A Visual Formalism for Interacting Systems

Paul C. Jorgensen
School of Computing and Information Systems
Grand Valley State University
Allendale, Michigan USA
jorgensp@gvsu.edu

Interacting systems are increasingly commonmany examples pervade our everyday lives: automobiles, aircraft, defense systems, telephone switching systems, financial systems, national governments, and so on. Closer to computer science, embedded systems and Systems of Systems are further examples of interacting systems. Common to all of these is that some ”whole” is made up of constituent parts, and these parts interact with each other. By design, these interactions are intentional, but it is the unintended interactions that are problematic. The Systems of Systems literature uses the terms ”constituent systems” and ”constituents” to refer to systems that interact with each other. That practice is followed here.

This paper presents a visual formalism, Swim Lane Event-Driven Petri Nets, that is proposed as a basis for Model-Based Testing (MBT) of interacting systems. In the absence of available tools, this model can only support the offline form of Model-Based Testing.

1 Existing Models for Interacting Systems

To support offline MBT of interacting systems, a model must be capable of expressing the ways in which constituents interact. The best known, and most widely used, such model is the Statechart model [3]. Statecharts have been incorporated into the Unified Modeling Language (UML) and further codified into Types, I, II, and III of UML Statecharts. The model presented in this paper has been shown to be formally equivalent [2] to all three types of UML statecharts.

Statecharts contain orthogonal regions, and these nicely represent distinct devices. As such, they can also be used to represent systems that interact with each other. The statechart broadcasting mechanism is the only vehicle for communicating interactions among the orthogonal devices/components. There is an elaborate language on transitions among blobs in an orthogonal region. Taken together, these notations result in a very ”dense” model that is best understood by executing the statechart with an engine. There can be no doubt about the expressive power of statecharts. In a conversation [5] at a Grand Rapids (Michigan) avionics company, representatives of the i-Logix company related a success story in which exhaustive execution of a statechart model of a fully-deployed ballistic missile launch control system revealed a legitimate sequence of events that would launch a missile that was not known to the developing defense contractor.

Contemporaneously with Harel’s work on statecharts, a North American industry group proposed the Extended Systems Modeling Language (ESML) which described how activities in a traditional data flow diagram could communicate with each other [1]. The ESML prompts allowed an activity in a data flow diagram to Enable, Disable, Activate, Pause, Resume, Suspend, or Trigger another activity. Synonyms of many of these verbs are available in the statechart transition language.

A.K. Petrenko, B.-H. Schlingloff, N. Pakulin (Eds.):
Tenth Workshop on Model-Based Testing (MBT 2015)
EPTCS 180, 2015, pp. 41–55, doi:10.4204/EPTCS.180.3

2 Comparing Statecharts with Swim Lane Event-Driven Petri Nets

While they are clearly a powerful modeling technique, there are some problems with statecharts (as originally defined). Statecharts:

- Are best understood when executed by a customer using a statechart engine,
- Are a top-down model,
- Can only be composed under very limited circumstances [4], and
- Are both rigorous and complex, a five-day training course is recommended [6].

These limitations are all answered by Swim Lane Event-Driven Petri Nets, specifically they are:

- intuitively clear, once the basic mechanism of Petri Net transition firing is understood,
- a bottom-up model. As such, they work well in agile developments.
- easily composed, particularly if the composition is accomplished in a database with well-designed queries.
- easy to learn

3 Event-Driven Petri Nets and Swim Lane Petri Nets

Basic Petri nets need two slight enhancements to become Event-Driven Petri Nets (EDPNs) [7]. The first enables them to express more closely event-driven systems, and the second deals with Petri net markings that express event quiescence, an important notion in object-oriented applications. Taken together, these extensions result in an effective, operational view of software requirements.

Definition: An Event-Driven Petri Net (EDPN) is a tripartite-directed graph (P, D, S, In, Out) composed of three sets of nodes, P, D, and S, and two mappings, In and Out, where:

- P is a set of port events
- D is a set of data places
- S is a set of transitions
- In is a set of ordered pairs from $(P \cup D) \times S$
- Out is a set of ordered pairs from $S \times (P \cup D)$

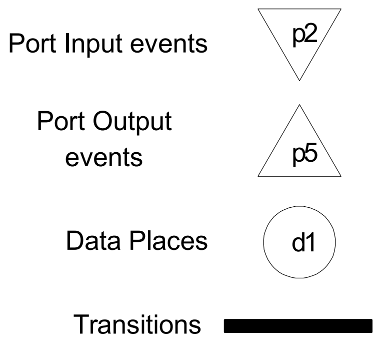

Figure 1: Event-Driven Petri Net elements

The drawing conventions for EDPNs are in Figure 1. Other than explicit representation of discrete events, EDPNs are very similar to ordinary Petri nets. The other main difference is that an ordinary Petri net is a closed system, in which tokens and markings are determined only by transition firing. EDPNs support a concept of event quiescence, which is an extension to deadlock in an ordinary Petri net. Since events are (usually) from external devices, an EDPN more accurately represents an event-driven system (so EDPNs represent ”open” systems.) Both ordinary and Event-Driven Petri Nets can be placed into UML-style ”swim lanes.” The most convenient interpretations of swim lanes is that they ”contain” interacting constituent systems.

4 Communication Primitives for Interacting Systems

Integration testing for a single system is based on the assumption that the units being integrated have all been individually, and thoroughly, tested. The usual goals of integration testing are to find faults that are due to interfaces among the units, and as such, would not be revealed by unit testing. This clearly extends to testing interacting systems where we assume that the constituent systems are all thoroughly tested and function correctly. The goal of testing for interacting systems is to focus on the ways in which constituent systems communicate; here we present a set of communication primitives for that purpose. The first two are from ordinary Petri nets, followed by the communication primitives from the Extended Systems Modeling Language (ESML) [1]. The ESML primitives are supplemented by three primitives that represent service requests among interacting systems. Each primitive is briefly described and illustrated by accompanying figures next.

4.1 Petri Net Conflict

Figure 2 shows the conflict pattern of ordinary Petri nets. With the given marking, both transitions are enabled. Firing either one disables the other. In EDPNs, if place p2 is replaced by a port input event, we have a context-sensitive input event, where outputs of the same physical input depend on the context in which the event occurs.

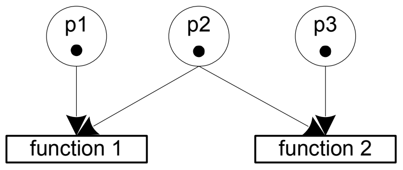

Figure 2: Traditional Petri Net conflict

4.2 Petri Net Interlock

The Petri net interlock pattern is used to set a priority. With the marking shown in Figure 3, the secondary action cannot fire until the preferred action fires. If these are linked, we have the pattern for mutual exclusion.

4.3 ESML Enable, Disable, and Activate

The work of the ESML committee began with activities in a data flow diagram that could control other activities. Here the place labeled ”e/d” is used to enable, and later, to disable the controlled action in Figure 4. Clearly the enabling and disabling actions are from other constituents. Once a transition is enabled, it remains enabled due to the output leading back to the e/d place. Notice that the enable/disable

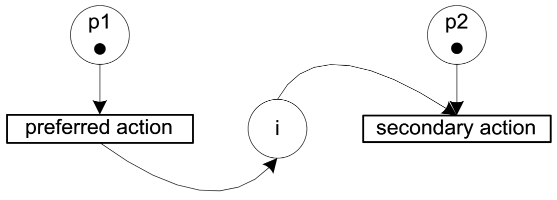

Figure 3: Traditional Petri Net interlock

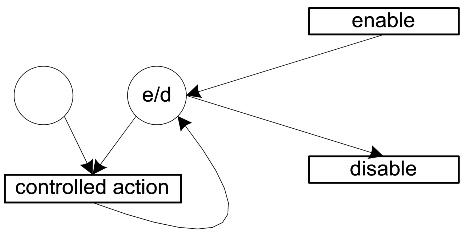

Figure 4: ESML Enable, Disable, and Activate

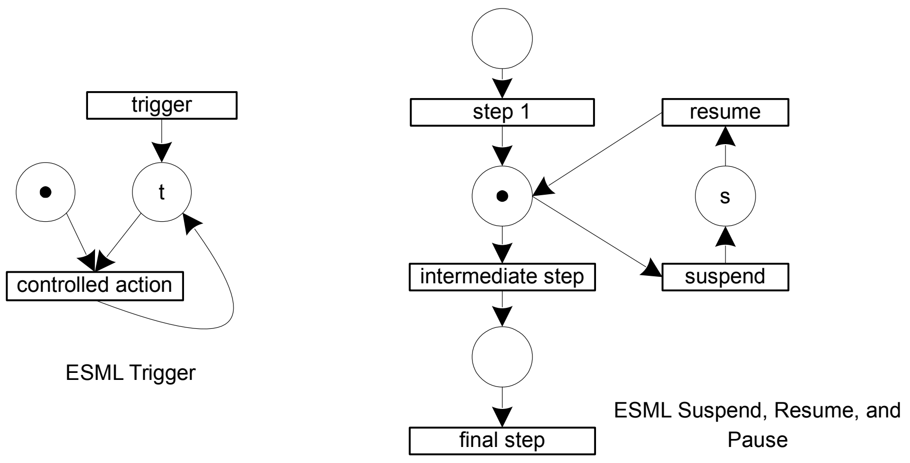

Figure 5: ESML Trigger, Suspend, Resume, and Pause

place establishes Petri net conflict between the controlled and disable actions. The enable portion essentially gives permission for the controlled action to occur, but in terms of marking, it may have to await additional inputs. The edge from the controlled action back to the enable/disable place assures that an enabled transition remains enabled. The activate prompt is simply a sequence of enable followed by a disable.

4.4 ESML Trigger

The ESML trigger prompt is stronger than the enable prompt. It essentially requires the controlled action to fire as soon is it is fully enabled, as in Figure 5. A trigger prompt can be paired with a disable prompt. As with the Enable prompt, once a controlled action has been started by a Trigger Prompt, the Trigger place (marked t in Figure 5) remains marked due to the output that leads back to it. (Of course, this assumes that the other inputs are still available to the controlled action. As with the Enable prompt, a Trigger prompt can be removed by a Disable prompt.

4.5 ESML Suspend, Resume, and Pause

The ESML suspend, resume, and pause prompts (Figure 5) were designed to interact with an ongoing activity without losing any of the work done prior to the point of suspension. The suspend place “s” is actually an interlock to assure that the resume action must follow a suspend action. The suspend action can be used to transfer temporary control to a more important action. As with activate, the pause prompt is a sequence of suspend followed by resume.

4.6 Service Requests

The last three patterns are directed at communication among constituents. When constituent A requests a service from constituent B, the request place is similar to an enable prompt (Figure 6). Another interpretation is that a request is a message. Once a request is made, constituent A awaits a response from constituent B.

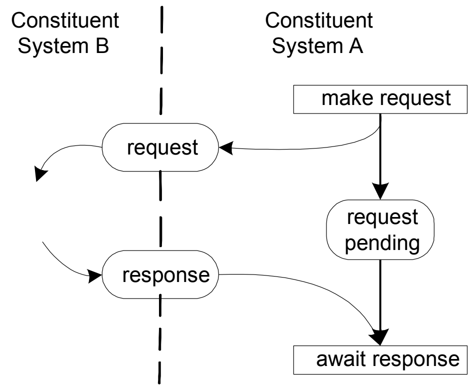

Figure 6: SoS Request

4.6.1 Accept and Reject Requests

On receipt of a request, constituent B may choose to accept or reject the request. When the service has been provided, constituent B returns a ”done” response to the waiting constituent A (see Figure 8). Similarly, if constituent B rejects the request, constituent B returns a ”not done” response to the waiting constituent. Notice that the request place is in Petri net conflict with respect to the two responses.

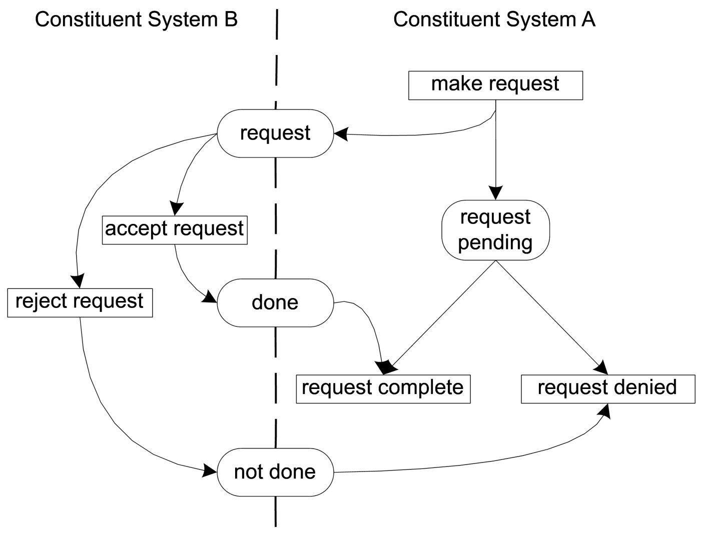

Figure 7: SoS Accept and Reject responses to a Request

4.6.2 Postpone Request

Consider situations in which a constituent may have the latitude to postpone a request. This most likely happens because the constituent has more urgent tasks. Note the use of an interlock to show the task priority in Figure 8.

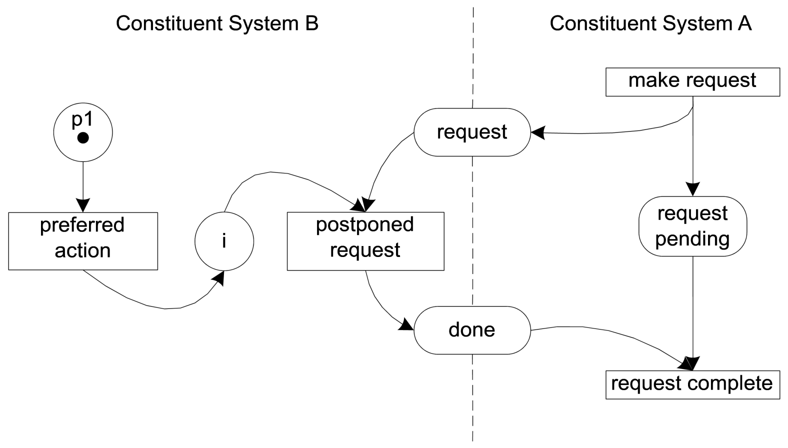

Figure 8: SoS Postpone

5 Example: a Garage Door Controller

A system to control a motorized garage door is comprised of several components: a drive motor, the garage door wheel tracks, and a wireless control device. There are two safety features, a light beam near the floor, and an obstacle sensor. These latter two devices operate only when the garage door is closing. If the light beam is interrupted (possibly by a pet) the door immediately stops, and then begins to open. Similarly, if the door encounters an obstacle while it is closing (say a childs bicycle left in the path of the door), the door stops and reverses direction as with a light beam interruption. There is a third way to stop a door in motion, either when it is closing or openinga signal from the wireless control device. In response to this signal the door stops in place. A subsequent signal starts the door in the same direction as when it was stopped. Finally, there are sensors that detect when the door has moved to one of the extreme positions, either fully open or fully closed. Figure 9 is a SysML context diagram of the garage door controller.

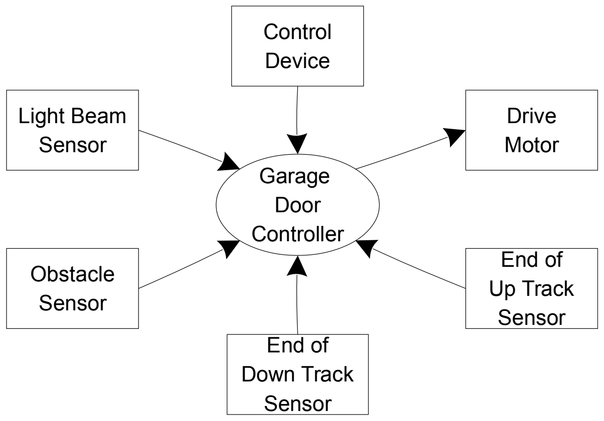

Figure 9: SysML Context Diagram of the Garage Door Controller

This example is deliberately small, yet it suffices to illustrate several of the previously described interactions. Here are a few examples:

- When the door is open, a signal from the wireless keypad triggers the drive motor.
- When the door is in motion, either opening or closing, a signal from the wireless keypad triggers the motor to stop. (Note: this portion of the problem could be interpreted as an ESML Pause.)
- When the door is closing, the Light Beam and the Obstacle Sensors are enabled.
- The fully closed sensor disables the Light Beam and the Obstacle Sensors.
- An input from either the Light Beam or the Obstacle Sensor triggers the drive motor to stop and then begin opening.

5.1 Statechart model of the Garage Door Controller

A full statechart model is given in Figure 10. The orthogonal regions describe local views of the full garage door, the motor, and both safety sensors. The input events and output actions are:

e1: wireless control signal	a1: start drive motor down
e2: light beam interruption sensed	a2: start drive motor up
e3: obstacle sensed	a3: stop drive motor
e4: end of down track reached	
e5: end of up track reached	

Some of the transitions are marked with output actions lettered a, b, c, d, and e. These will illustrate the broadcasting mechanism of statecharts. The sequence corresponds to the Statechart execution for the following scenario: pre-condition: door is up, Input event sequence: e1, e1, e1, e2, e5, and the post-condition: door is up. This corresponds to what an offline model-based tester would do to identify a test case for the scenario. Good practice dictates making an ”execution table,” as shown in Table 1.

Table 1: Execution Table for the Sample Scenario

Garage Door Statechart Execution Table						
	In States					
Step	Door	Motor	Light Beam Sensor	Obstacle Sensor	Input Event	Broadcast Output
0	s1	s9	s11	s15	e1	a
1	s5	s10	s12, s13	s16, s17	e1	b
2	s3	s9	s11	s15	e1	c
3	s5	s10	s12, s13	s16, s17	e2	d
4	s6	s8	s12, s14	s15	e5	e
5	s1	s9	s12, s14	s15		

To use a statechart for offline MBT, the tester must begin with a full statechart that shows only the input events that cause transitions in the orthogonal regions. Next, a scenario is postulated, and an exercise similar to the one above is followed to create an execution table. This is then the skeleton of a MBT test case. If the tester has access to a statechart engine, the process is greatly simplifiedthe tester simply selects a starting condition and defines a sequence of inputs. The engine produces an execution table similar to Table 1.

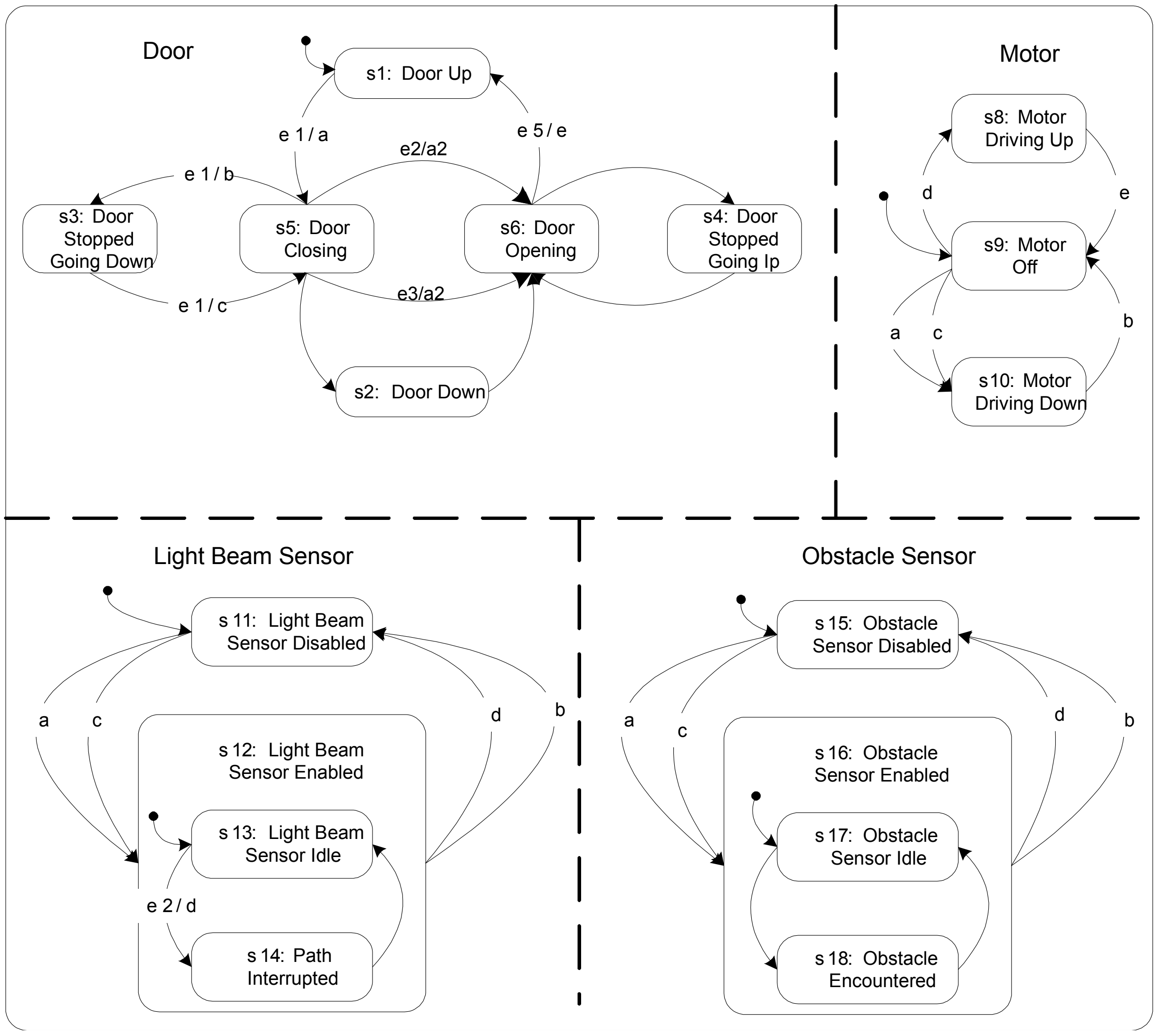
Door
s1: Door Up
e 1 / a
e 5 / e
e2/a2
e 1 / b
s3: Door Stopped Going Down
s5: Door Closing
s6: Door Opening
s4: Door Stopped Going Ip
e 1 / c
e3/a2
s2: Door Down
Motor
s8: Motor Driving Up
d
e
s9: Motor Off
a
c
b
s10: Motor Driving Down
Light Beam Sensor
s 11: Light Beam Sensor Disabled
a
c
d
b
s 12: Light Beam Sensor Enabled
s 13: Light Beam Sensor Idle
e 2 / d
s 14: Path Interrupted
Obstacle Sensor
s 15: Obstacle Sensor Disabled
a
c
d
b
s 16: Obstacle Sensor Enabled
s 17: Obstacle Sensor Idle
s 18: Obstacle Encountered

Figure 10: Full Garage Door Controller Statechart

5.2 Swim Lane EDPN models of the Garage Door Controller

Table 2 contains a legend for all the EDPN elements in Figures 11, 12, 13, 14, and 16. The basic operation of the Garage Door Controller is shown as an Event-Driven Petri Net in Figure 12 (the intermediate stopping and the safety devices are omitted). Places d1: Door Up and d4: Door Down are contexts for input event p1: wireless keypad signal. Since d1 and d4 are mutually exclusive, the context sensitivity is resolved. For now, assume the door is fully open (d1 is marked). When the transition t1 fires, output event p7: start drive motor down occurs, and the door is in the state d2: Door Closing. After some time interval (13 seconds in my garage) the door reaches the end of the down track, which is represented here as the input event p2. When transition t2 fires, the output p9: stop drive motor occurs, and the Garage Door Controller is in the fully closed state d4. If event p1 occurs again, transition t3 can fire, which causes output event p8 to occur and leaves the garage door in the state d5: Door Opening. Once the end of the up track is reached, input event p3 occurs, transition t4 fires, the motor is stopped (output event p9),

Table 2: Swim Lane EDPN elements for the Garage Door Controller

Input events	Output events (actions)	Data Places
p1: wireless keypad signal	p7: start drive motor down	d1: Door Up
p2: end of down track hit	p8: start drive motor up	d2: Door Closing
p3: end of up track hit	p9: stop drive motor	d3: Door Stopped going down
p4: brief motor pause		d4: Door Down
p5: light beam sensor		d5: Door Opening
p6: obstacle sensor		d6: Door Stopped going up

Table 3: Selected Paths in Figure 10

Path Description	Transition Sequence
1. Close an open garage door.	t1, t2
2. Open a closed garage door.	t3, t4
3. Open closed door and then close it.	t3, t4, t1, t2

and the door is back in the fully open state (d1). When described as an Event-Driven Petri Net (without swim lanes), we have the full picture, but we miss the interactions among devices. The interactions are apparent, and we could show the overall execution by a marking sequence.

System test cases can be derived directly from an Event-Driven Petri Net. A system test case corresponds to a sequence of transitions that fire. Since the EDPN in Figure 11 is 3-connected (a true path exists to and from every transition) there can be a countable infinite set of distinct paths. Table 3 lists three sample paths.

Deriving a full system test case is straightforward. The test case for Path 1 in Table 2 is:
Name: Close an open garage door.
Pre-conditions: Garage Door is open
Event Sequence

Input Events	Output Events
1. p1: wireless keypad signal	2. p7: start drive motor down
3. p2: end of down track hit	4. p9: stop drive motor

Post-conditions: Garage Door is closed

Figure 12 shows a detailed view of the interaction between the garage door closing and the enabling/disabling of the two safety features. This view is more appropriate for offline model-based testing, as it allows the tester to focus on specific interactions. In Figure 13, the interactions between the motor and the two safety devices are shown. This can be considered to be a continuation of the enabling and disabling of the safety features shown in Figure 13. If either safety device input occurs, the Trigger prompt immediately causes the motor to reverse and drive the garage door to the open position. Notice that Figures 13 and 14 could be composed into a larger, more expressive, Swim Lane EDPN. This becomes cumbersome, and even unwieldy quickly, as we see in Figure 14.

For comparison, look at Figure 14 – a fairly complete Swim Lane Event-Driven Petri Net showing the enabling of the safety features (light beam and obstacle sensor) and how an event from either device stops and then reverses the garage door motion. In Figure 14, assume an initial marking of place d1:

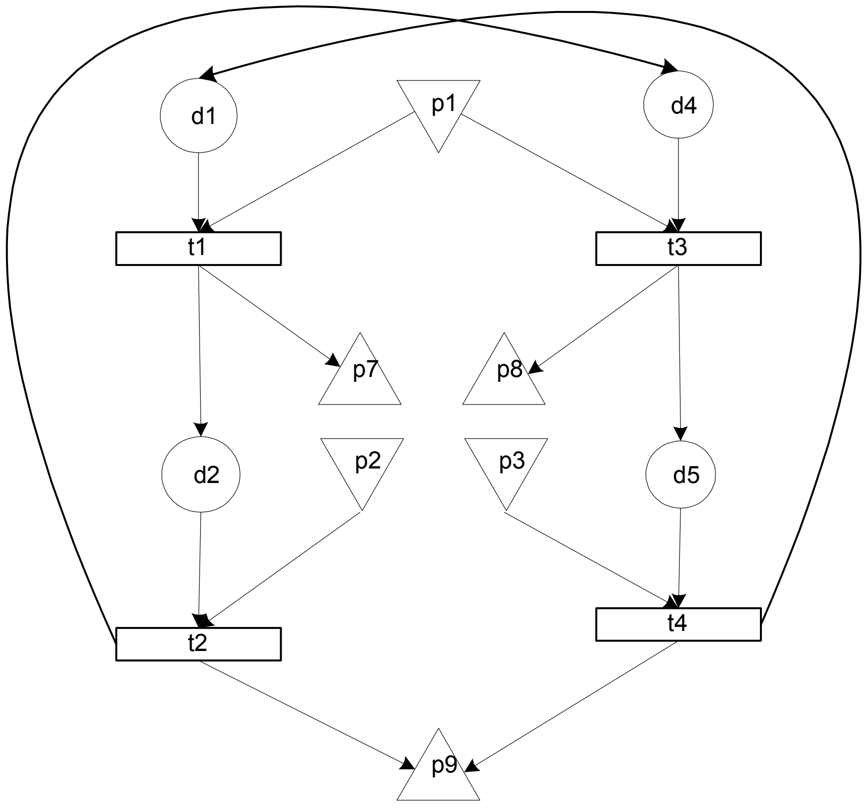

Figure 11: EDPN of the basic garage door operation

Door Up. If event p1: wireless keypad signal occurs, transition t1 fires with four results: the light beam sensor and the obstacle sensors are enabled, the drive motor is started in the down (closing) direction, and the Garage Door is in the d2: Door Closing state. If event p5: light beam sensor occurs, transition t3 fires, causing output event p9: stop drive motor, leaving the garage door in state d3: Door Stopped going down. (The scenario for the obstacle sensor is symmetric to that for the light beam sensor.) The next event is p4: brief motor pause, which allows transition t4 to fire because it has been triggered, and this causes output event p8: start drive motor up to occur, leaving the garage door in state d5: Door Opening. Event p3 occurs when the end of the up track is reached, which stops the drive motor (p9) leaving the garage door in state d1: Door Up.

6 Concluding Thoughts

Swim Lane Event Driven Petri Nets provide several advantages for model-based testing:

- they are a bottom-up approach
- they can be easily composed
- they permit focused description of interactions among constituents
- they can be used for automatic derivation of system test cases
- they support a useful hierarchy of system test coverage metrics.

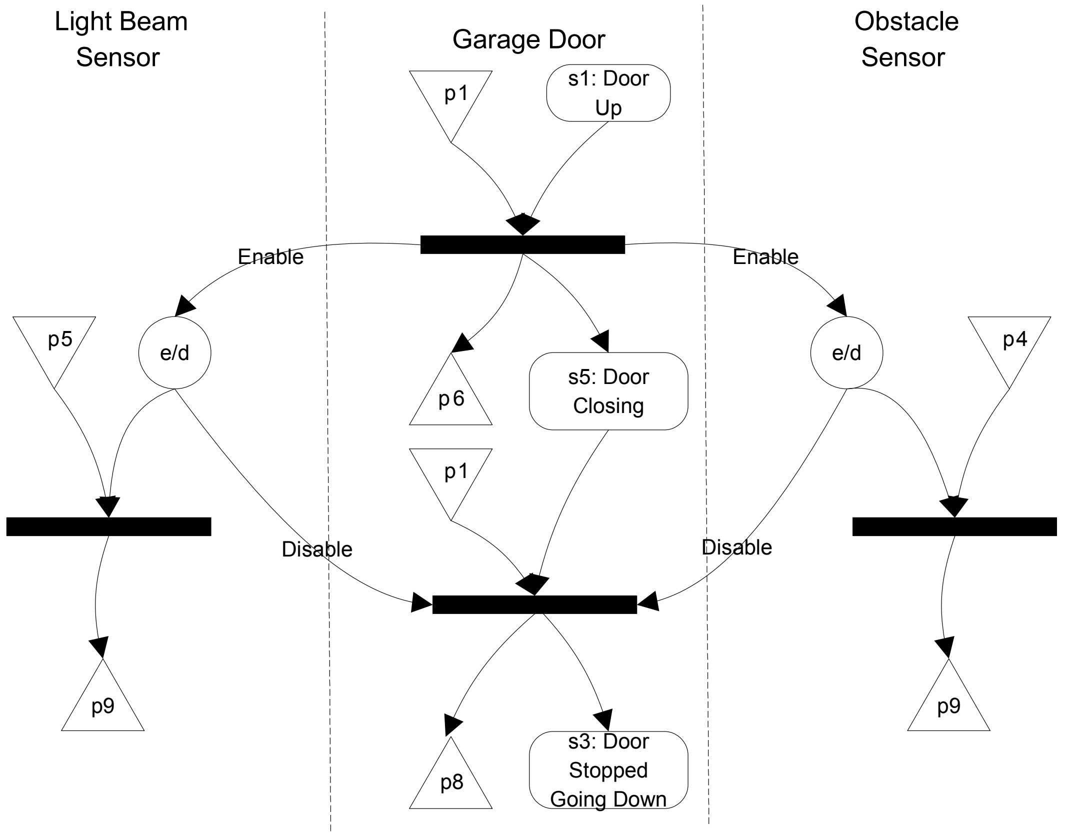

Figure 12: Swim Lane detailed view of safety feature enabling and disabling

The biggest limitation of Swim Lane Event Driven Petri Nets is that the drawings do not scale up well. They are vulnerable to a diagrammatic explosion similar to the ”finite state machine explosion.” All (graphical) Petri net models suffer from the problem of space-consuming diagrams. Figure 15 presents an elegant answer to this issue. Rather than compose Swim Lane EDPN diagrams, we can populate a database with the E/R description in Figure 15. With such a formulation, questions of connectivity are reduced to well-constructed database queries. One clear advantage of this is that now there is no practical limit to Swim Lane EDPN composition.

6.1 Graphical Composition of Event-Driven Petri Nets

Graphical composition of Event-Driven Petri Nets is straightforward, but it easily expands to spatial difficulty. Figure 16 shows two Event-Driven Petri Nets for closing and opening the garage door; their composition was shown earlier in Figure 11.

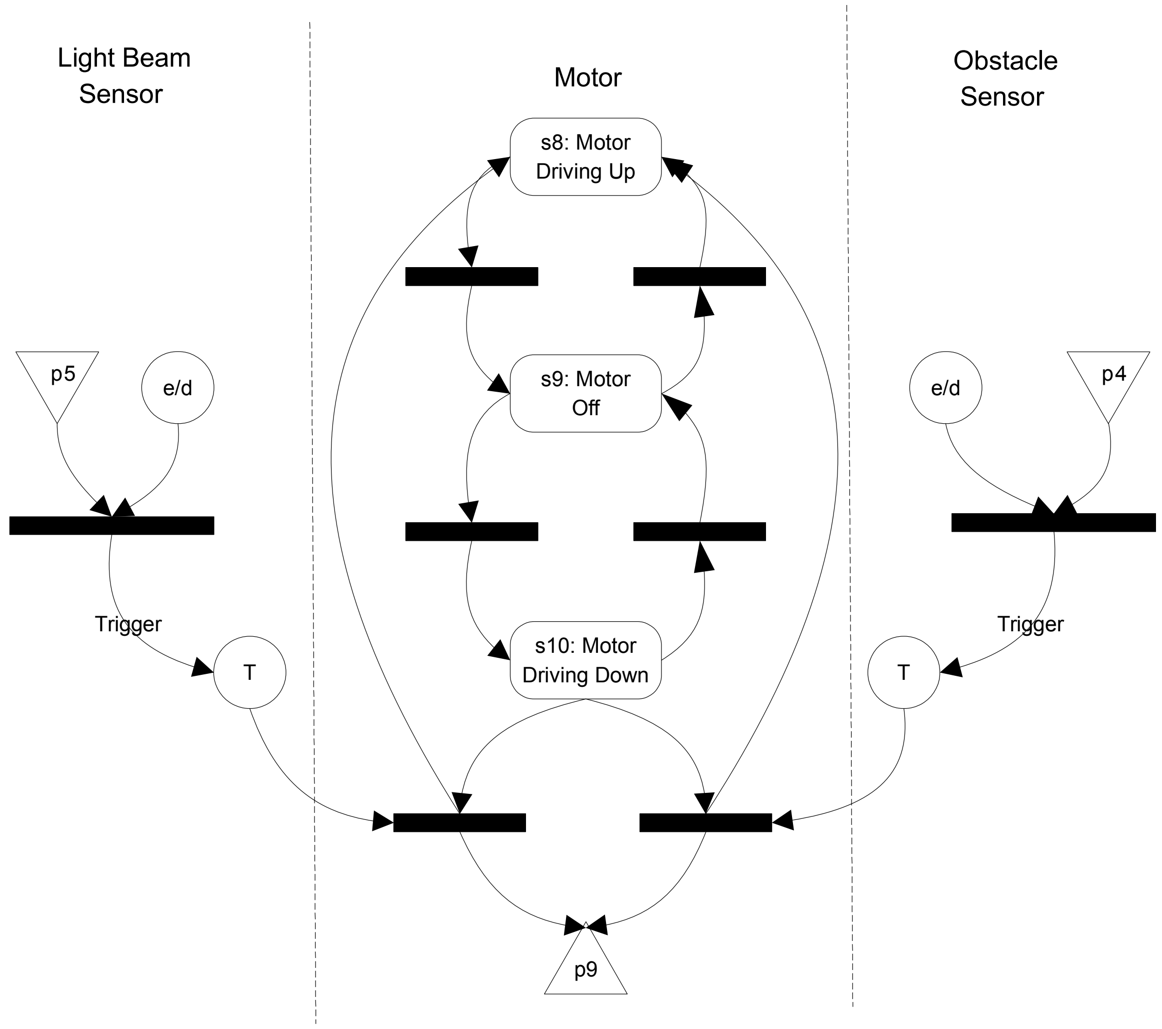

Figure 13: Swim Lane detailed view of safety feature

6.2 Composition of Event-Driven Petri Nets in the E/R Model Database

Database for Door Closing EDPN							
EventInput		EventOutput		DataInput		DataOutput	
Event	Transition	Event	Transition	Data	Transition	Data	Transition
p1	t1	P7	t1	d1	t1	d2	t1
p2	t2	P9	t2	d2	t2	D4	t2

Database for Door Opening EDPN							
EventInput		EventOutput		DataInput		DataOutput	
Event	Transition	Event	Transition	Data	Transition	Data	Transition
p1	t3	p7	t3	d3	t3	d4	t3
p3	t4	p8	t4	d4	t4	d1	t4

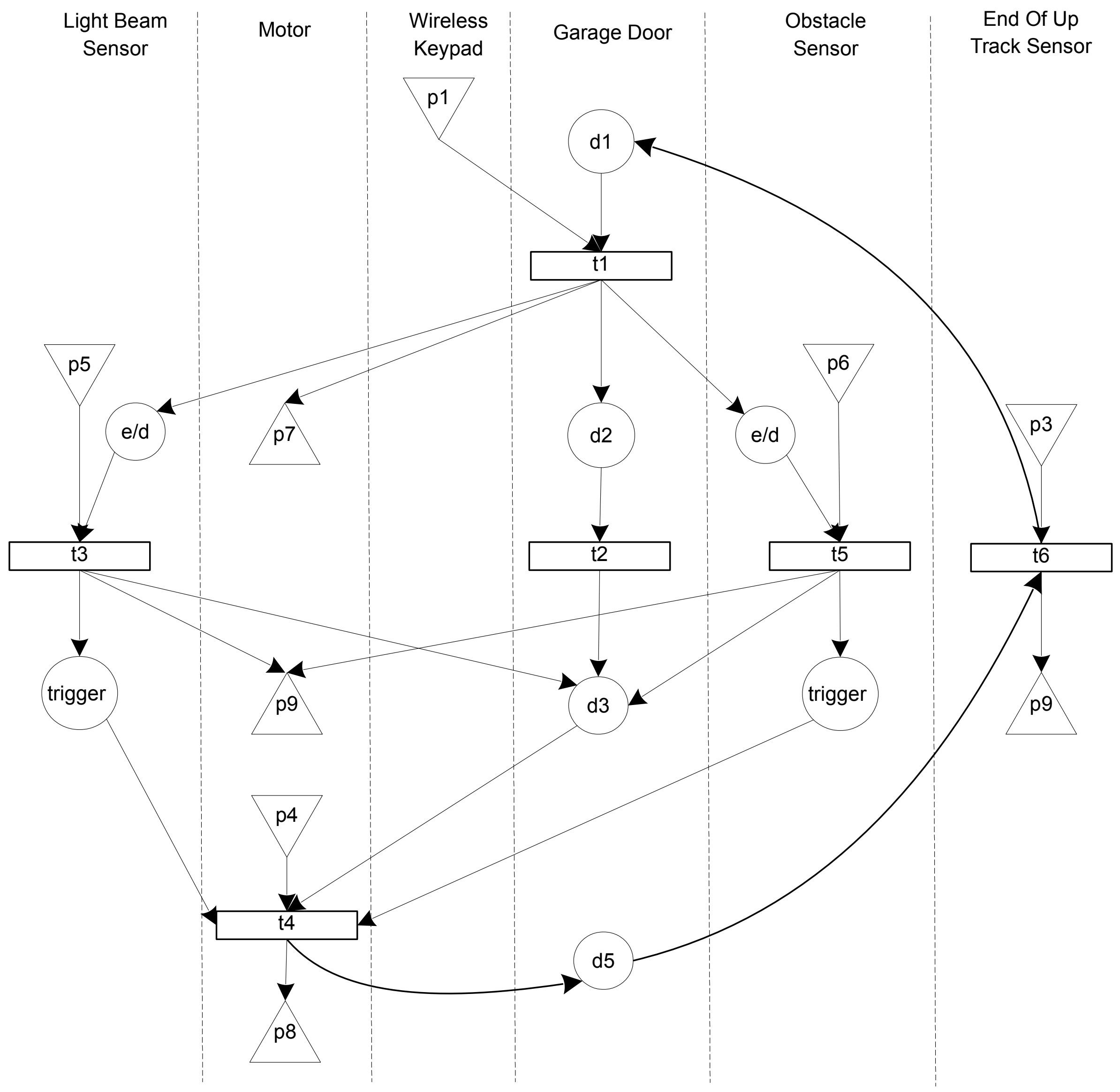

Figure 14: Swim Lanes for Light Beam and Obstacle Sensor Enabling

Database for Composition of Door Closing and Door Opening EDPNs							
EventInput		EventOutput		DataInput		DataOutput	
Event	Transition	Event	Transition	Data	Transition	Data	Transition
p1	t1	p6	t1	d1	t1	d2	t1
p1	t3	p7	t3	d2	t2	d3	t2
p2	t2	p8	t2	d3	t3	d4	t3
p3	t4	p8	t4	d4	t4	d1	t4

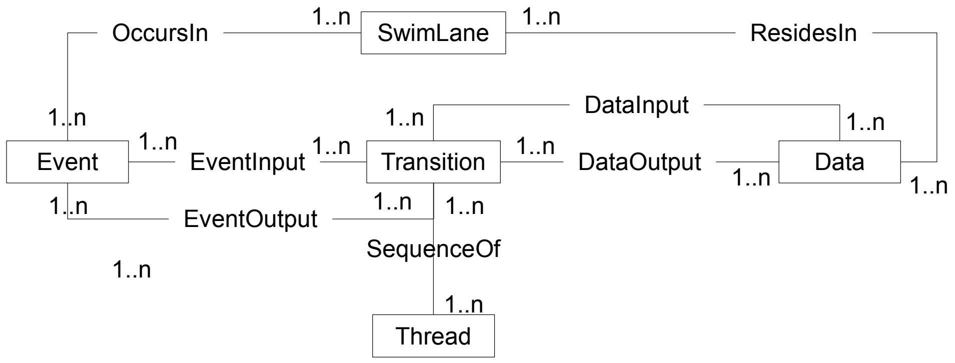

Figure 15: E/R Model of Swim Lane EDPN Database

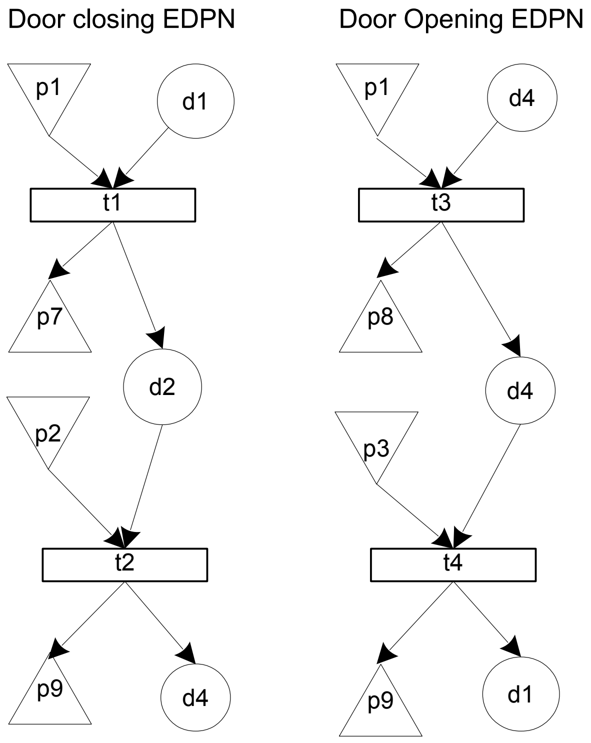

Figure 16: Individual Door Closing and Door Opening Event-Driven Petri Net

6.3 Further Guidelines

As a guideline, Swim Lane EDPNs are best used to focus on particular interactions, and then using the database approach to keep the overall model. While test cases can be derived by inspection of a graphical model, this is more difficult from the underlying database. Finally, Swim Lane EDPNs support the definition of the set of system level test coverage metrics given in Table 4. These metrics all refer to a set of test cases T derived from a corresponding Swim Lane EDPN description.

Table 4: Test Coverage Metrics

Test Cover	Description
Ct	every transition
Cp	every data place
Cie	every input event
Coe	every output event
Ccontext	Cie in every context

References

[1] W. Bruyn, R. Jensen, D. Keskar & P. Ward (1988): *An Extended Systems Modeling Language Based on the Data Flow Diagram. ACM SIGSOFT Software Engineering Notes* 13(1), pp. 58–67, doi:10.1145/43857.43866.

[2] B. DeVries (2013): *Mapping of UML Diagrams to Extended Petri Nets for Formal Verification*. Master's thesis, Grand Valley State University, Allendale, Michigan, April 2013. Available at `http://scholarworks.gvsu.edu/cistechlib/156`.

[3] D. Harel (1988): *On visual formalisms. Communications of the ACM* 31(5), pp. 514–530, doi:10.1145/42411.42414.

[4] D. Harel (1998): *Private conversation at OOPSLA Conference*, Vancouver, British Columbia. Canada.

[5] i-Logix (1990): *i-Logix technical presentation to Smiths Industries*. Grand Rapids, Michigan.

[6] i-Logix (1991): *i-Logix training cours*. Andover, Massachusetts.

[7] P. Jorgensen (2008): *Software testing - a craftsman's approach*, 4th edition. Taylor & Francis, ISBN 978-1-4665-6808-0, pp. I-XXIX, 1-464