

Time and Synchronization in Executable UML



Leon Starr

Software Model Engineer
Model Integration, LLC.
San Francisco, California, USA

www.modelint.com

www.linkedin.com/in/modelint

leon_starr@modelint.com

Abstract

This article illustrates the platform independent timing and synchronization rules that modelers use and architects implement to support Executable UML® applications. UML standardizes behavioral notations such as statecharts and sequence diagrams, but it does not define any synchronization rules. One of the most powerful features of Executable UML is that it does not constrain the implementation with unnecessary sequencing. In traditional programming languages, the stream of processing is sequential by default unless intentionally diverted through a variety of platform specific concurrency mechanisms. The opposite is true in Executable UML. Everything happens in a concurrent, platform independent timeframe unless there is explicit synchronization. This allows us to create application models that can be deployed onto arbitrarily distributed platforms. This capability is increasingly relevant as platform technology evolves from embedded to parallel to distributed to cloud and back, sometimes in the duration of a single project! So it is ever more critical that all the hard work you put into your analysis, models and designs not fall apart as the target platform inevitably shifts.

UML is a registered trademark of the Object Management Group

Note to hardcopy/PDF readers: You may want to access the convenient hyperlinks available in the always up-to-date online version found here along with my other writings: <http://knol.google.com/k/-/2hnjef6cmm97II7>

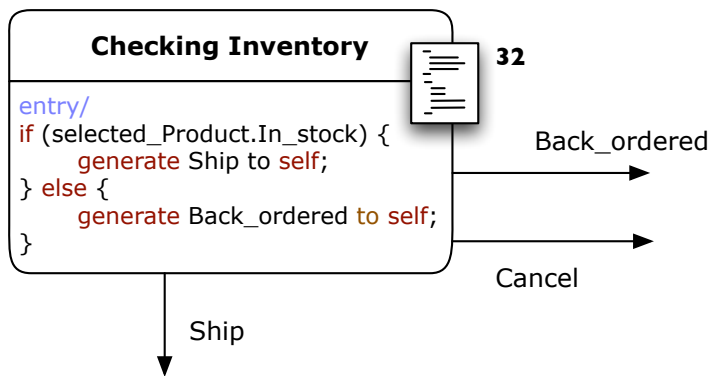
UML DOES NOT PRESCRIBE SYNCHRONIZATION RULES

With all of the dynamic diagram types in UML: sequence diagrams, collaboration diagrams, statecharts, activity diagrams and such, it may be surprising to learn that UML does not supply any timing and synchronization rules. This is simultaneously a feature and a curse. It is a feature in that you are free to define your own rules. It is a curse in that... you are free to define your own rules.

The notational limits of statecharts when it comes to definitive synchronization can be illustrated with a few simple examples.

Ambiguous example 1: Busy when event occurs

In this order processing application we have an object, **Order 32** sitting in the CHECKING INVENTORY state. It is executing the procedure inside the state when a Cancel event occurs.

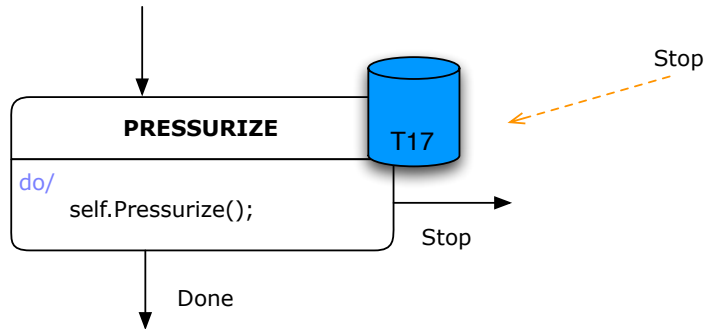


What does **Order 32** do when a Cancel event occurs during processing?

Is the processing interrupted and the Cancel transition taken? If so, what about any cleanup required? There is probably none in this particular case, but you can imagine what might be required in a more complex procedure. Or does the event get dropped since it arrives while the Order object is busy? Or could the event be saved until completion of the procedure? The point here is not that any particular rule is best, just that you need rules. UML can accommodate a range of possibilities, including none. Let's take a look at another example.

Ambiguous example 2: Interrupting a continuous action

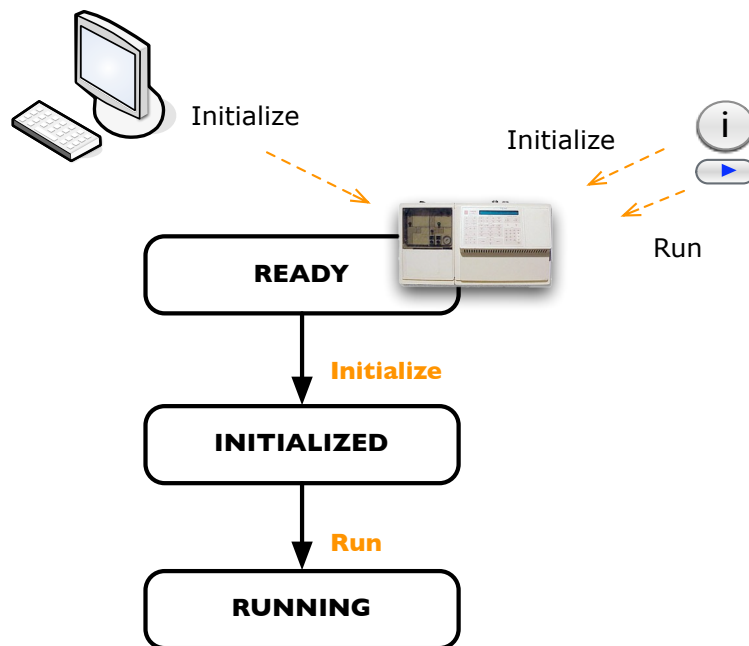
The mixing tank object **T17** is executing a **do/** action when the Reset event occurs. In this case, let's go ahead and assume that we want to interrupt the activity.



How, exactly, will this be handled? At what level of granularity is interruption possible? And, again, what about cleanup? The statechart offers notation, but no guidance as to how to process the interrupt. Okay, one last example...

Ambiguous example 3: Sequencing events

Here we have a chemical measurement instrument commanded from buttons on the front panel of the device and/or via a remote workstation connected on a network. A human operator first presses the Initialize and then the Run buttons triggering the corresponding events. Simultaneously, a command that triggers another Initialize event arrives on the network. (Perhaps the operator and remote user are unaware of the potential conflict). What happens?



Are the two events commanded from the button panel processed in the intended order? Could the network Initialize event happen in between, and if so, will it be ignored or processed?

At the end of this article we will revisit these three examples and see how Executable UML resolves the ambiguities.

RESOLVING SYNCHRONIZATION AMBIGUITY

In these three examples we've seen that UML notation alone does not resolve real-time ambiguity in statecharts. Given that we are modeling a real-time system, what are the options for proceeding?

Option 1 - Informal specification

Maybe you are building high level models that aren't intended to solve any synchronization problems. You just want a very informal specification of behavior and plan to leave all the hard work to the developers. Leave it to the coders and fix things in beta. Yee-ha!

Option 2 - Mix code into the specification

We can elaborate the weak statecharts by writing the procedure in each state using a traditional programming language like C/C++ or Java. The code can invoke concurrent operating system facilities such as threads, semaphores, critical section marks and so forth. This may work, but it doesn't guarantee consistent rules across all statecharts (or even within a single statechart), so the models may be quite complex in the end. Let's assume though, that considerable thought goes into the coding and a relatively simple and global scheme is devised to keep the complexity from getting out of hand. You will then end up with largely platform specific models. This compromises the quality of the analysis (mixed in implementation complexity), precludes early simulation and testing (you've got to write all the code first) and limits the models to the target platform (redesign and retest required to move to another platform).

So the procedure coding approach may address the problem of implementing synchronization, but you have to tread through a lot of implementation muck to get there. (And you have to keep treading each time you build a statechart!) The difficulties of debugging systems that use concurrency mechanisms are well documented out on the internet.

Option 3 - Model synchronization requirements with UML

Here, the idea is to apply a simple set of rules consistently across all of the statecharts. Both the analyst and the designer benefit from this scheme. The analyst need not get bogged down in complex synchronization mechanics and can build models that clearly [expose](#)¹ application requirements. The designer can more easily implement a lean, efficient state machine engine based on a uniform rule kernel. The designer need not pick through and accommodate varied and specialized control mechanisms scattered and buried inside the actions of each statechart.

The synchronization rules may be simple, but they must simultaneously offer the ability to capture complex application requirements and be implementable on a wide variety of platform technologies.

Consider some of the advantages over options 1 and 2 if this can be done.

You will be able to build unambiguous, testable models of complex application behavior. These models will be platform independent (PIMs).² In other words, the implementation specific concurrency and synchronization mechanisms will be factored out of the application models. What will remain are the application specific synchronization requirements essential to all implementations. Well defined execution rules make it possible to run the models on a virtual machine for testing purposes prior to implementa-

¹ How to Build Articulate UML Class Models [1]

² Model Driven Architecture (MDA) terminology [2]

tion. Rough metrics concerning peak and average signal/data exchange, relative action execution and dwell time and various other latencies can be gathered. Finally, a single executable [domain](#)³ can be implemented on a variety of concurrent platforms without necessitating changes to the models themselves⁴. This means that once the application synchronization is tested and validated it need not be re-worked and retested just because a platform with varying distribution, timing and synchronization mechanisms is targeted.

These advantages should be compelling for anyone building complex real-time systems that must accommodate evolving platform technology. (Especially if task or processor distribution is in flux)

In this article we will explore option 3 as plenty has already been written about options 1 and 2!

But first, a lot of assumptions have been piling up, not least of which is that it is even possible for synchronization to be platform independent. Let's see.

AN EXAMPLE OF PLATFORM INDEPENDENT TIMING AND SYNCHRONIZATION

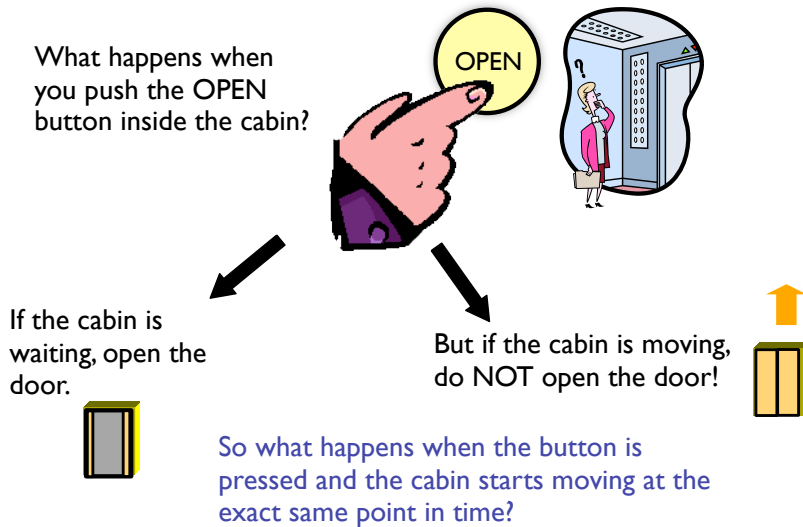
Some developers believe that timing and synchronization can be sorted out only with the platform and target programming language taken fully taken into consideration. Without a detailed understanding of how threads, tasks, semaphores and other synchronization facilities, including the hardware, work, how can you definitively express and test any synchronization scheme? This is the thinking pushing most developers toward option 2. But maybe it doesn't have to be that way.

Consider a synchronization problem in an elevator⁵ control system. A passenger standing inside the cabin is going to push the **OPEN** button. If the cabin is at rest, the doors should open. If the doors are already open, nothing happens. If the cabin is moving and the doors are closed, again, nothing should happen. Now what happens if the open button is pressed at the exact same instant that the cabin wants to move?

³ A domain is a type of package that contains a complete set of executable content, modeled or coded. There are some other Executable UML specific definitions of a domain described in Mellor-Balcer[3].

⁴ The accompanying marking model (MDA terminology [2]), however, will likely change. It marks various model features to be handled specially by the model compiler. Each MVM will make various marking features available appropriate to the class of targeted platform. For example, certain classes might be marked for flash memory implementation.

⁵ That, of course, would be a "lift" control system for my British friends.



Is it necessary to consult the pthreads reference book before you can answer? Maybe dust off your favorite real-time OS manual just to be sure? Of course not! You already know what is supposed to happen. The cabin can never be moving while the doors are open. But how do we express the resolution of simultaneous events? Informal text can't solve the problem. An implementation certainly can.

Option 3 says that we need to find a middle ground where we can unambiguously model the synchronization requirement in a single Executable UML model. This model can then be transformed into many different code sets, each implementing the exact same rules but using the unique features of a particular platform. How do we do this?

Defining and executing platform independent synchronization rules

First we define a set of rules for interpreting statechart behavior. Actually, we don't have to define them since they are conveniently defined in Executable UML. (Though there is nothing to stop⁶ you from defining your own!) For example, one Executable UML rule says that "a procedure in a state always completes before the object processes any new events". We'll get to the rest of these rules soon.

A modeler / analyst will then model one or more application domains using class, state and action models that conform to the Executable UML rules.

During development, the statecharts may be populated with instance data and executed either by hand (coins on sheets of paper) or, ideally, using a model interpreter that supports the Executable UML rules. These do exist, or you can build your own⁷ if you are looking for something to do.

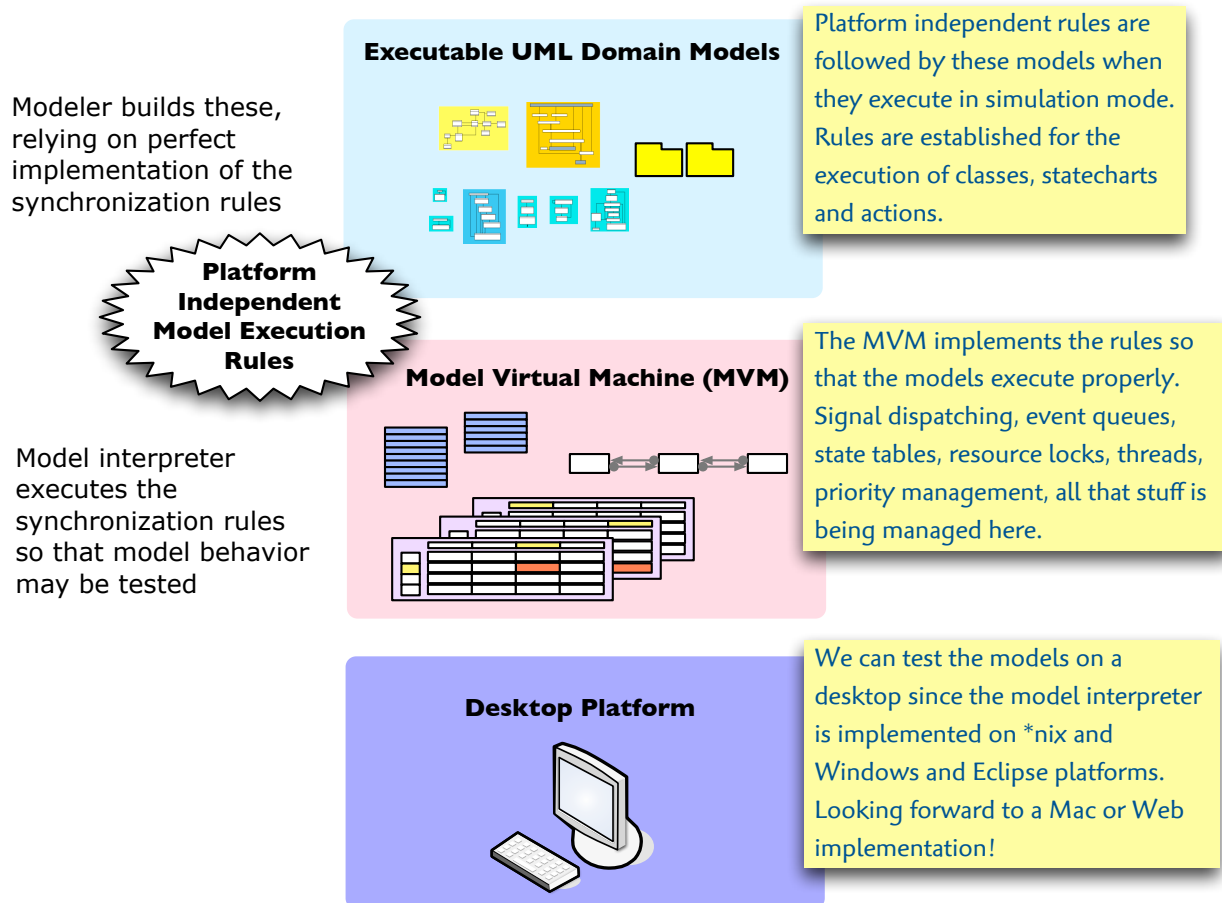
The model interpreter or model virtual machine (MVM) must be constructed so that the rules observed by the modeler are correctly implemented.

⁶ Well, ... time and money, perhaps. More about this at the [end of the article](#).

⁷ Send me an [e-mail](#) when you get it working!

Here is an environment that executes UML models on a desktop:

Running the Models in Simulation Mode



The MVM will allow us to put all of our objects into an initial state, trigger an initial event and then let the models execute actions and move state by state. Ideally, breakpoints can be set on states, events and individual actions within procedures. (It's actually more than an ideal. These things do exist out in the wild). There is, of course, more to the MVM than just executing state synchronization rules. Some UML actions access attributes and navigate class model relationships. So, in addition to statechart rules, executable data and computation semantics must also be supported by the MVM.

SIMULATION IS FINE, BUT WHAT ABOUT IMPLEMENTATION ON THE TARGET PLATFORM?

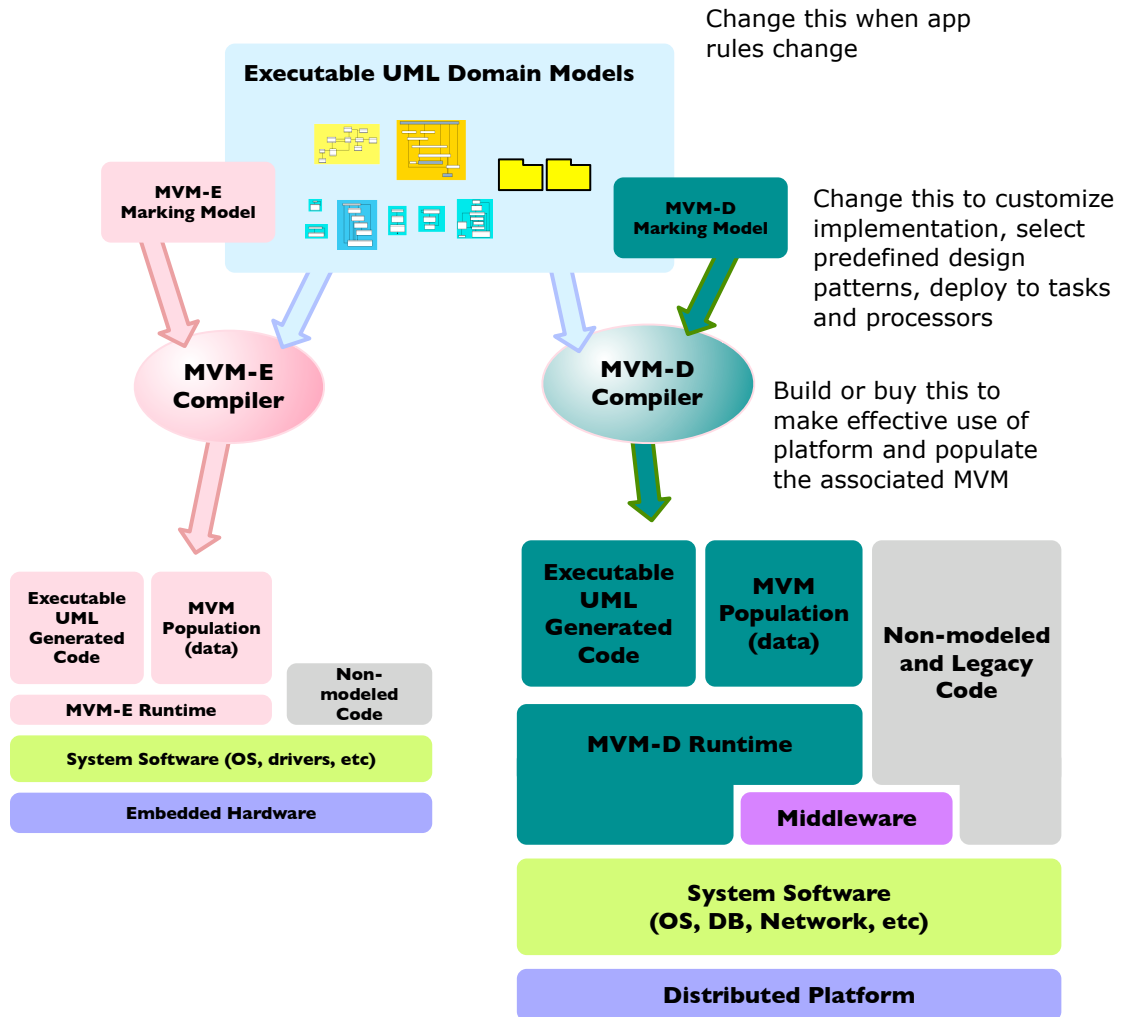
It's nice to run your models and test them against platform independent synchronization rules. But the desktop implementation of the model execution rules will be inappropriate for a highly embedded target or a highly distributed target. In fact, the desktop model interpreter is just one of many possible platform targets.

Let's say that we are compiling to a highly embedded platform. In that case you are probably implementing in C instead of Java. Yes, I know there's an embedded Java runtime environment, but I'm talking highly embedded, 50 - 256k, for example. There are many places where a Java virtual machine won't fit. But

you can find room for the model execution rules, though, in extreme environments, you will limit the types of applications that can be supported.

So the implementation solution is not to require that each platform accommodate a full scale Executable UML virtual machine. Instead, application realities such as the quantities of persistent and non-persistent objects, quantity of read only data and lots of other factors can be taken into account to create an Executable UML run time environment suited to a class of platform.

Model Compilation on a Target Platform



GUIDING PRINCIPLES OF THE TIME AND SYNCHRONIZATION RULES

Having surveyed the benefits desired and the technical concerns that must be addressed, we can list the guiding principles of the Executable UML time and synchronization rules:

- 1) Simplicity
- 2) Minimize platform assumptions
- 3) Leverage existing theory

SIMPLICITY

Define as few rules as necessary. There's enough complexity in the application without folding in a set of synchronization mechanics riddled with exceptions, redundancies, event juggling and priority schemes. As stated earlier, this benefits both the analyst and the designer. Subtle bugs often result because faults in the application logic are clouded by complex language mechanics. Executable UML models are quite easy to review compared to a lot of the UML gunk out there.

MINIMIZE PLATFORM ASSUMPTIONS

If the synchronization, data access and other execution rules are truly platform independent, the application models will be deployable on a variety of platforms. Since the platform is not known ahead of time, it is best to assume the worst. A rule assuming a global clock is available, for example, would limit deployment to single CPU targets. An assumption that a group of classes in a subsystem should run in the same thread will have limited portability. So we would prefer to go the other way and make the most minimal assumptions about synchronization possible knowing that the models, once tested, will work the same way on any arbitrarily distributed platform.

LEVERAGE EXISTING THEORY

Executable UML builds on existing theory. The data semantics, for example, are built on relational theory. The synchronization rules borrow from distributed networking theory⁸.

Now, on to the rules.

⁸ [Lampert](#) [4]

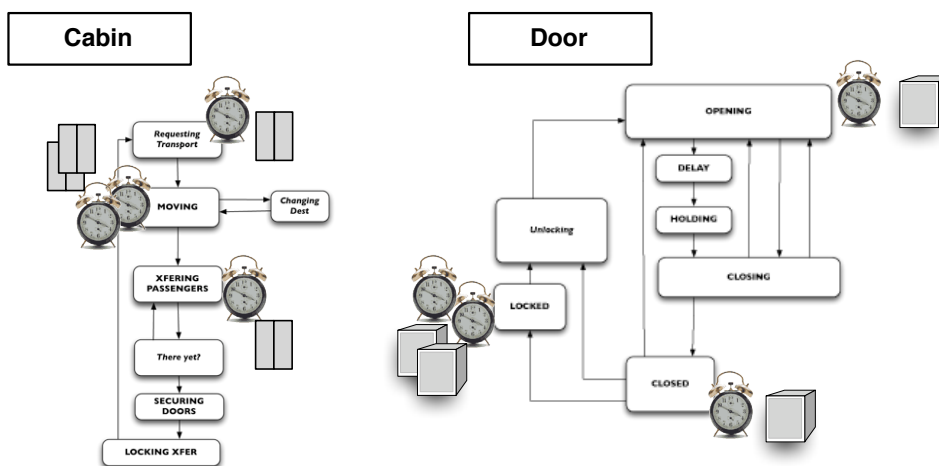
The Rules

THERE IS NO GLOBAL CLOCK



The concept of a global clock, where all model elements are running in the same absolute timeframe is difficult and often impossible to implement on distributed or multi-processor platforms. It is a nice convenience if you are targeting a single CPU, but we can't count on that. In fact, the actual implementation of the object state machines could be looping in a single thread within a single task in the degenerate case of distribution. At the other extreme, each object could be running in its own processor. By assuming the worst case of distribution, we know that we can target all cases.

So there is no global clock. We assume that time is local to each object. In this illustration we depict each object running in its own local time.

There is no global clock



At any point in time, each object:

- is in one state
-  has its own local time
-  is either executing a procedure or waiting for an event

Each class may be populated with some number of objects. Here we are concerned only with those classes with statecharts. I like to picture the statechart of a class like a game board traversed by each object of the class. At any frozen point in time, each object is in exactly one state of its class's statechart.

So all objects belonging to the same class are driven by a single statechart. Across multiple classes, we have all of the objects, each in a current state either executing the entry procedure within that state or just waiting around for an event to occur.

Obviously there are a few implications to this way of thinking that must be taken into account by the remaining rules.

THE DURATION OF A PROCEDURE IS UNKNOWN

Each state contains a single entry procedure that we will refer to as the state's "procedure" from here on as done in Mellor-Balcer[3]. Within this block there may be multiple actions. It is also possible for the entry procedure to be empty or to just contain a comment.

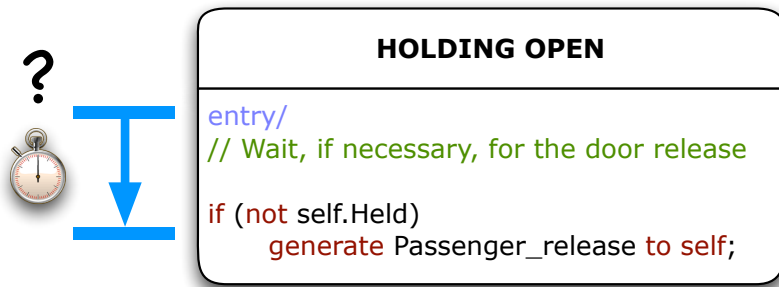
While standard UML notation provides for entry, exit and do actions, Executable UML uses only entry actions. Why are the other two omitted? Simplicity. It turns out that you can do it all with entry actions. You can also do it all with exit actions, for that matter. In the development of the Executable UML profile, much attention has been focused on having one way to solve a problem instead of fifteen. That way a given set of requirements should reduce down to a common model rather than multiple alternatives with no objective criteria for distinguishing one from the next.

It is a lot easier to read through a collection of statecharts all using entry actions than it is to parse through a different scenario for every state.

You may also note that procedures are associated with states ([Moore](#) style) as opposed to transitions ([Mealy](#) style). The debate as to which style is better is akin to that for curly braces in C code and just as tedious. There is no significant complexity difference between styles. The Moore style used in Executable UML does lead to a simplified state table (the thing that actually gets compiled) and makes it easy to specify the re-execution of an action (go back to the same state).

When an object enters a state, it begins executing the entire procedure immediately. At this point in time we will call it a busy object. It takes an unknown period of time to complete the procedure. That is because the speed of execution is platform dependent. Upon completion, the object waits for an event to occur. Objects never automatically advance to another state. At this point in time we have a waiting object.

Platform dependent duration

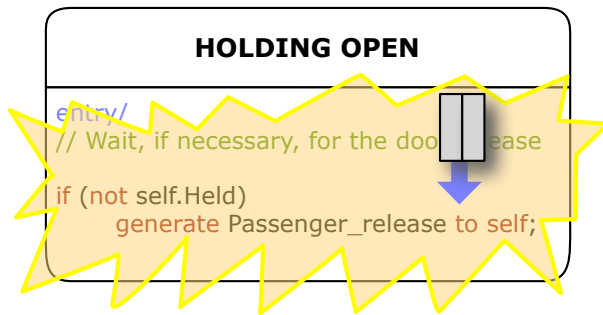


In a platform independent world, the duration of procedure execution is finite, but unknown.

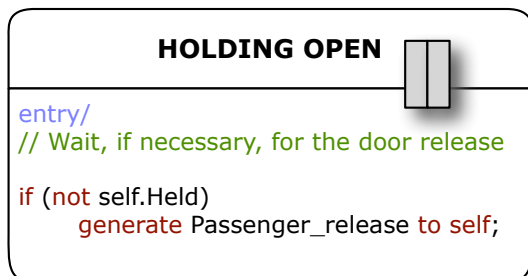
EVENTS ARE HELD FOR BUSY OBJECTS

Waiting objects are idle until one or more events occur. If one of these events corresponds to a transition exiting the current state, the waiting object will follow the transition and get busy again.

Busy objects do not see any events. When events targeted at a busy object occur, they are held. At least one of these held events will become visible when the busy object has completed its procedure and becomes a waiting object.

Busy Object

If events occur now, they will not be seen by the busy object. But they will be held by the MVM for later processing.

Waiting Object

After the action is executed, the object can see pending or newly occurring events.

The object waits in this state until an event occurs that triggers a transition.

Implications for the modeler

Since you cannot interrupt a procedure with an event, a state effectively defines the granularity of interruption within a modeled domain. So if you have an activity you might like to interrupt, you should break it up into a sufficient number of states. Conversely, a collection of actions that won't be interrupted can be grouped into the same state. (You can put the actions into an object method and just invoke it from one or more states to keep from drawing giant state rectangles!).

The modeler can't just assume that a state will be busy for a short or long time and just hope it completes before some other parallel activity. "This action should be really fast, so let's assume that it finishes before this other thing..." This lazy approach will not be rewarded kindly in a platform variant world! If a procedure must finish before or after some other activity, it needs to be explicitly coordinated with interlocking states or some other definitive mechanism.

If the modeler is not careful, a set of models might work fine during simulation and then break on a target implementation running at a different speed. Even worse, a system may work fine both in simulation and on the initial target platform. Then, some day, the platform changes either with new hardware or improved efficiency and *then* the models break! Platform independent testing puts the burden of careful synchronization and clear understanding of the synchronization rules on the modeler. That said, it is much easier to solve and test these problems in Executable UML than it is at the implementation level. And models that pass these tests should prove more robust and resilient than the brittle alternative frozen around a single platform.

Implementation notes

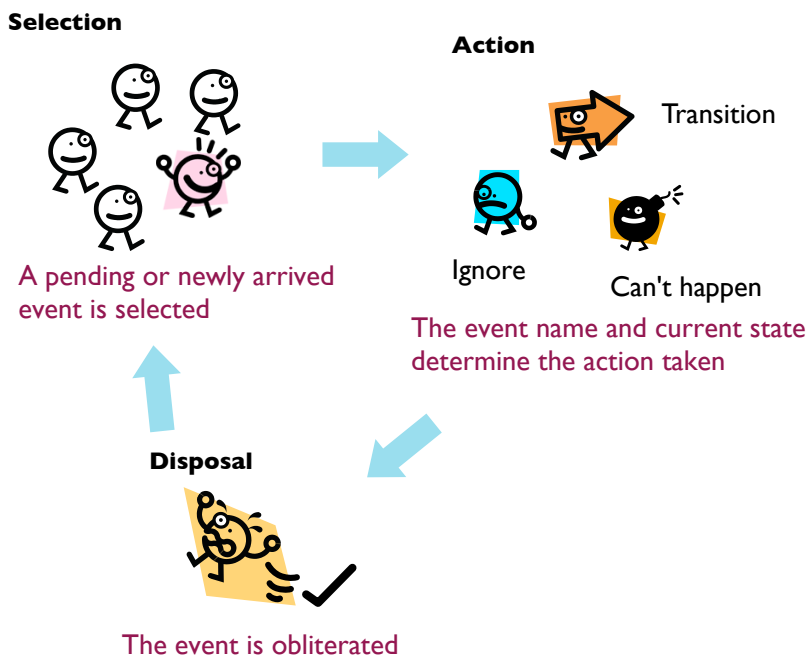
While a procedure cannot be interrupted by an event at the model level, the MVM may suspend incomplete procedures to process system interrupts and then resume again. This is analogous to your desktop operating system swapping out your spreadsheet application momentarily and then swapping it back without you realizing it.

ALL EVENTS ARE PROCESSED AND DISCARDED

If an object is waiting, events are processed as they occur. If one or more events are collected while an object is busy, they are processed one at a time once the object finishes executing its current procedure. In either case, each event is subject to the same event processing cycle.

The cycle consists of three phases: selection, action and disposal. For a waiting object, the event processing cycle is triggered when the next event occurs. This cycle is applied once for each event until a state transition occurs or there are no more pending events.

The event processing cycle repeats until a state transition occurs or there are no more pending events.



Selection

If there are pending events, the MVM (not the object) selects and presents one to the waiting object. If not, the MVM will present the next incoming event.

You may be surprised to learn that pending events are not necessarily queued. They can be, but it doesn't make any difference as far as the models are concerned. When event ordering is critical, it must be modeled explicitly without hoping that the MVM picks the right event. So the MVM is free to use whatever algorithm it likes to choose among pending or simultaneously occurring events. In practice,

pending events are typically queued. This is a nice split between the MVM's and the modeler's responsibilities. The modeler shouldn't be worrying about how the MVM accomplishes its job as long as the synchronization rules work. Event prioritization, event ordering and granularity of interruption can always be managed with a properly synchronized and domain partitioned set of models without relying upon any special MVM magic⁹.

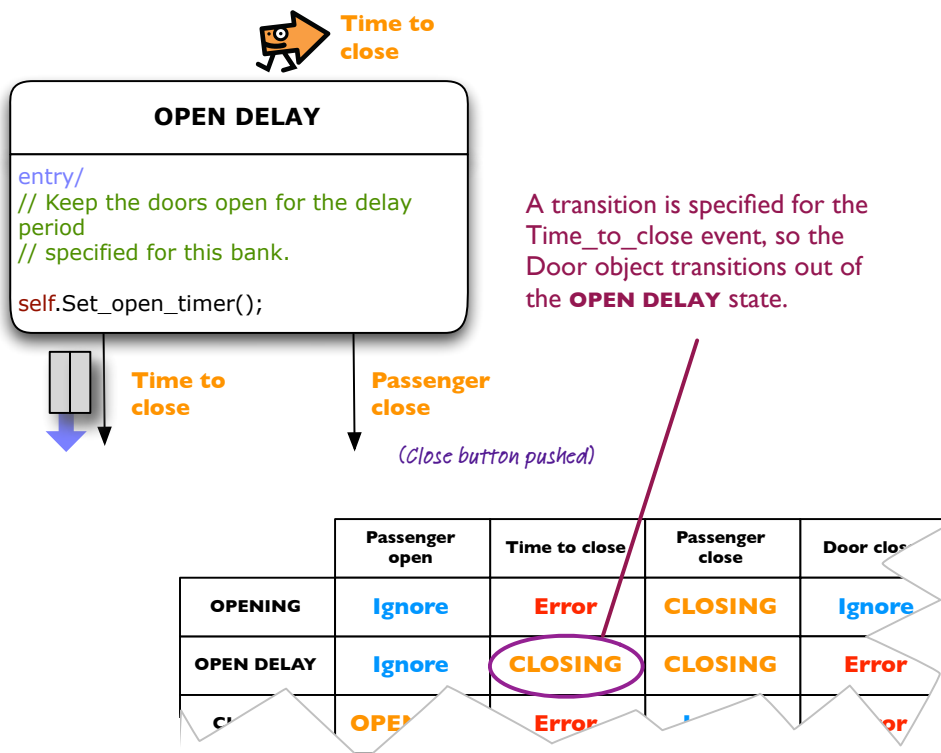
Action

The processing of an event in a given state is defined by the modeler as exactly one of the following choices: **transition**, **ignore** or **error**¹⁰.

Transition

If a transition has been specified for the event, then the object will follow that transition to the destination state. Any parameter values supplied with the event will be made available to the procedure in the destination state.

Event triggers a transition



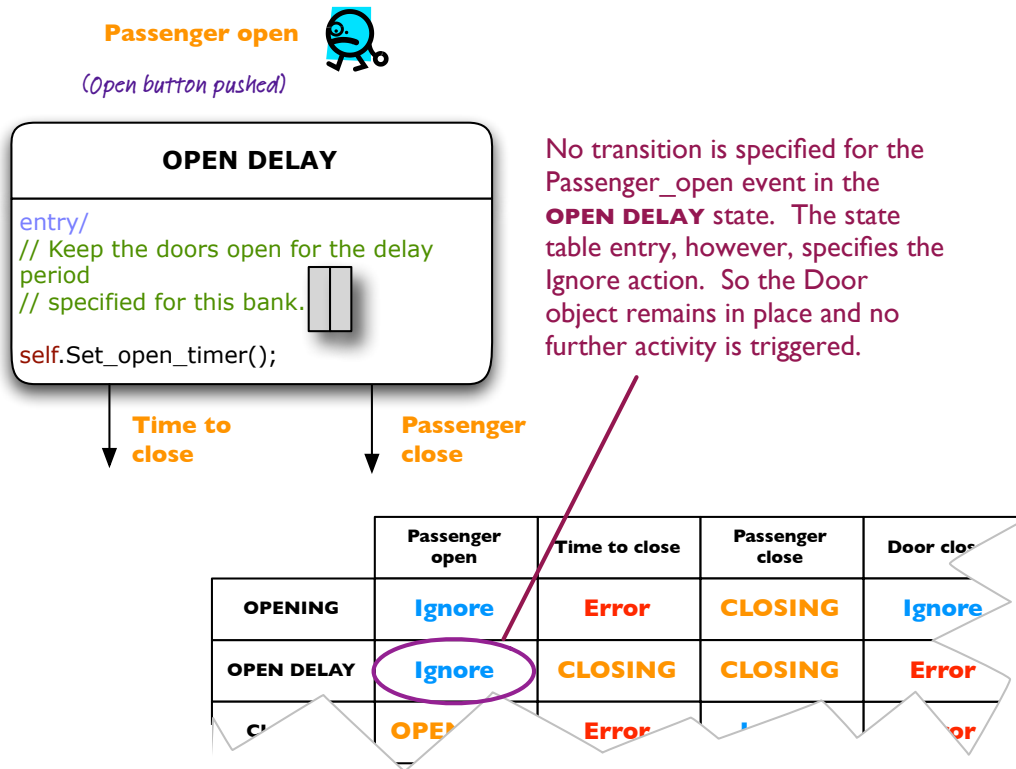
⁹ Novice modelers often demand special features from the MVM that would unnecessarily complicate the synchronization rules instead of just thinking the problem through and building the correct models. Event prioritization a commonly requested superfluous feature.

¹⁰ "Error" is also called "can't happen" or CH in the Executable UML community.

Ignore

Otherwise, if **ignore** has been defined in the state table (not visible on the statechart), no action will be taken. The object remains in the current state. Imagine the response to someone pressing a button multiple times to call the elevator cabin to a floor. After the initial call is registered, but before the cabin arrives, it is safe to drop any subsequent presses.

Event ignored

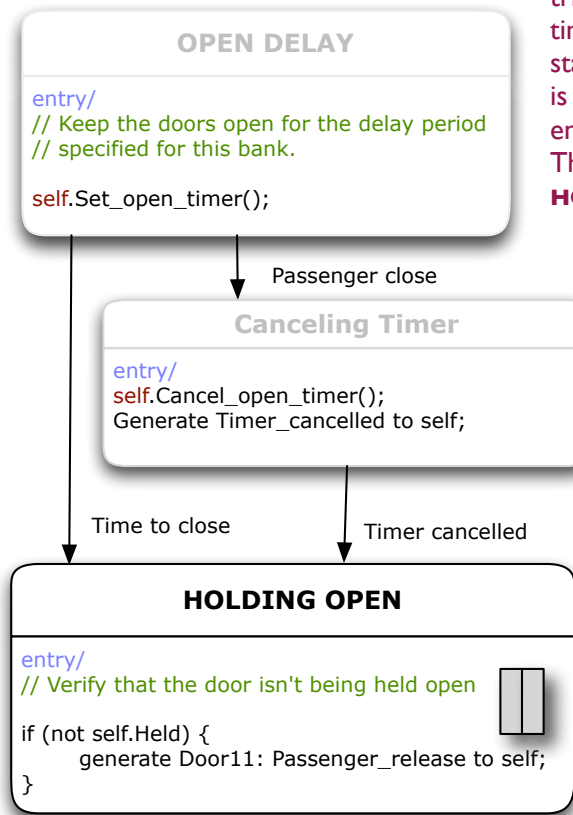


Error

Finally, if **error** has been defined in the state table (again, not visible on the statechart), an error response is triggered in the MVM. This is similar to an exception in program code that bubbles up to the interpreter or run-time environment without being caught. A standard response to **error** will be designed into the MVM appropriate to the platform¹¹.

I should probably point out that the state table is not part of UML. It is, however, necessary to fully define responses for each event-state combination. In practice, the process of carefully filling in each cell of the table will reveal several holes in an apparently complete statechart. Statecharts are great for identifying and understanding lifecycle behavior patterns. State tables are critical for ensuring completeness and directing attention to logical blind spots.

¹¹ The exception response of a Mars rover, an embedded cardiac pacemaker and an inventory control system will differ widely.

Event can't happen!

The `Time_to_close` event is triggered by the expiration of a timer set in the **OPEN DELAY** state. The timer either expires or is cancelled before an object may enter the **HOLDING OPEN** state. Therefore, it cannot happen in the **HOLDING OPEN** state.

The modeler has marked the state table entry for the `Time_to_close @ HOLDING OPEN` as **error**.

If the `Time_to_close` event does, in fact, happen in the **HOLDING OPEN** state, an exception is triggered in the MVM.



Time to close

Either the model must be fixed (perhaps the `self.Cancel_open_timer()` operation didn't correctly kill the timer. Or the MVM has issues! (Usually the MVM is correct and the modeler must figure out what he or she did wrong to anger the gods).

Disposal

The event is discarded. Once processed, an event is gone forever as far as the MVM is concerned. The modeler is, however, free to register the occurrence of an event or just the parameter data delivered, for later reference, by entering or updating data in the class model. Consider, for example, the modeled event "Seismic Activity (Magnitude:2.7, Epicenter:San Francisco, Time:09:24)" which could trigger a state procedure that creates a new object in the Earthquake class and sets the corresponding attribute values. The general rule is that if you want a fact to persist, you need to put the appropriate data into the class model. Taking advantage of this fact can greatly simplify statecharts that attempt to field the same event in every single state for fear of missing it.

When a signal is delivered to an object busily executing a procedure, the MVM creates a corresponding event and holds it. If more events occur, they are also held. Eventually, the procedure completes and the

MVM presents one of the held events to the object. The object will process this event as transition-ignore-error. In all three cases, the presented event will be destroyed in the process.

Signals and Events

When a signal is delivered to an object, it triggers an event of the same name. In proper UML speak, a signal is sent and an event occurs. Since the only events that occur in Executable UML are triggered by signals, we often use the terms signal and event interchangeably.

If the event is ignored, it evaporates away and the MVM presents another held event, if there are any left. If the event triggers a transition, again the event disappears with any supplied parameter values passed to the procedure of the subsequent state. If more than one event was pending, it will continue to be held until completion of the next procedure. Does this mean that held events may be processed several states downstream? Yes! Events persist until they are processed, at which point they are destroyed. (This suggests yet another technique to avoid having to field the same event in every state of a statechart). Both techniques will be employed in the upcoming door statechart example, so keep scrolling!

Events are not necessarily prioritized or queued by the MVM. In other words, the modeler should not count on any such thing. Events happen. If you need events to happen in a particular order, or if certain events must supersede or preclude lesser events, you need to manage this in the synchronization of your states. Remember that events are held, not necessarily queued.

The MVM sees two kinds of events. Self directed and non-self directed. When an object sends a signal to itself, usually as part of an if-then action or just to trigger an advance at the end of a procedure, it results in the occurrence of a self-directed event. Collaboration among other objects of the same or different class happens when a signal is addressed to any other object.

Self directed events are always presented to an object before any non-self directed events are seen.

For the moment, let's consider only non-self directed events.

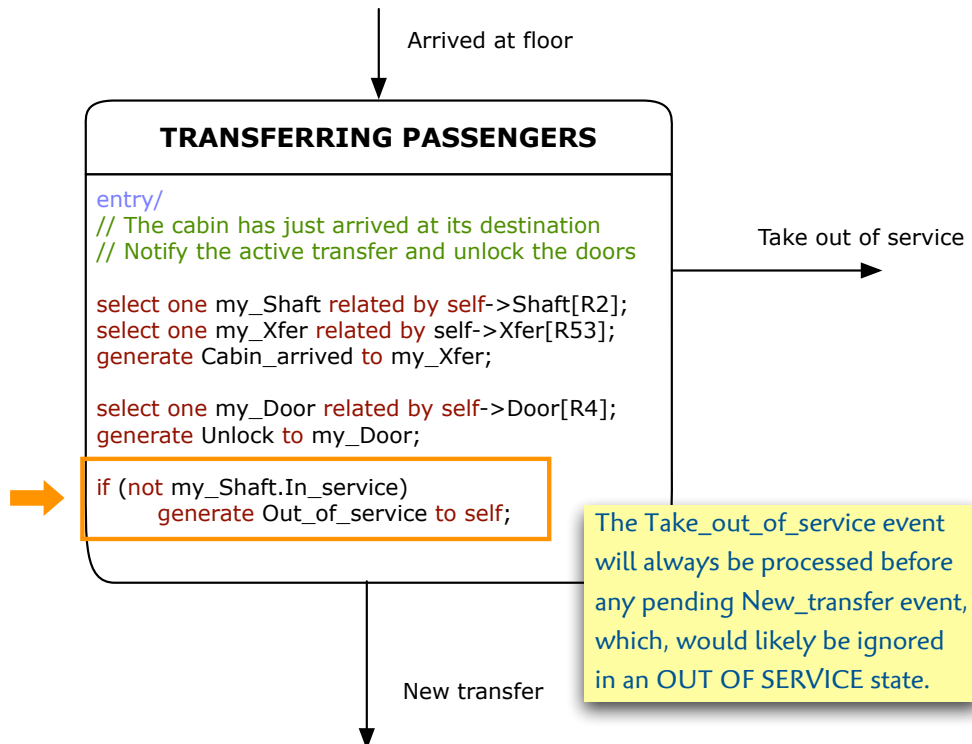
The MVM saves these as they occur. There is no queuing requirement specified in Executable UML (nor is there a need for one). It is the modeler's responsibility to make sequence explicit, not hide it in the MVM behavior. That said, most implemented MVMs queue incoming events as a matter of implementation convenience. Just don't count on it.

SELF DIRECTED EVENTS TAKE PRECEDENCE

Now having said all this, there is one exception to event prioritization which applies only to self-directed signals.

You may have noted in the previous example that an object can force itself to advance (or re-enter the same state) by sending a signal to itself. This feature is especially useful when local decisions are made as was shown below:

Self directed events are processed first



Either signal generated as a result of the if-then-else test is relevant only if processed in the current state. In fact, you may notice that an object will never wait in a state like this because a self-directed, transition triggering event is guaranteed to be waiting upon procedure completion. (My own personal convention is to use initial caps to title such states as opposed to all uppercase for potential dwell states. It would be nice if the supporting tools would use color or some other visual feature to make this distinction). So, the rule is that an object never sees a non-self directed event before all pending self-directed events have been processed.

A self directed signal is sent from an object to *itself*. This is not to be confused with an event sent from one object to another object of the same class!

Note to methodology/MVM definers: I think this is a bit of turbulence on an otherwise aerodynamic system. Why not just replace self-directed events with the concept of forced transitions? Then we could dispense with event priorities entirely. Am I right?

SIGNAL DELIVERY TIME IS UNKNOWN

When a signal is generated, it is sent immediately. Signals are never lost and delivery (in the form of a corresponding event) is guaranteed. The delivery time, however is unknown since this is platform dependent. So the modeler should not make assumptions that rely on a signal arriving within an absolute time window¹².

Implementation note

Now you may rightly assert that signals do get lost in the real world. Yes they do. But as far as the modeler is concerned they don't. The MVM is responsible for ensuring delivery. So the modeler should not add model logic to compensate for lost signals¹³. Of course, the MVM might screw up and lose a signal. For that matter memory might get corrupted, again, not of concern to your models. It is the MVM's responsibility to recover, not your application models. Typically this will involve rebooting your domain or resetting it to an earlier image. The range of soft, cold and hard restarting supported by an MVM is something that varies from one platform to another.

SIGNALS FROM THE SAME SENDER ARRIVE IN THE ORDER SENT

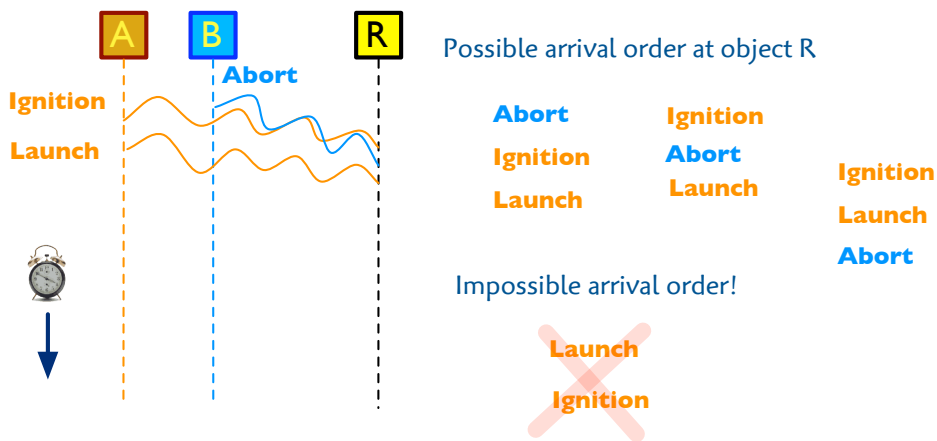
Without a global clock, there is no way to resolve the time at which two signals from different senders (objects or external entities) are issued. Consider two senders A and B. A sends **ignition** and B sends **launch** at the same time to a recipient object R. But A and B each have their own local, unsynchronized clocks. So even if each signal is time-stamped by its sender, the "same time" cannot be resolved without a global clock. So R will see either **ignition -> launch** or **launch -> ignition**.

Signals from the *same sender* will, however, be seen in the order sent. This is because timestamps from a single sender's local clock can be resolved by the MVM. If sender A issues **ignition, launch** and B sends **abort**, object R may see any of the following: **abort -> ignition -> launch** or **ignition -> abort -> launch**, or **ignition -> launch -> abort**. In each case, **ignition** is seen before **launch** by object R.

¹² Having worked on a number of video frame rate systems, I can hear the collective gasp out there! In such systems the underlying MVM on the target platform must provide mechanisms to ensure that hard time boundaries are respected and the corresponding events are triggered on time. In other devices, critical interrupts must be handled immediately. Some procedures may have to finish within critical time windows. All of this is transparent to the model level, however. This is why we don't have a one-size-fits-all MVM! Each class of platform requires its own MVM.

¹³ This is analogous to assuming that a function call in the C programming language will, in fact, invoke the function. You don't write extra C code to recover in case the function calls don't get made. Of course, you might not like the return value, but you assume that the function actually does get called.

Order is guaranteed for signals originating from the same sender



This is not unlike the situation where two friends e-mail you simultaneously. The internet does not run on global time and will deliver the messages in either order depending on various latencies. On the other hand, an individual message that you receive contains perfectly ordered text. The packets from a single sender are kept in order even if they arrive sporadically thanks to local timestamps.

The power of this highly distributed time model is that you can build models that reliably interface with an asynchronous and often chaotic world. And once tested against the Executable UML rules, these models will work on any platform. But with great power comes great responsibility¹⁴. The modeler must now test against chronologically variant scenarios. Here are a few examples to give you a sense of what is involved.

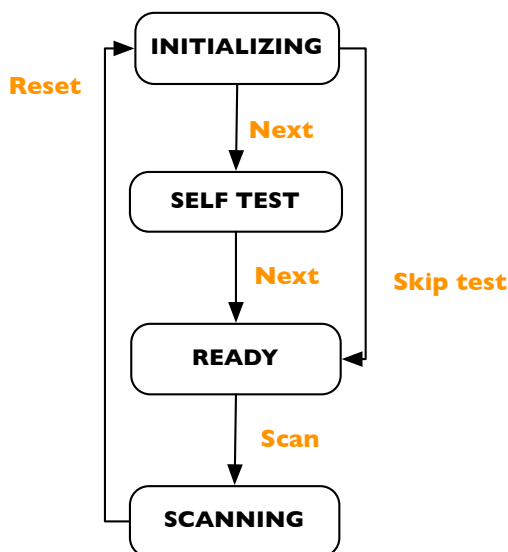
¹⁴ Spiderman

Same events, but one test passes while the other fails

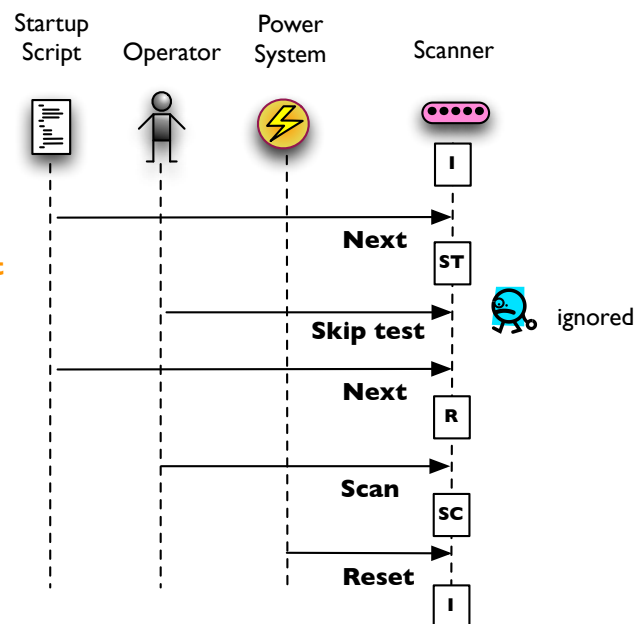
In this first example we have a medical scanning device that interfaces with a startup script, a human operator and a power management system. The operator will want to skip the self test phase and just start a scan. But the startup script is running and at any point the power system may need to reset the machine.

Correct final state in Test #1

Medical Scanning Device



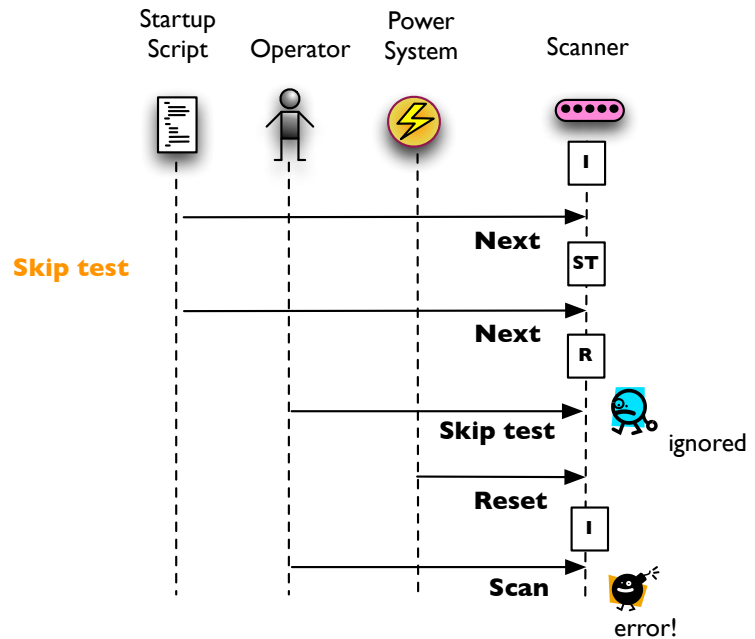
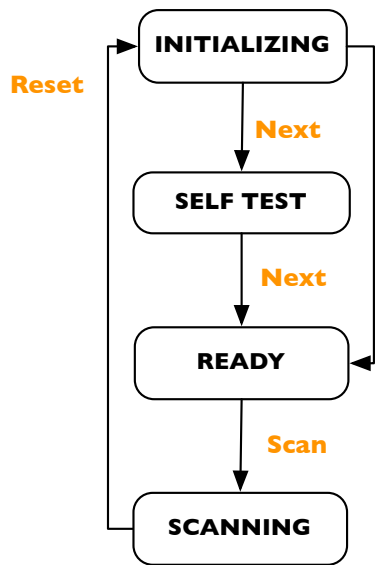
The operator is ready before the startup script has finished. But that's okay because "skip test" is just ignored if a test is already in progress. Fortunately we make it into the **READY** state before the operator's Scan event is received. Then a power problem occurs and we recover normally by re-initializing.



Assuming the device starts out in the INITIALIZING state, we will see the following state sequence: INITIALIZING, SELF TEST, READY, SCANNING and INITIALIZING again. So the idea is that the startup script is still running when the operator's commands arrive. Then, during the scan, the power system re-initializes the device. Lack of synchronization between the operator, startup script and power system is troubling. But no error occurred in this Test #1. Let's try again, possibly on a different platform (or with different tuning in the simulation) and see what happens.

Same events, but test 2 fails!

Medical Scanning Device



The exact same signals are sent as those in Test #1. This time, though, the startup script executes a bit faster so the 2nd Next event is seen before the operator's Skip Test event. That's okay, because the READY state is configured to ignore the Skip Test event if it occurs. But the Reset event occurs before the Scan command this time. The INITIALIZING state is not expecting a Scan event and registers an error when this happens. So the second test gives us an error!

There are many ways to resolve the discrepancies between tests 1 and 2. We could use a combination of events and interlocking states to keep the three external entities from stepping on one another. For example, we might queue operator commands (as objects) until the startup script is finished and then generate events as we dequeue the operator commands. We could also have the operator interface hold in some state until triggered from the READY or newly added WAITING TO INITIALIZE states. Or we could just add lots of **ignore** responses to the state table to make the state model more resilient. Be careful though, the practice of just ignoring anything you don't care about willy nilly leads to oblivious state models that don't synchronize well with each other and the outside world. Use the **ignore** response sparingly! My approach is to initialize all state-event responses to error, when I am building a new state model, and then to selectively and strategically insert **ignores** as required. Each **ignore** will carry a strong justification, which should be included in the documentation. (If you use a spreadsheet like I do, sometimes, you can add comments to the table cells).

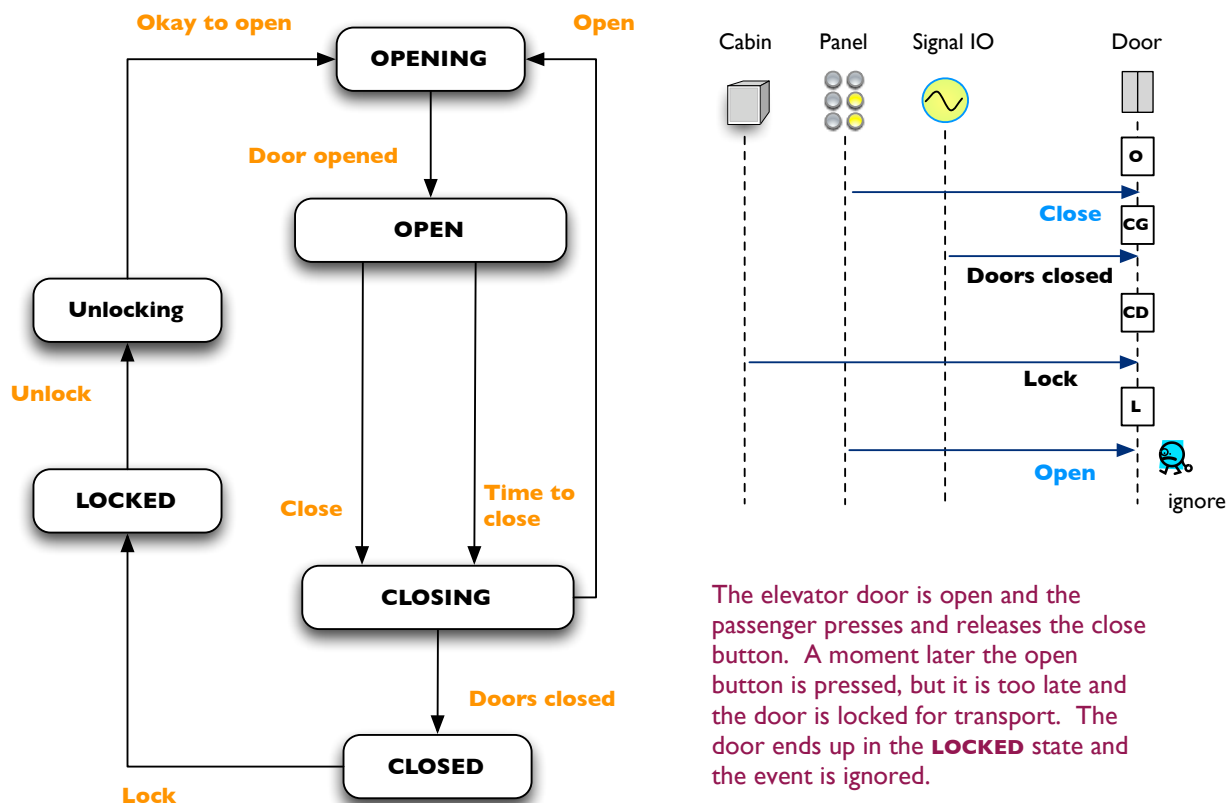
The main point here is that the modeler must remember that signals from different senders are synchronized only if explicitly modeled. Only these signals from the same sender are guaranteed to arrive in order.

Strategic use of the ignore response

In this next example it is unrealistic to coerce the input from multiple senders into a convenient sequence. We instead want our state chart to respond resiliently while accurately synchronizing with another statechart. This is our elevator door-cabin problem mentioned at the beginning of this article. We want to ensure that the cabin never moves unless the door is closed. But we can't prevent a passenger from pushing the open button just at the point that the cabin starts to move. Furthermore, if the cabin wants to move at the exact time that the user pushes the open button we can't know which event will be seen by the door first. We can solve this problem by adding a LOCKED state to the door statechart and employing the **ignore** response in all but the CLOSED state.

An elevator door will receive unsolicited and uncoordinated open and close requests from the passenger and opened and closed events from the door sensors. There will, however, be coordination to ensure that the cabin does not move until the doors are closed.

Passenger opens and closes the elevator door



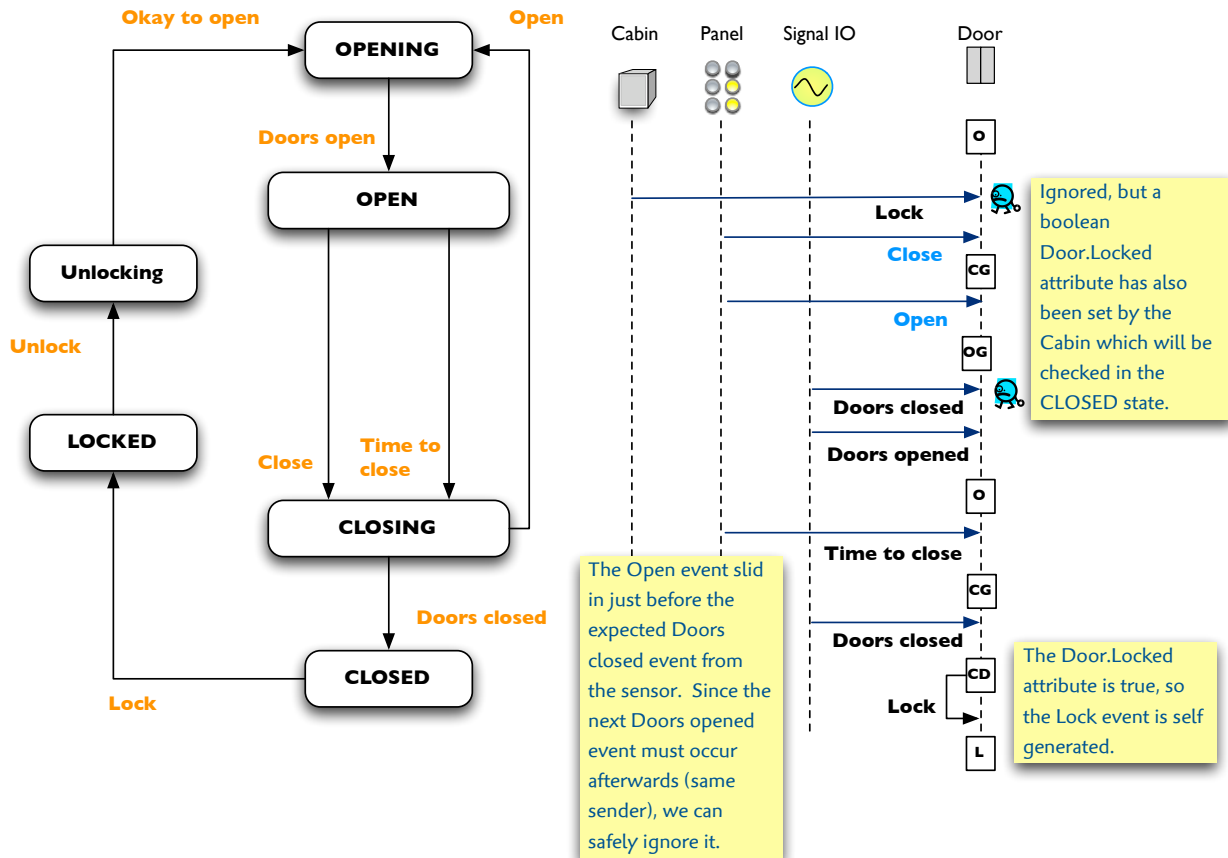
The door starts off in the OPEN state when the passenger presses the close button. The corresponding event pushes us into the CLOSING state where we wait until the sensor reports back with a Doors_closed signal. Now in the CLOSED state a Lock signal is received from the cabin (it wants to

move). The door enters the LOCKED state where user open/close requests will be ignored. In fact, the user tries to open the door, but too late, the event is ignored.

Naturally, this is only one of many possible event arrival sequences.

But what would have happened if the timing was slightly different?

Controlled chaos



Things appear to go wrong from the outset. The cabin is ready to move but may not proceed until the door is in the LOCKED state. The Lock event is received, but ignored by the door object since it is currently in the OPEN state. According to our rules, the Lock event is then discarded. Fortunately, though, the cabin (trust me,¹⁵ it did) set the Door.Locked attribute to true in the same procedure that sent the Lock signal. We will refer to that attribute value later.

The operator pushes the Close button and we see the corresponding event and transition to the CLOSING state. At this point we are expecting the Doors_closed event from the door sensor. But at the exact moment that the doors fully close, or perhaps a split second before, the passenger hits the Open button. The passenger and door sensors do not coordinate with one another, so the Open event could snake in first and put us in the OPENING state. The doors are commanded to open in the state procedure and, upon completion of the procedure, the door object sees the tardy Doors_closed event. Note

¹⁵ Or don't and just download the [Elevator Case Study](#) [5] and see for yourself!

that there is no possibility of the impending `Doors_opened` event getting here first as it is from the same sender and must arrive *after* the `Doors_closed` event. And we can't exit the `OPENING` state until we have the `Doors_opened` event, so, if the `Doors_closed` event did not get processed in the `CLOSING` state, it will get handled in the `OPENING` state. And the point is moot in the `OPENING` state, so the `Doors_closed` event is safely ignored. (Assuming we set this correctly in the state table which I did not do the first time!)

We eventually transition into the `CLOSED` state and execute its procedure (not shown) which checks the status of the `Door.Locked` attribute. Since it is true, the door object sends a `Lock` signal to itself (again, not shown). This pushes the door object into the `LOCKED` state. The attribute will be reset to false¹⁶ later in the `Unlocking` state.

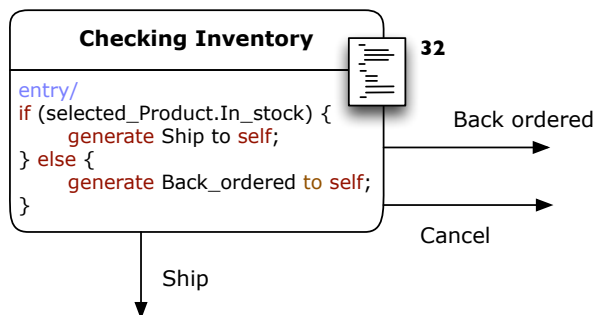
It should be clear by now that a firm understanding of the Executable UML synchronization rules is essential to addressing real world concurrency. Notation, by itself, doesn't get the job done. It should also be clear that executable modeling keeps the modeler honest. You are no longer allowed to leave the hard problems for the programmer/designer!

RESOLVING AMBIGUITY

Now let's go back to the original examples and interpret them using the Executable UML synchronization rules.

Example 1: Busy when event occurs

Order 32 is executing the `Checking Inventory` procedure when a `Cancel` event occurs. What happens?



We know that a busy object does not see incoming events. So the state procedure will execute and produce either the `Ship` or `Back_ordered` event. Assuming the `Cancel` event occurs during this very short time window, it will be held by the MVM. We also know that self directed events are selected and presented by the MVM first. So **Order 32** will always exit on a `Ship` or `Back_ordered` event, since each triggers a transition. The `Cancel` event would be retained in the event pool until some future state where it would most likely be ignored.

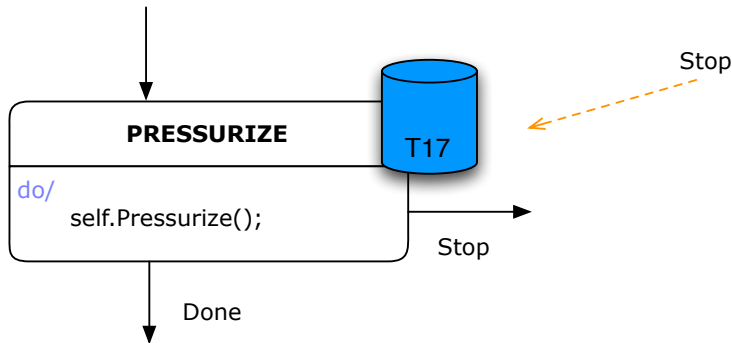
Since we know that we can't interrupt a procedure with an event, we wouldn't build the statechart this way. We would probably intercept the `Cancel` event before or after the `Checking Inventory` state. Another idea would be for some operation or external function to set a boolean attribute on the `Order`

¹⁶The cabin and door statecharts are coordinated so that there is no possibility of a write conflict on the `Door.Locked` attribute. Download the [models](#) [5] if you are interested in seeing more detail.

class such as `Order.Cancelled`. Then, in a subsequent state, we could check it: if `self.Cancelled`, generate `Cancel` to `self`; In any case, we would certainly remove the `Cancel` transition from this example.

Example 2: Granularity of interruption

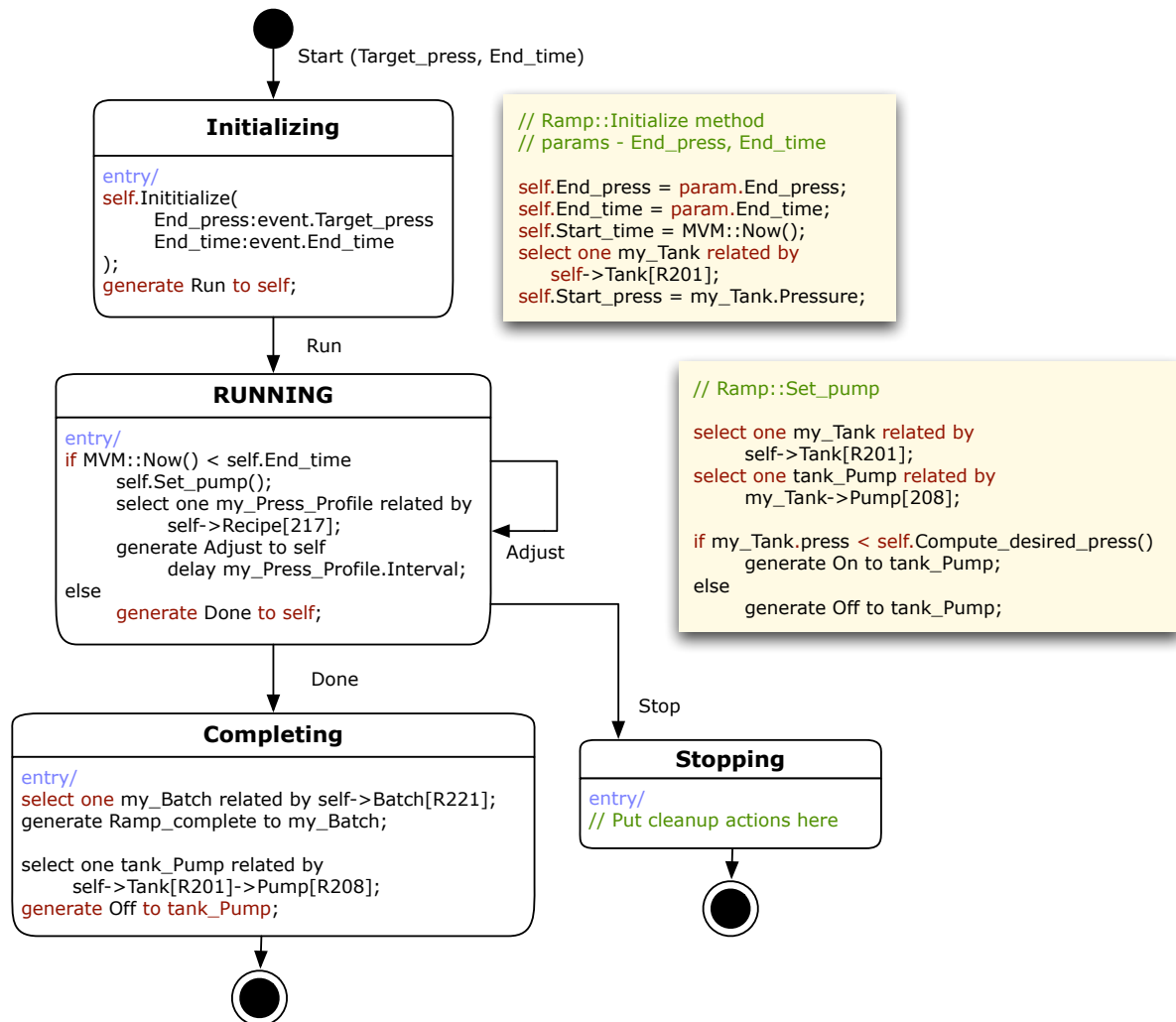
To answer our original question about granularity of interruption for a continuous process, we can break this single state example into discrete components.



This will result in a more controllable but bigger statechart. Don't worry though, because we can then package the more complex statechart in a service domain and just start and stop it from our application domain¹⁷. First, we will define the behavior of a `Pressure Ramp` class.

¹⁷This is one of many easy ways to avoid the unnecessary complexity and arbitrary leveling of statechart hierarchies.

Pressure Ramp Class



The Pressure Ramp is created by specifying a target pressure and some future time. It immediately starts up and adjusts the pressure by turning a pump on and off until the completion (end) time.

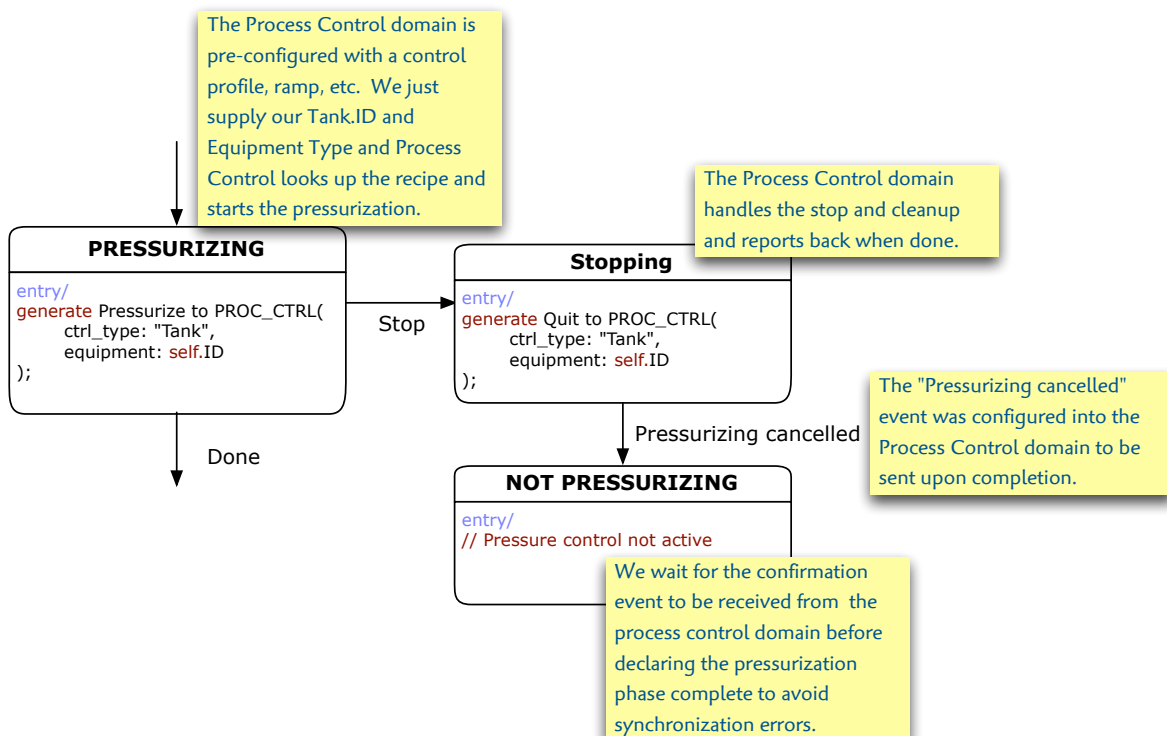
The procedure's actions are organized into methods, shown on the right, which are invoked from the appropriate state. This keeps our statecharts more readable, printable and testable. State actions should focus on control with the data processing and computation encapsulated in the methods.

As you can see, we can interrupt the pressure ramp with the Stop event. If we are busy executing the RUNNING procedure, this event will be pooled until after we generate the delayed Adjust event. At this point we see the pending Stop event and transition out cleanly. This is a nice solution since we don't corrupt any data on the way out. Keep in mind that the MVM can shut us down even in the middle of a state procedure if, say, someone hits an E-Stop button. Normally, we would gracefully interrupt by sending a Stop signal, though. So it is the modeler's responsibility to anticipate real world interrupts and provide the appropriate cleanup mechanism if the interrupt can reasonably be handled in the application domain.

The Pressure Ramp statechart may be more eloquent, but the original example was concise and intuitive. We should, and can, get the best of both worlds. First we recognize that the Pressure Ramp concept is not specific to pressure. If we wanted to control temperature instead, we could globally replace “Press” with “Temp” on our statechart. So the renamed Ramp class could be subsumed as part of a generic process control service domain that could then be configured and directed from our higher level (client) Pressure application domain.

Along with the Ramp, we would have loaded appropriate control profile data into the Process Control domain, and bound it to corresponding entities in the Pressure Application domain. Now we can create the following simplified statechart in our application.

Tank class



The entry procedure in the tank’s **PRESSURIZING** state triggers pressure control feedback activity in the Process Control service domain. The triggering event does not specify a ramp since the whole purpose of a domain separation is to insulate the analysis and implementation of one domain from another. So we just specify our type “Tank” and ID value knowing that the bridge to the Process Control domain maps¹⁸ to the relevant content, a ramp and any control profile data in this case.

We then sit in the **PRESSURIZING** state until we get a `Done` event (triggered by completion of the Ramp or whatever mechanism is running in the Process Control domain). Alternatively, a `Stop` event occurs which puts us in the **Stopping** state where we tell the control mechanism to quit. (Another way to do it is to route the `Stop` event directly to the Process Control domain and just have a `Stopped` event bubble

¹⁸ Future article on analysis domains and bridges planned.

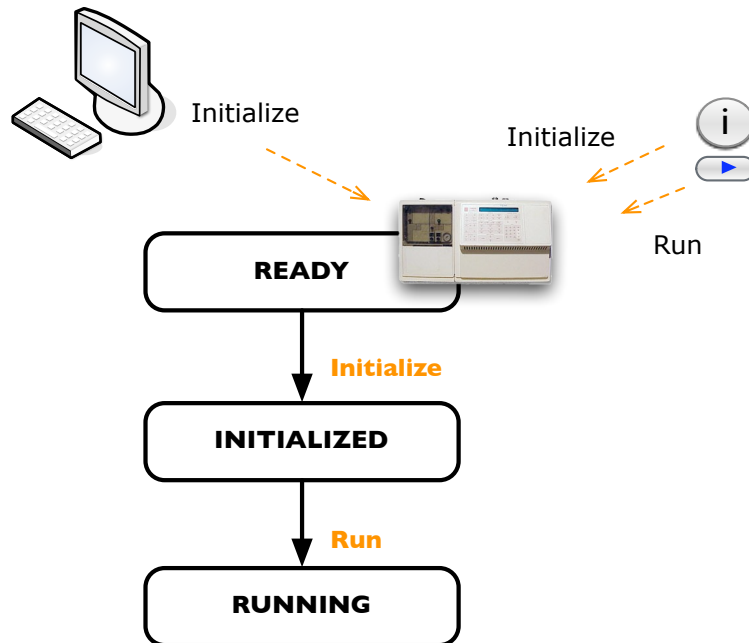
up to the Application domain. Either way, we end up in the, for lack of a better name, NOT PRESSURIZING state.

Now the application reflects the relatively high level view of the Tank, the Process Control domain handles ramps, profiles, control curves and such with an even lower level Signal IO domain managing the analog/digital interfaces to the sensors and actuators. Bridge'em all together and we have ourselves a system!

So with the availability of domain packaging and clear synchronization rules, we dispense with the need for special **do/** semantics in Executable UML. The **entry/** procedure¹⁹ is all we need.

Example 3: Sequencing events

In this last example we have a scientific measurement instrument commanded from both an LCD button panel on the front of the station and via a remote workstation connected on a network. A human operator presses the Initialize and Run buttons in sequence. Simultaneously, another Initialize command arrives from the network. What happens?



We know that the Initialize event from the front panel will be seen before the Run event. But the Initialize event from the Network could slip in at any point. An object sitting in the READY state will certainly see one of the two Initialize events first and then transition to the INITIALIZED state. At this point, either the Run event or the second Initialize event will be seen. If **ignore** is specified as the response in the INITIALIZED state (likely), the second Initialize event will be dropped, then Run will be seen followed by a transition into the RUNNING state. If, instead, the Run event is detected second, the object will transi-

¹⁹You can define an Executable UML with exit/ procedures only and might be just as good. There are many equivalent state machine formulations (Mealy/Moore), etc out there. The point is to keep it simple. Inclusive use of both entry/ and exit/ actions and incorporation of state history, while we're at it, is a recipe for needlessly confusing models.

tion to the RUNNING state where the 2nd Initialize must be processed somehow (**transition**, **ignore**, **error**).

Now is this the desired behavior? Without the requirements at hand, you don't know. But you do know precisely how this model will behave given a specific scenario.

MAKING YOUR OWN RULES

The Executable UML described in this article is documented in a number of books and supported by multiple vendors. More importantly there are at least a dozen MVMs and model compilers that support the rules described in this article (including two running on my laptop). See the tool references at the end of this article for more information. Additionally, several more specialized MVM's and model compilers have been built which are custom built internal to companies that deliver embedded and distributed systems.

While Executable UML defines a robust, simple and time tested set of rules, you are free to invent your own. Here are some pro's and con's to consider.

Pros

- + Adapt to special circumstances in your application or platform. However, you'll probably get more leverage by directing your talents at a better model compiler / MVM implementation.
- + Adapt to the capabilities of non-Executable UML modeling tools.

Cons

- There is a huge investment of time and effort to ensure that the rules are internally consistent and have no unforeseen consequences. Plus you'll have to write yourself a book describing the rules for any future modelers.
- If you change the timing and synchronization rules, you lose compatibility with any existing Executable UML model compilers, model level debuggers and MVMs. So you will have to build these as well.
- All my example models will break if you change the rules! So you'll need to write your own tutorials for prospective modelers ;)

If you do decide to define your own Executable UML, the OMG has recently published a guide to profiling UML accordingly [8]. Good luck!

The important thing is that you somehow arrive at a set of timing and synchronization, data access and other rules agreed upon by both the modelers and the platform developers. Ultimately you need to decide whether you want to devote your ingenuity to modeling the application, defining a new methodology or designing an efficient platform. I've made my choice to focus on the first item since there seems to be plenty of bright folks focusing on the other two.

Summary

Executable UML adds executability to UML by introducing a number of features including platform independent time and synchronization rules. Other features necessary to make UML a full fledged development language are a precise underlying data model for the class diagrams and a rigorous action language.

A key characteristic of the time and synchronization rules is that concurrency is the default paradigm. We assume that each object operates in its own local timeframe. Procedure sequencing and state synchronization must be explicitly modeled and tested. You could rightly say that Executable UML comes with no “artificial sequencers”²⁰. This makes it possible for a single modeled domain to run on a wide variety of target platforms. There is no need to tweak the models to take advantage of added available parallelism since the models are already tested to run with maximal concurrency. The more parallel your platform, the more efficiently the models may run.

An MVM is necessary to run Executable UML models. Ordinarily, the requirement for a VM implies a significant burden on the implementation. In reality, though, the MVM can be custom built for various classes of platform. (One common misconception is that a separate MVM must be created for each platform. Not true, especially with a data driven MVM design!) The MVM implementation on each platform class, however, will be quite different. A highly embedded MVM may optimize for small event queue sizes, assume fixed instance populations and even provide a way to explicitly bind instances to absolute memory locations. A highly distributed MVM will may manage a sophisticated threading, fault tolerance and multiprocessor scheme.

But the same time and synchronization rules are supported regardless of MVM implementation. Some MVM’s may support a variety of marking²¹ features so that various model components in an Executable UML domain can be compiled more efficiently. Some model compilers take into account initial data populations in addition to the models and markings to fully optimize the implementation.

The MVM of a desktop simulation environment, in particular, will be quite different than that of a highly distributed or highly embedded platform.

Since synchronization problems can be definitively resolved, serious models can be built. This also requires a rigorous action language to define the state procedures. It’s a topic for another article to be sure. But parallelism available at the state level can be (and is) carried into the procedures so that multiple actions run in parallel as data dependencies permit.

With the support of Executable UML, intellectual effort can be focused on resolving complex systems at the application level. The resulting distillation of intellectual property is highly portable. Organizations that produce spin-off products from an initial design should benefit greatly.

On the down side, a whole new set of development skills and mindset is required to build Executable UML models. The ability to focus on an application without getting caught up in issues that are the domain of the MVM and platform is difficult for many developers. Also the ability to think and test in concurrent terms takes a bit of practice. That said, the days of sequential thinking in system design are long gone.

²⁰ New coffee mug idea - I’m on it...

²¹ The “marking model” is an MDA concept. See the MDA Distilled book [2] for a good explanation.

Published Resources

- [1] [How to Build Articulate UML Class Models](#), OMG uml.org and Google Knol, Leon Starr, 2008
- [2] [MDA Distilled](#), Principles of Model Driven Architecture, Mellor, Scott, Uhl, Weise, Addison-Wesley, 2004, ISBN 0201788918
- [3] [Executable UML, A Foundation for Model Driven Architecture](#), Mellor-Balcer, Addison-Wesley, 2002, ISBN 0201748045
- [4] [Time, Clocks and the Ordering of Events in a Distributed System](#), Leslie Lamport, Communications of the ACM 21, 1978 (yes, really, 1978!)
- [5] [Elevator Application Executable UML Case Study](#), Leon Starr, Model Integration LLC, 2001, 2005
- [6] [Model Driven Architecture with Executable UML](#), Raistrick, et al, Cambridge University Press, ISBN 0521537711
- [7] [Executable UML](#), Wikipedia entry
- [8] Semantics of a Foundational Subset for Executable UML Models, 2nd Revised Submission, OMG, Aug 2008 (I have not been able to find this on the web yet, sorry). I had my copy e-mailed to me by Steve Mellor. Will update the online version of this article as soon as I find it.
- [9] [Semantics of a Foundational Subset for Executable UML Models \(FUML\)](#), OMG, 2008. (Seems to be a subset of document [8]) <http://www.omg.org/spec/FUML/>

Tool and Vendor Resources

If you are interested in tools that support Executable UML, here are a few:

[Nucleus BridgePoint](#), Mentor Graphics, http://www.mentor.com/products/sm/uml_suite/

[iUML](#), Kennedy-Carter, <http://www.kc.com/>

[Pathfinder Solutions](#), <http://www.pathfindermda.com/>

[OOA Tool](#), Kavanagh Consultancy, <http://ooatool.com/Products.html>

For my own work, I use BridgePoint mostly. But I work with whatever tools my clients are using including Artisan, Rose-Rhapsody (IBM).