Using the Actor Framework in LabVIEW

By Allen C. Smith and Stephen R. Mercer

The Actor Framework is a software library that supports the writing of applications in which multiple VIs run independently while communicating with each other. In these applications, each VI represents some actor carrying out an independent task in the system. The actor maintains its internal state and sends messages out to the other actors. Many techniques exist for creating such applications in LabVIEW. The Actor Framework focuses on being easy to learn (relative to other possibly-more-powerful tools) while mitigating risk of deadlocks/race conditions and maximizing code reuse.

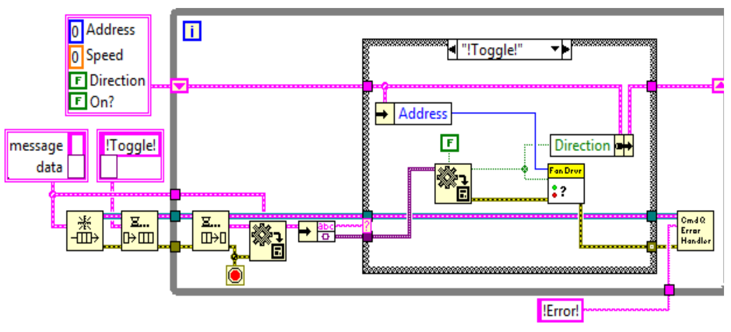


Figure 1: A Sample Queue-Driven State Machine

We created The Actor Framework to fill a hole we observed in the standard architectures used by LabVIEW programmers. The Actor Framework builds on the well-known queue-driven state machine[1] (QDSM) model. The QDSM is one of two patterns used in the majority of LabVIEW applications (the other being the Producer/Consumer model). Individual QDSMs are well-encapsulated software modules that can combine to create large systems with rich behavior. Many LV-written applications do this successfully, but we observed two common pitfalls. First, although the QDSM is good for defining a module, we noted that the modules themselves often have limited reuse potential. In many applications, a state machine written to represent one subsystem or piece of hardware would be duplicated in its entirety to support another similar subsystem or piece of hardware. Second, every application framework that we investigated had at least one potential timing-related bug in its communication scheme. When individual programmers create a messaging scheme, they often introduce subtle errors involving resource contention, excessive coupling between modules, missed messages or deadlocks.

By applying some common object-oriented techniques and design patterns to the original queue-driven state machine, we created a framework that provides:

- all of the benefits of QDSM design,
- significantly increased flexibility,
- more reuse potential, and
- reduced coupling between modules.

.

Basic Actor Framework

The Actor Framework is a set of classes that provide common state and message handling functions.

The model provides the following two parent classes from which you create child classes:

- **Actor**—State data in a module.
- **Message**—Messages passed between actors to trigger state changes.

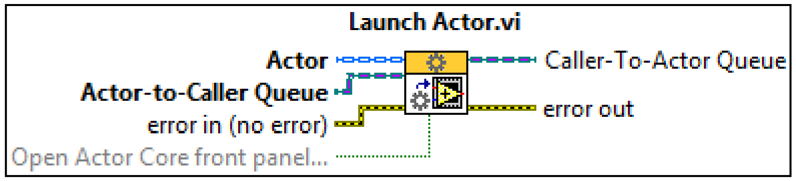
The Actor Framework includes two specific Message child classes called **Stop Msg** and **Last Ack**. These child classes are particular messages used when shutting down actors. Additional Message child classes are distributed with the framework, but are not included in the framework library itself since they not required in all applications. These additional message classes support optional use cases.

[1] Also known as a Queue-Driven Message Handler.

Under the hood, the framework makes extensive use of LabVIEW's queue functions. The public API, however, hide the raw queues using two classes: **Send Queue** and **Receive Queue**. These classes are wrappers for standard LabVIEW queues of type Message. They are used to enforce the direction of of framework messages. When a callee actor creates its queue, it wraps it in a **Send Queue** object and returns that object to its caller. The caller can only enqueue messages on this queue. The caller cannot dequeue from the queue and cannot release the queue. Similarly, **Receive Queue** can only dequeue messages, never enqueue them. It is not used within the framework, and is provided for symmetry. By limiting what can be done to the queues in various parts of the code, the framework is able to make guarantees about timing and message delivery that otherwise could not be proven.

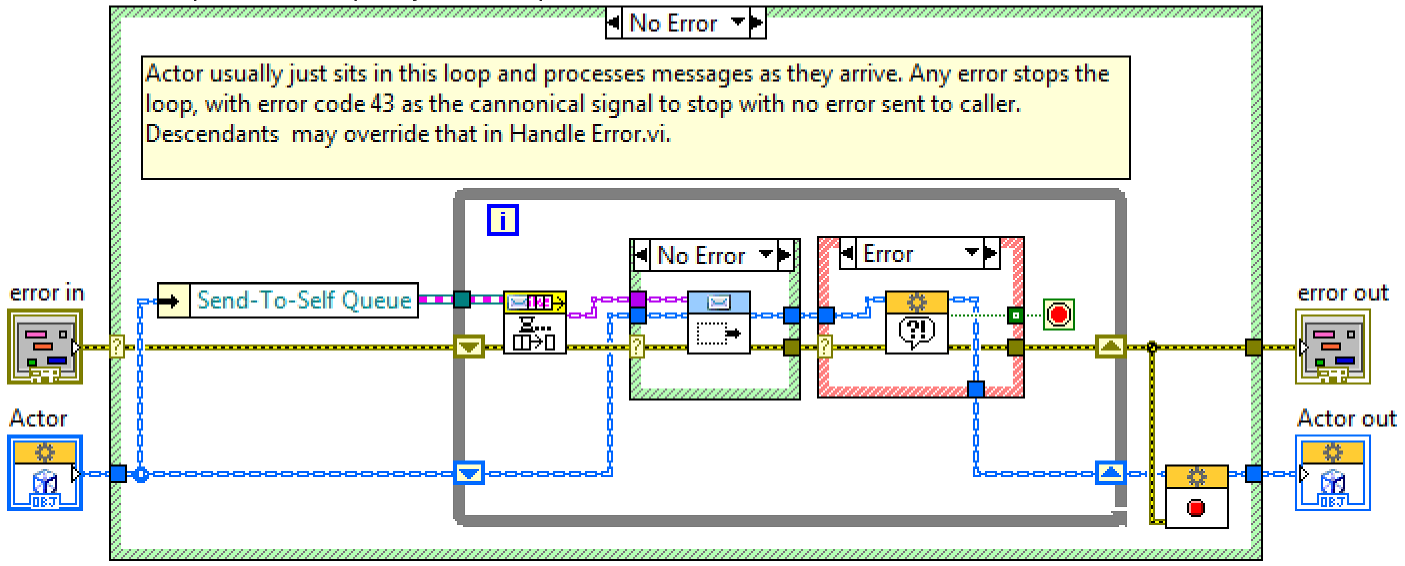
Actors

In the Actor Framework, an actor is a LabVIEW object that represents the state of an independently running VI. Because the actor object is shielded inside the running VI and all outside access is done by sending messages to the VI, the term "actor" is also used for the VI itself. The flexible usage of the term becomes natural once you understand how tightly paired the actor object becomes with the VI hosting it. All actor classes inherit from **Actor**. An actor object is a normal by-value, dataflow object with accessors and methods. You can make any existing by-value class into an actor simply by setting it to inherit from **Actor**.



Actor provides exactly one public method called *Launch Actor.vi*. This method uses VI Server to launch an independent reentrant clone of the private method, *Actor.vi*, which defines the QDSM. Launch Actor.vi has two required inputs: the initial actor object and a queue reference. The VI returns a new queue reference as output. You use the input queue to pass messages from the launched actor (the callee) back to the caller. You use the output queue to pass messages from the caller into the callee. Note that Launch Actor.vi does *not* return the **Actor** object; it is passed from *Launch Actor.vi* to *Actor.vi* and is no longer available to the caller. All further interaction with the object is done by passing messages into the queue.

Actor.vi calls *Actor Core.vi*, which is a protected scope, dynamic dispatch VI:



No Error
Actor usually just sits in this loop and processes messages as they arrive. Any error stops the loop, with error code 43 as the cannonical signal to stop with no error sent to caller. Descendants may override that in Handle Error.vi.
error in
Actor
Send-To-Self Queue
No Error
Error
error out
Actor out

Figure 2: The eldest implementation of Actor Core.vi

You should recognize *Actor Core.vi* as a queue-driven state machine. As each message arrives, the message invokes methods on the actor (see "Messages," below). An error will stop the state machine, although special consideration is given to error code 43, Cancel Error, which does stop the machine but does not propagate up to callers.

A child class can override Actor Core.vi, though the override VI must invoke the Call Parent node. Because the Call Parent node runs the state machine, it should be in parallel, not serial, with other operations you add in the override VI. The override VI can include a user interface, calls to launch nested actors, or additional control logic for the state machine. You will find more details about these overrides later in this document.

User Interfaces for Actors

The Actor Framework does not provide the method VIs of your actor with any directly access to controls, indicators, or parallel loops that might be part of its override of *Actor Core.vi*. You must define that access within your actor.

Here are two easy ways to define that access within your actor:

1. Include a reference in your actor's private data to either your *Actor Core.vi* or to its front panel objects. Bundle these references into the actor before invoking the Call Parent node. Your actor can then access these references from within any of its methods. This method allows your methods to control the user interface of the *Actor Core.vi*, although it may not be the most efficient approach. This approach works well for interfaces that are only displaying information and not reacting to users' actions.
2. Create a set of user events to manage your front panel updates. Create these events in your *Actor Core.vi* and bundle them into the actor object prior to invoking the Call Parent node. Then create a loop in your *Actor Core.vi* that runs in parallel to the Call Parent node which is dynamically registered for the events. When the actor receives a message, the handling function for that message can generate the appropriate event and pass along any relevant data. Alternatively, you may choose to use queues, notifiers, or other mechanisms to communicate with your parallel loop rather than events. Regardless of which mechanism you prefer, we recommend that you select only one data transfer mechanism within any given actor, and keep the total number of such mechanisms to a minimum within the application as a whole. Be sure to provide a mechanism to stop your parallel loop and trigger that mechanism by overriding *Stop Core.vi* in your actor child class.

Messages

Use message objects to modify the state of an actor object. Message objects take the place of individual cases in a traditional queue-driven state machine. Message objects inherit from **Message**, and must override the *Do.vi* method. Child Message classes usually include some attribute data and a *Send <message>.vi* method. Depending on the application, there may be additional methods for reading or writing the message data.

The queues passed to and returned by Launch Actor.vi take messages as their data type. A VI that needs to interact with an actor must have access to the actor's send queue. The VI creates a message of the desired type and posts it to the actor's queue. Typically the VI accomplishes this by invoking the relevant *Send <message>.vi*. *Send <message>.vi* combines creating the message, writing its data, and posting it to the queue in a single call. You can perform these tasks separately if desired.

You must provide a message for every method of your actor that you want to access remotely. A message typically represents a call to one method of the class, and the message class' private data is the input parameters needed to invoke that method. When your actor receives a message, it invokes that message's *Do.vi* method. *Do.vi* in turn invokes the appropriate operation on the actor. Most applications have a one-to-one correspondence between a message class and an actor's method, but there are rare cases where a single message represents multiple operations on the actor. Creating these message classes may sound tedious, but the Actor Framework includes a scripting tool, discussed later, to eliminate the busy work.

You can use messages you create for a specific actor class with any child of that actor. For example, consider a fan controller that exposes a public method *Toggle On.vi* for turning the fan on or off. The programmer creates a corresponding message class, **Toggle On Msg**, that the controller's caller can use to invoke that method. Figure 3 shows the *Do.vi* for **Toggle On Msg**.

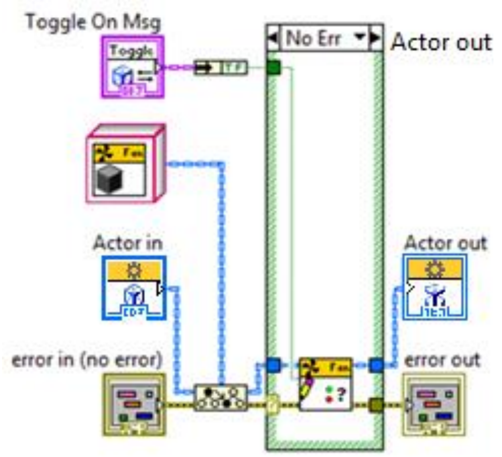


Figure 3: Toggle On Msg.lvclass:Do.vi

What about outputs from the methods? The more you work with the Actor Framework, the more you will find yourself creating many methods that have no outputs other than the actor object and error out. This is because the message passing environment encourages methods that are more about reacting to information sent by other objects and less about requesting information from other objects. Often, instead of having output parameters, the method itself will send a message to its caller or other actors as part of its execution.

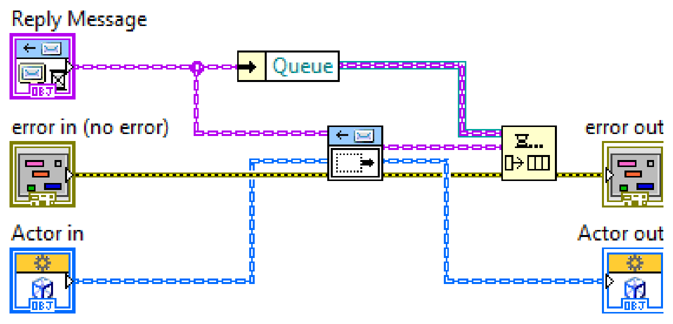


Figure 4: Do.vi method sending reply to caller

Having said that, needing to pass data back to a caller after a message is finished being handled is not uncommon. If the actor method does have outputs, those can be packaged into a separate message to be passed back to the caller. The actor object has a method called *Get Queue For Sending To Caller.vi* which *Do.vi* can invoke to return data to the caller. This assumes, of course, that the message came from the caller originally. If the original sender was not the caller, the sender may provide an instance of **Self-Addressed Message** in the original request to use for the reply. In either case, the caller/sender hears this reply asynchronously. In other words, the caller/sender is not waiting for a reply but is instead proceeding with its own work. The use of asynchronous messaging between actors is strongly preferred within the Actor Framework. Refer to the **Synchronous Reply Messages** section below for information about actor interactions requiring synchronous replies. More information about replying to messages is discussed in the **Messages from Callee to Caller** section, below.

Message Types

The Actor Framework ships with six message types.

1. **Message** — The parent message class. Only messages that are children of this class are valid messages within the framework. An actor that receives an instance of **Message** will take no action, so this can be used as a null message. Descendants of **Message** must override *Do.vi*.
2. **Stop Msg** — Tells an actor to shut down. The actor will stop handling further messages and will perform any cleanup work defined in its *Stop Core.vi*. If your actor launches any nested actors in its *Actor Core.vi* override, your actor should also override *Stop Core.vi* and pass the stop message along to those nested actors. Stop Msg has two different Send methods – one sends the message at Normal priority. The other sends the message at Emergency priority, which is even higher than the High priority. This is the only message in the framework that can ever be sent at Emergency priority. Review the Context Help for the Send Emergency Stop.vi for further information.
3. **Last Ack** —Gives a caller the final state of an actor that has shut down. The Last Ack.lvclass carries the callee's final error (if any) and the final value of the actor for use by the caller.
4. **Batch Msg** — Collects several other message objects together into one so they are sent to an actor in a single atomic burst. *Send Batch.vi* takes an array of message objects and bundles them into a single message. The *Do.vi* of **Batch Msg** iterates through the array in a For Loop. This guarantees that no other messages can be enqueued in between the batched messages, which can be a concern if you enqueue each message individually.
5. **Self-Addressed Msg** — Wraps a regular message and the send queue of the actor that receives the message. *Address Message.vi* creates the wrapped message and records the queue for the message to be sent back along. *Send Self-Addressed Msg.vi* sends the preconfigured message to the specified queue, making this message type completely self-contained. Use it to send messages from a source that has no knowledge about the recipient.
6. **Reply Msg.lvclass**—Defines synchronous messages. Refer to the next section.

Synchronous Reply Messages

When a message requires that your caller wait for a synchronous response to the message, you can create a new message class that inherits from **Reply Msg**. A reply message is sent using *Send Message and Wait for Response.vi.* It bundles a single-use queue into the message and then waits for a reply on that queue (with provision for timing out if desired). Usually your new message class will have its own Send <message>.vi which wraps *Send Message and Wait for Response.vi*. Your new message class will override *Do Core.vi* instead of *Do.vi* and return a message for the caller. Figure 4 shows an example of how the **Reply Msg** *Do.vi* passes that message back to the caller.

We strongly recommend that you do not override *Do.vi* of the **Reply Msg** class. Instead, override the protected *Do Core.vi*. Have *Do Core.vi* call the appropriate method of your actor and whatever other functionality you require, then return a suitable message to send to the caller. The default behavior returns an instance of **Message**. You do not have to override *Do Core.vi* if the only feedback your message requires is acknowledgement of receipt.

Using **Reply Msg** creates the potential for deadlock situations. For example, if the caller sends a synchronous message to the callee at the same time that the callee sends a synchronous message back to the caller, the application will deadlock. To minimize the risk, use **Reply Msg** sparingly. If two actors are able to communicate with each other (as when one is the caller and the other is the callee), if one of them can send reply messages, avoid having any reply messages on the other. Also take care to avoid cascading reply messages, where the caller's reply causes the callee to send a reply message, as this "echo chamber" effect can lead to infinite message spawning.

Messages from Callee to Caller

When a caller wants to send a message to the callee, the caller knows the type of the actor that it launched, so caller can easily choose the right type of message to send. But when an actor wants to send a message to its caller, the actor does not necessarily know what type of caller it has, so it does not know what type of message to send. Ideally, actors should be usable by many different callers. That maximizes code reuse. But making the actor truly independent of the caller requires more effort than is warranted for the situation. We identify three techniques for defining messages that an actor sends to its caller caller:

1. **The High Coupling Solution** — In this solution, the callee actor is written specifically for one type of caller, and can never be used with any other type of caller. In this case, the callee knows details about the caller's interface, so it may simply call the Send Message VI of the appropriate message. The message type is thus hardcoded into the actor.
2. **The Low Coupling Solution** — This solution works best when you have a known inheritance hierarchy of callers. In this solution, you create a matching hierarchy of callee actors. So suppose you have callers **Hard Drive**, **Air Conditioner**, and **Fire Suppression**, each of which needs to launch a **Fan** actor. You would create the **Fan** class, and then inherit from it **Fan For Hard Drive**, **Fan For Air Conditioner**, and **Fan For Fire Suppression**. Each caller launches its specific callee actor. All of the main **Fan** class' code is shared, but when it comes time to send messages, the **Fan** class has a dynamic dispatch method for doing the sending, and each of the children overrides that method to send the message appropriate for its caller. Each callee is still coupled to a specific caller, but the system as a whole is flexible to accommodate new caller types.
3. **The Zero Coupling Solution** — In order to make the callee actor independent from the caller, the caller must tell the callee what types of messages to send so the callee avoids picking a type itself. The best implementation of this solution has the caller record a message into the callee at the time the callee is launched. The callee provides a *Set <Type> Message.vi* method, where *<Type>* is the particular event that will trigger the message to be sent. The caller sets the exact message it wants to receive when this event occurs. When the event happens, the callee sends the chosen message, without any knowledge of what kind of caller is receiving that message.

   Often the callee will define an abstract Message class that it uses as the input type for *Set <Type> Message.vi*. Callers create their own specific child of this abstract class. This setup gives the callee a way to set data into the message through the API defined by the abstract class, and gives the caller a way to deliver that data in a form the caller can consume through the various overload and extension VIs.

Considerations for Implementation

Debugging Actors

Every actor runs on its own. Stopping one actor won't stop the others unless you have set up messages to go through the tree and shut everyone down. If you work with the framework for any time at all, you will at some point accidentally leave an actor running without its panel open. Your VI Hierarchy will be locked because it is running, but there's absolutely no way you'll be able to get that Actor Core.vi to pop open so you can click the Abort button. There are tools to help with this frustrating situation.

The first quick workaround is to close your project and re-open it. You do not have to restart LabVIEW as long as you're working inside a project. Closing the project will abort all the running VIs in that project.

A better solution is to use the optional input on Launch Actor.vi for **Open Actor Core front panel? (F)**. Wiring True to this input will make the front panel pop open when the actor is launched, giving you access to that actor's own Abort button. Using this terminal in release code is not recommended (for reasons explained in the Context Help of Launch Actor.vi), but while debugging, this can be a major headache saver.

As far as debugging the actual logic of your applications, figuring out where a given message came from and whether or not Message A on Actor X was handled before or after Message B on Actor Y can be tricky. Before you try building some sort of event logging system, please check out the Desktop Execution Trace Toolkit or the Real-Time Execution Trace Toolkit from National Instruments. This tool has deep knowledge of LabVIEW and its APIs, giving you insight into execution ordering, memory allocation and error propagation. If you do not have access to these toolkits, you may find these “timing probes” useful: https://decibel.ni.com/content/blogs/EvanP/2010/10/04/simple-sexy-labview-timing-probes

Actors in Libraries

Use libraries to manage and distribute your Actor classes. In addition to the child actor class, the library should contain most messages that target that actor and any additional tightly coupled classes, such a configuration object for the actor (refer to the **Configuring Actors** section, below), or the component objects of a composition.

You may opt to bundle a family of Actors in a single library, especially where that family shares a single set of messages. Bundling messages in the library gives you the option to mark some messages as private; this should be done for any messages that are used exclusively by classes in your library, such as messages that the actor sends to itself from its own *Actor Core.vi*.

Because a library in LabVIEW always loads all of its contained libraries into memory, you should package into your library only those message classes that are necessary for the actor itself to operate or are so commonly used as to warrant including. The Actor Framework library itself does not include the **Batch**, **Reply**, and **Self-Addressed Msg** because they are optional components of the framework and you can write many applications that use no such messages.

Configuring Actors

Often, you want to configure an actor with data from elsewhere in the system, such as in a configuration file. Before calling Launch Actor.vi, you can use the methods of the actor class to directly set values into the actor object. After calling Launch Actor.vi, you can give data to the actor object through its message queues. Although you could create multiple messages to set individual fields of the actor, consider creating a single Configuration class that encapsulates the configuration information. You can set the configuration object through normal method calls, then pass a single configure message to the Actor. These “large blocks of data” messages should be reserved one-time initialization of the object, not for updating the object while your application is running. While running, you should use messages that update only the fields that have changed rather than rebroadcast the entire object.

Generally, the Actor should never provide its Configuration as returned data, as this can lead to race conditions. Refer to the **Avoid Get and Set Messages** section, below for more information. If you want to cache configuration data from a shut down actor, obtain that data from the final value the actor returned in the Last Ack message. The actor may have a *Get Configuration.vi* which is simply not exposed as a message.

Implementing Actors Composed of Multiple Actors (Composition and Aggregation)

In LabVIEW, a data object can be described composed of other data objects when those other objects are part of its private data. Actor objects are generally only accessible through their message queues; in this case, an object is composed of other actors when it has the actors’ message queues as part of its private data. Generally, the composed object will itself be an actor, hereafter called the caller actor. The components are callee actors. There are two styles of caller actors: compositions and aggregations.

In compositions, the caller actor is responsible for creating and destroying its callee actors. The standard implementation is to create, configure and launch the callees in the *Actor Core.vi*, prior to starting any While Loops or invoking the Call Parent node. The caller actor stores the **Send Queue** returned by *Launch*

Actor.vi for each component as part of its private data. You will need to override Stop Core.vi in the caller to send stop messages to each callee so they all shut down together. A slight variation on this style has the caller actor create its callees dynamically in response to a message received.

In aggregations, the callee actors are created separately from the caller actor. The callees' queues will be given to the caller rather than the caller creating them. If your callees are created prior to calling Launch Actor for your caller, you can store the queues through a direct method call on the caller actor. If callees are created independently, use a message to pass them to the caller actor. There are no fixed rules in an aggregation for how to handle Stop. Sometimes the aggregator is consider the owner for the callees and when it gets the Stop message, it passes it to the callees. Sometimes the aggregator is merely an interface that the callees talk through, and the caller can stop but the callees go on. Choose the approach that matches your application's needs.

Avoid Get and Set Messages

Avoid creating messages or message pairs that retrieve an attribute from an Actor, modify it, and return it to the Actor. These types of operations can introduce the same kind of resource contention issues as seen with Data Value References (DVR) or functional globals. Worse, they open the door to race conditions.

An actor runs a parallel task. If your Get/Set operation consists of two messages, then the Actor continues to run between your Get and Set messages. Another task could access and modify the same attribute concurrently causing a race condition.

If your Get/Set operation consists of a single (probably **Reply Msg**-type) message, then you have locked the Actor for the duration of your operation. If your caller attempts to access your Actor as part of your modify operation, your application will deadlock. It is appropriate, of course, for a caller to send a message requesting an update (or regular series of updates) from a nested actor. It is also appropriate for a caller to push an update to the nested actor. Issues only arise when you logically link the get and set operations; that is, when you get attributes for the purpose of modifying and returning them.

Benefits of the Actor Framework

The Actor Framework provides numerous tangible benefits. It is significantly more flexible and extensible than traditional state machine architectures, and is safer and more efficient than by-reference class hierarchies.

The pattern is similar to the functionality provided by a by-reference class because the design permits us to treat an Actor's Receive Queue as a handle to the Actor. Other by-reference designs rely on some container type, like data value references (DVRs) or single-element queues (SEQs) to create data that is shared between two independent VIs. While effective, this approach has the potential for race conditions and deadlocks, especially if there are multiple by-reference objects instantiated in the system. Furthermore, certain operations that are routine in languages with native by-reference support are cumbersome or even prohibited in the LabVIEW environment. The Actor Framework avoids many of these shortcomings by focusing on an asynchronous infrastructure instead of trying to replicate the by-reference patterns of other languages.

More Extensible and Reusable Code

The Actor Framework creates more potential for code extension and reuse than traditional state machines. A child Actor can reuse its parent's functionality in three ways:

1. To change a parent's behavior, use a child with an **override** method.
2. To add behavior to a parent, **extend** the parent by adding new methods to a child.
3. To give the parent's control logic additional control logic, **decorate**[2] the Actor Core.vi with an added control loop.

[2] "Decorate" is not an industry standard term in this context. We have coined it to describe this third aspect of reuse that is unique to task frameworks.

Messages are reusable within a class hierarchy, so a message that works on a parent still works on a child. In addition, messages themselves are inheritable, which can significantly simplify message senders. Finally, since messages are strongly typed, they reduce the risks associated with the untyped messages typically used in queue driven state machines.

Actors Objects Are Testable

You can test many actors without launching the Actor Core. Any of the member VIs of the actor class can be run directly. Those that do not attempt to send replies to other actors can be tested in isolation just like any other VI, without having to create the Self and Caller queues. You can verify the resulting state of an actor object by comparing it to another instance of a known value, just as you can check the contents of a LabVIEW cluster. This greatly facilitates automated testing.

If a method does communicate with another actor, you will need to initialize to initialize the Self and Caller queues. You can use a VI found on disk called:

    <vilib>\ActorFramework\Actor\ Init Actor Queues FOR TESTING ONLY.vi

This function provides a backdoor for setting the queues inside the actor, which are normally private, so you can mock up an actor and test whether, given a particular input message, it returns the expected result messages.

You can fit the Actor to a test harness or perhaps a TestStand sequence. The harness can invoke the Actor's various methods, and you can compare the results to expected values in the form of Actor instances of known states. If the Actor incorporates a user interface into its Actor Core, you can pass the reference to the harness into the Actor instead of the reference to the Actor Core, and verify results by checking the values of updated controls and indicators. Note this is harder to do if you have opted to employ user events for front panel updates, which argues against that method. If the user interface associated with the Actor is a separate VI, you can test the Actor and user interface in different harnesses.

The VI Tester from JKI is a system that is particularly good at providing testing for all object-oriented applications, including the Actor Framework. You can investigate that tool here: http://forums.jki.net/topic/985-vi-tester-home-page/

Guaranteed Delivery of Stop Message

A common problem in other messaging frameworks occurs during shutdown. One component sends the stop message to another component and then quits. Because a refnum in LabVIEW has the same lifetime as the VI that created it, many frameworks have a problem that the queue can be destroyed before the other component receives the message. Many go to elaborate extremes to ensure message delivery before the first component exits. With the Actor Framework, each component creates its own receive queue, so that queue has the same lifetime as the receiver, not the sender. That means that any component – caller or callee – can send the stop message and then shut itself down without worrying about the message not being delivered.

Questions?

The Actor Framework community forum: https://decibel.ni.com/content/groups/actor-framework-2011
Download expansions and variations of the framework here: https://decibel.ni.com/content/docs/DOC-17193