

**X**: (motor\_in, { O\_GO\_FWD, O\_GO\_REV, O\_STOP })

**S**: ”STOP”, “PREP\_MOVE\_FWD”, “PREP\_MOVE\_REV”, “MOVE\_FWD”, “MOVE\_REV”, “PREP\_STOP”.

**Y**: (motor\_out, { O\_GO\_FWD, O\_GO\_REV, O\_STOP })

**δext**: Receives inputs from the input port and initiates appropriate state transitions.

**δint**: defines state changes based on current state.

**λ**: based on the input value and the current state sends the output signals to the output ports.

**ta**: real-time advance function.

The DEVS formal specification of the *Line Tracking Robot* model is as follows:

CM = < X, Y, D, {Md}, EIC, EOC, IC, select >,

Where

**X**: (START\_IN, N)

(LIGHT\_IN, N)

**Y**: (MOVEL\_OUT, N)

(MOVER\_OUT, N)

**D**: {Sensor Unit, Control Unit, Movement Unit}.

**Md**: {M(sensor unit), M(control unit), M(movement unit)}

**EIC**: { (Self, START\_IN), (Control Unit, CU\_START\_IN\_TOP) }

{ (Self, LIGHT\_IN), (Sensor Unit, SU\_LIGHT\_IN\_TOP) }

**EOC**: { (Movement Unit, MU\_MOVEL\_OUT\_TOP), (Self, MOVEL\_OUT) }

{ (Movement Unit , MU\_MOVER\_OUT\_TOP), (Self, MOVER\_OUT) }

**IC**: { (Sensor Unit, SU\_LIGHT\_OUT\_CU), (Control Unit, CU\_LIGHT\_IN\_SU) }

{ (Control Unit , CU\_START\_OUT\_SU), (Sensor Unit, SU\_START\_IN\_CU) }

{ (Control Unit , CU\_MOVEL\_OUT\_MU), (Movement Unit , MU\_MOVEL\_IN\_CU) }

{ (Control Unit , CU\_MOVER\_OUT\_MU), (Movement Unit , MU\_MOVER\_IN\_CU) }

**select**: {Sensor Unit, Control Unit, Movement Unit}.

The DEVS formal specification of the *Sensor Unit* model is as follows:

CM = < X, Y, D, {Md}, EIC, EOC, IC, select >,

Where

**X**: (SU\_START\_IN\_CU, N)

(SU\_LIGHT\_IN\_TOP, N)

**Y**: (SU\_LIGHT\_OUT\_CU, N)

**D**: {Light Sensor}.

**Md**: {M(light sensor)}

**EIC**: { (Self, SU\_START\_IN\_CU), (Light Sensor, ls\_start\_in) }

{ (Self, SU\_LIGHT\_IN\_TOP), (Light Sensor, ls\_light\_in) }

**EOC**: { (Light Sensor, ls\_light\_out), (Self, SU\_LIGHT\_OUT\_CU) }

**IC**: { Ø }

**select**: { Light Sensor }.

The DEVS formal specification of the *Control Unit* model is as follows:

CM = < X, Y, D, {Md}, EIC, EOC, IC, select >,

Where

**X**: (CU\_START\_IN\_TOP, N)

(CU\_LIGHT\_IN\_SU, N)

**Y**: (CU\_START\_OUT\_SU, N)

(CU\_MOVEL\_OUT\_MU, N)

(CU\_MOVER\_OUT\_MU, N)

**D**: {Sensor Controller, Movement Controller}.

**Md**: {M(sensor controller), M(movement controller)}

**EIC**: { (Self, CU\_START\_IN\_TOP), (Sensor Controller, sctrl\_start\_in) }

{ (Self, CU\_LIGHT\_IN\_SU), (Sensor Controller, sctrl\_light\_in) }

**EOC**: { (Sensor Controller, sctrl\_start\_out), (Self, CU\_START\_OUT\_SU) }

{ (Movement Controller, mctrl\_movel\_out), (Self, CU\_MOVEL\_OUT\_MU) }

{ (Movement Controller, mctrl\_mover\_out), (Self, CU\_MOVER\_OUT\_MU) }

**IC**: { (Sensor Controller, sctrl\_mctrl\_out), (Movement Controller, mctrl\_sctrl\_in) }

**select**: { Sensor Controller, Movement Controller }.

The DEVS formal specification of the *Movement Unit* model is as follows:

CM = < X, Y, D, {Md}, EIC, EOC, IC, select >,

Where

**X**: (MU\_MOVEL\_IN\_CU, N)

(MU\_MOVER\_IN\_CU, N)

**Y**: (MU\_MOVEL\_OUT\_TOP, N)

(MU\_MOVER\_OUT\_TOP, N)

**D**: {Motor Right, Motor Left}.

**Md**: {M(motor right), M(motor left)}

**EIC**: { (Self, MU\_MOVEL\_IN\_CU), (Motor Left, motor\_in) }

{ (Self, MU\_MOVER\_IN\_CU), (Motor Right, motor\_in) }

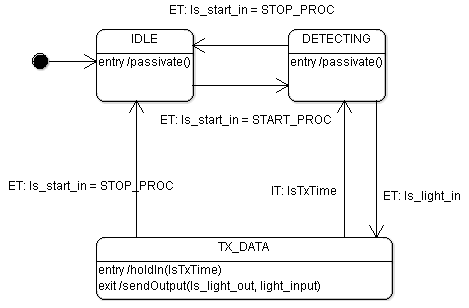
**EOC**: { (motor\_out, Motor Left), (Self, MU\_MOVEL\_OUT\_TOP) }

{ (motor\_out, Motor Right), (Self, MU\_MOVER\_OUT\_TOP) }

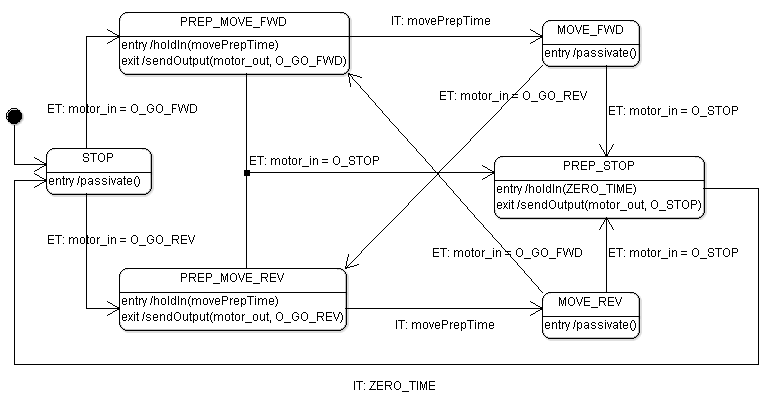
**IC**: { Ø }

**select**: { Motor Right, Motor Left }.

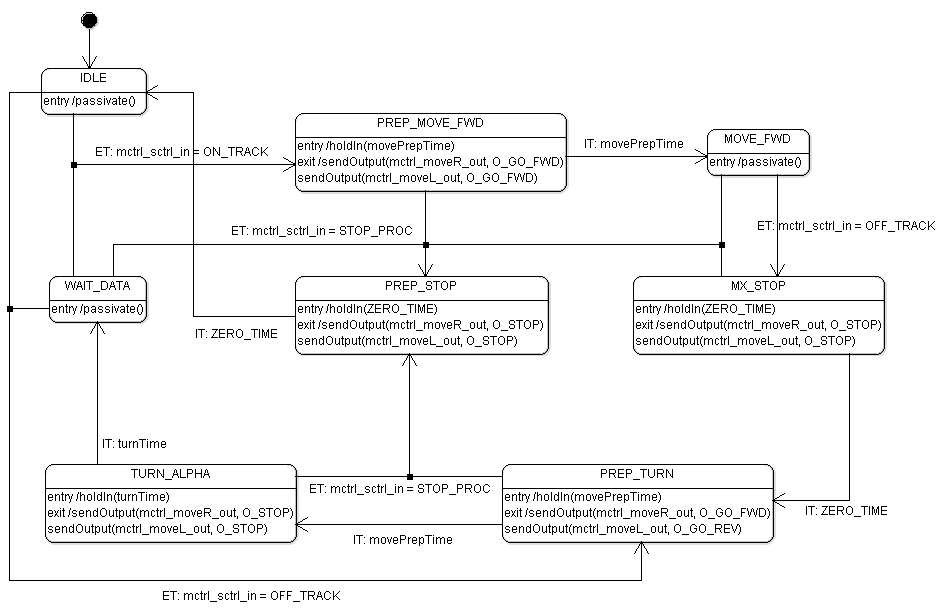
# Appendix B



**Figure B.1:** Light Sensor state diagram

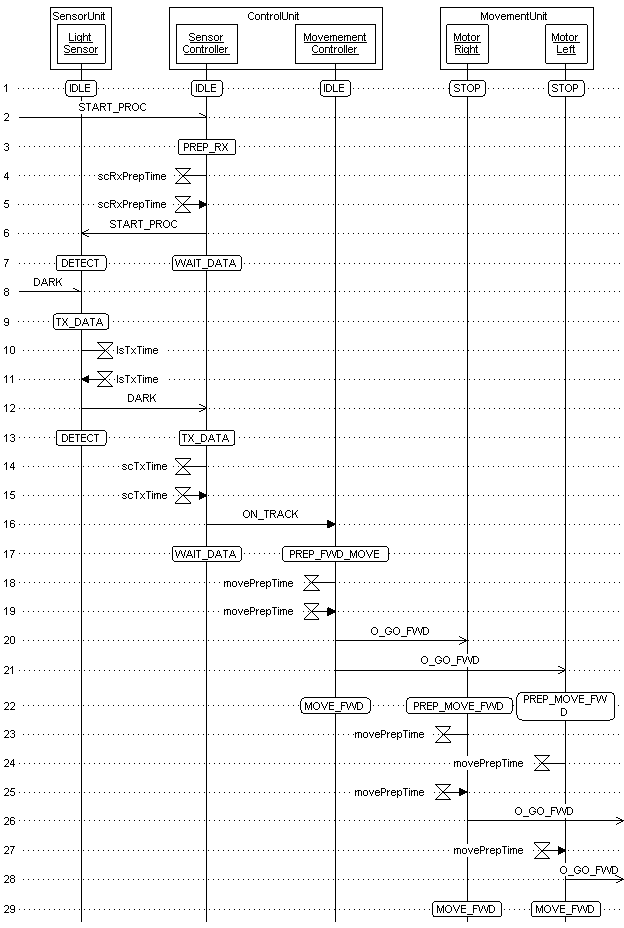


**Figure B.2:** Motor state diagram

****

**Figure B.3:** Movement Controller state diagram

# Appendix C



**Figure C.1:** Sequence diagram of lines 1 and 2 of Table 1