

CLA Summit 2010

LabVIEW Exceptions

Nate Moehring
3/8/2010

1

Error handling is a difficult subject in any language, not just LabVIEW. If NI is considering enhancing the error handling features in LabVIEW, why not take a look at what other languages provide in terms of error handling features and techniques and how they might translate into LabVIEW.

Improving LabVIEW Error Handling

- Many great ideas have already been shared
 - Replace Native Error Cluster (NEC) with OO
 - Wrap OO in NEC (w/ multiple error support)
 - Central Error Handling
 - Local Error Handling
 - Specific Error Handler
- All of these ideas focus on reading and writing error wires and terminals.

Nate Moehring
nate@themoehrings.com

2

It is right and appropriate for LabVIEW to focus on wires and terminals in the dataflow paradigm, but could there be an event-driven alternative or complement to error handling?

LabVIEW Exceptions (The Good)

- LabVIEW 7.x introduced Automatic Error Handling (AEH)
- If enabled in VI Properties, the compiler automatically adds a small amount of code to each unwired NEC terminal to call the General Error Handler (GEH) if an error occurred.
- This is a great feature to help developers catch exceptions during development and testing.

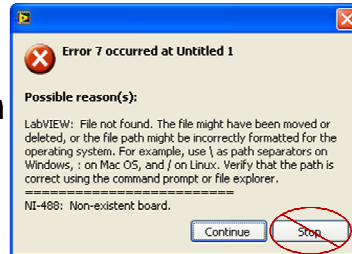
Nate Moehring
nate@themoehrings.com

3

It is a reasonable comparison to say that exceptions are like events, but for errors instead of data. It took several years for the Event Structure to become popular in LabVIEW, but now it has become the defacto method of responding to user activity on the front panel, rather than polling control terminals. Likewise, the current NEC system of error handling in LabVIEW is kind of like a polling architecture, polling error cluster outputs and leaving the user responsible for ensuring that the error gets from the source to an error handler. Exceptions, on the other hand, guarantee delivery of the error from the source to the destination without placing that responsibility on the user.

LabVIEW Exceptions (The Bad)

- However, this is typically not what you want in a deployed application.
- An unknowing user can bring your entire application to it's knees as a result of a harmless error.
- Yes, this feature can be disabled during deployment, but what if you still want to log these unhandled errors?



Nate Moehring
nate@themoehrings.com

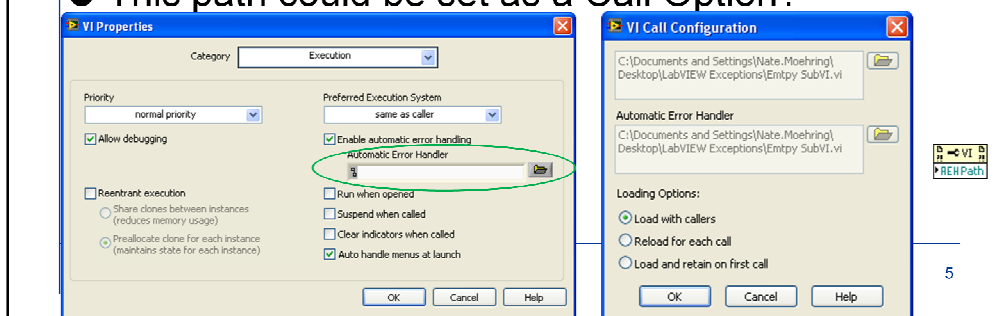
4

It would be very natural to desire that the default behavior of an unhandled exception in a deployed application to log silently to a debug file, but during development all unhandled exceptions should be brought to the testers' attention.

Configurable AEH

Wouldn't it be great if:

- VI Properties had a path control to specify what VI should be called instead of the GEH?
- This path could be changed programmatically via a property node?
- This path could be set as a Call Option?



If this is the only concrete thing that comes out of this presentation, I'd be a happy man.

Automatic Error Handler VI would probably be required to have a particular connector pane. This NI-defined connector pane should probably contain an LVException (dynamic dispatch?) input terminal, as will be described later.

LabVIEW Exceptions (The Missing)

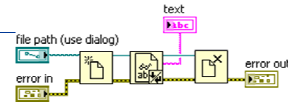
- Configurable AEH would be nice, but:
 - What if you want **different AEHs for different regions** of the block diagram?
 - What if you want to **run a different AEH based on the *type* of error** that occurred?
- Java, C++, .NET, and other popular languages have *try*, *catch*, and *throw* keywords to aid in error handling.
- Why not create a new TryCatch Structure to bring this powerful feature to LabVIEW?

Nate Moehring
nate@themoehrings.com

6

Exceptions Explained

- Consider a simple subVI:
- Internal to the subVI or primitives, many different errors could occur:
 - File not found
 - File is locked
 - File permission
 - Out of memory
 - Disk error
 - File didn't close



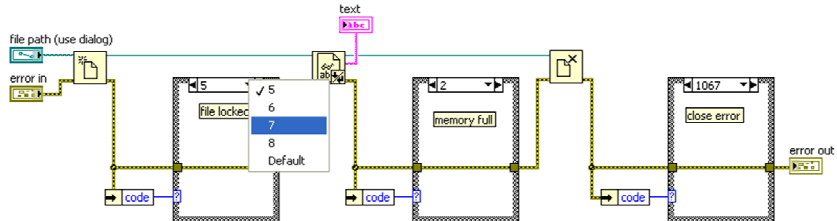
Nate Moehring
nate@themoehrings.com

7

In all examples, primitives represent any kind of node, primitives or subVIs.

Exceptions Explained

- If you wanted to handle each of these potential errors, you might draw something like this:

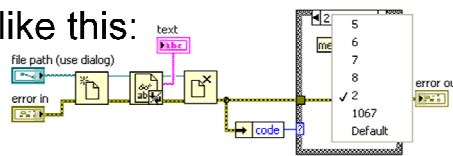


- The error handling code degrades the readability of the subVI.
- Errorcodes are hardcoded, difficult to read, limited in information, global, and outside of your control.

Handling the error after each node localizes the error response and is most representative of the behavior achieved with the use of exceptions.

More Typical Error Handling

- But Nate, I would write it like this:
- Yes, this is the way we've become accustomed to handling errors in LabVIEW, but is this the best way?
 - Where did the error actually occur?
 - How do you know what else executed after the error occurred?
 - Many subVIs and primitives do not have error clusters.
 - Some subVIs do not respect existing error conditions.
 - New error could mask original error.



- Previous error cannot disable execution of subsequent nodes if they do not have error inputs.
- Even if they have error inputs, no guarantee they have the “do nothing if error” case structure. I have found that removing the Error/No Error case structure from low level subVIs can greatly improve performance.
- Incoming errors can cause logic errors in the current subVI, causing new errors to be written to the error cluster that actually mask previous errors.

Exceptions Explained

- When an error is detected in a *try* block:
 - An exception object is instantiated and *thrown*.
 - Remainder of *try* block **does not execute**.
 - First compatible exception handler (*catch*) is executed.
 - *Finally* block is executed.
- Using exceptions:
 - Prevents subsequent code from executing.
 - Guarantees delivery of the error to an error handler.
 - Localizes the error and its response.
 - Preserves readability.

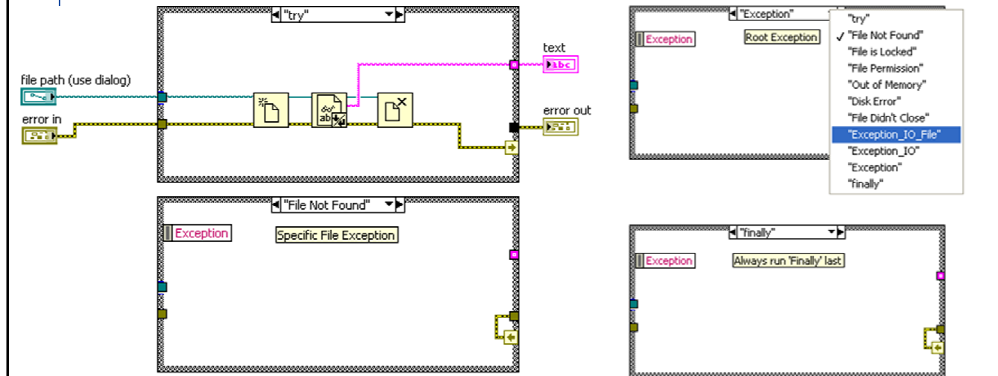
When relying on wires to carry error information, there is no guarantee that every error will reach an error handler. Some may be overwritten or dropped.

Joe Gerhardstein mentioned during his presentation that it was a long time before he even noticed the output error clusters on the In Place Element structure when dereferencing a Data Value Reference. In this case, those errors could have been missed, even though he normally would want to wire all error clusters. It was an unintentional omission that no longer guaranteed delivery of the error to an error handler (assuming AEH is disabled).

TryCatch Structure

If Error 7 occurred in File Open:

1. *Try* would stop executing at File Open.
2. File Not Found exception handler would **execute**. Event node would be populated with FileNotFound object, created by called primitive or subVI's throw statement.
3. *Finally* would **execute**.



Exception input is also available in the finally frame just in case the finally frame wants to use it. For example, one of the data members of the Exception_IO_File class could be a file reference. Thus, if the File Open passed but the File Read closed, the code within the primitive would be possible for populating the file refnum into the Exception object before throwing the exception. The finally frame could use this refnum to close the file properly, perhaps by calling a Close method or calling an accessor method to return the file ref.

TryCatch and Dataflow

- Execution **stops in try where exception occurred.**
 - Requires content of *try* to be executed in a single execution thread? Otherwise multiple exceptions may fire. Clumping still supported.
 - Unpopulated output tunnels in *try* would return default values.
- Subsequent frames can overwrite values populated in previous output tunnels.
 - Necessary for *catch* and *finally* frames to overwrite values populated by *try*.
- Support for Sequence Locals are necessary to pass data to subsequent frames, most likely rarely used.
 - No reason TryCatch couldn't also exist as a flat structure.

Nate Moehring
nate@themoehrings.com

12

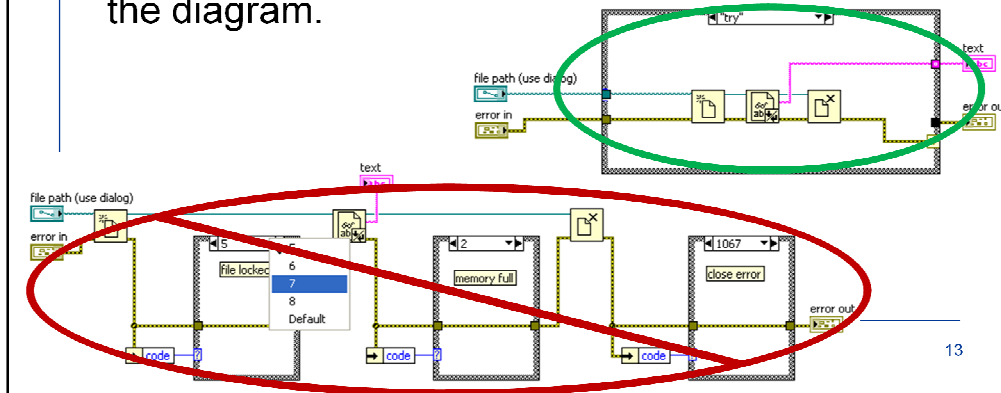
Because the contents of the try block represent an action that is intended to be atomic, meaning either the entire action passed or the entire action failed, it is most likely the actions in that try block would be serialized already. In other words, if actions are being done in parallel, how closely related are they? They probably shouldn't be in the same TryCatch structure. Plus, the serialization of clump execution would only happen within the try frame of the TryCatch structure. There's nothing to prevent you from putting TryCatch structures side by side to allow parallelism.

To implement a retry mechanism at a given failure point, the exception handler frame and finally frame would need to "roll back" the application to the state prior to the failed trail block. Then, outside the TryCatch structure, a while loop would retry the entire operation. This means any intermediate data such as the file reference opened by the Open File primitive would need to be stored either in the Exceptions object passed to the exception frame OR in a sequence local to make that data available to subsequent frame(s), in this case, closing the file in the exception frame to roll back application state.

Another possibility may be to use the Event Structure to catch Exceptions thrown by the throw exception primitive (just like a User Event) or internal to a LabVIEW primitive. However, this doesn't address the finally frame, and I think it muddies the purpose of the Event Structure.

Minor Benefit: Greater Readability

- In nominal case, reader only needs to read the *try* block to understand the intent of the subVI.
 - Finally block is primarily intended for resource deallocation.
- Error handling logic is still there but doesn't clutter the diagram.



13

Major Benefit: Enforcing Error Handling

- If the exceptions that a subVI can throw were specified in its VI Properties dialog, the **compiler** could **enforce** that all possible exceptions in the *try* frame **must have** associated **exception handlers in all callers**.
 - The compiler can help improve software quality by enforcing error handling!
 - To avoid excessive *catch* frames, callers can catch ancestor exception classes except when looking for specific error conditions.

Additional Benefits

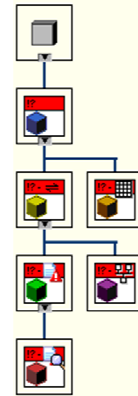
- Exception objects can contain more information and methods than the traditional NEC.
- Debug tools can be created to listen for runtime exceptions.
 - LabVIEW could automatically produce log files that capture application generated exceptions.

Example Exceptions

- Imagine needing to check an error cluster every time you:
 - Divide x by y (divide by 0)
 - Index an array (index out of range)
 - Build an array (out of memory)
- The error handling to perform these simple operations would be obnoxious.
- A TryCatch structure would allow you to respond to these types of errors without complicating your code or showing the AEH dialog to the user.

Creating Exceptions

- Recommend NI create a LVException class to act as the common ancestor for all exception classes to contain common required methods and data members such as:
 - Call chain
 - Status message
 - Time of error
 - Other data?
 - Show Dialog
 - Log Error
 - Other methods?
- Creating new exceptions is a simple matter of inheriting from LVException or any descendent exception class.



Nate Moehring
nate@themoehrings.com

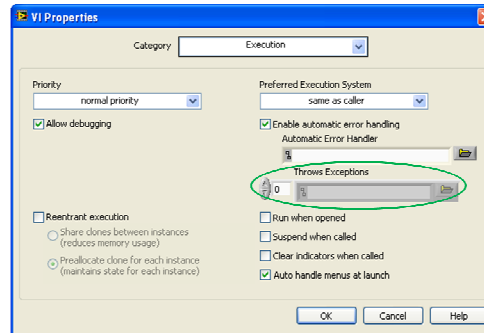
17

This example of exception hierarchy shows:

1. LV Root Object
 1. LVException
 1. LVException_IO
 1. LVException_IO_File
 1. LVException_IO_File_FileNotFoud
 2. LVException_Comm
 2. LVException_Memory

Declaring Exceptions

- All compiler-checked exceptions for the VI can be listed in the VI Properties.



- *Throws Exceptions* is an array of paths to Exception classes.

Nate Moehring
nate@themoehrings.com


18

Or array of object icons instead?

Declaring Exceptions

- What if no exceptions are declared in VI Properties?
 - subVI can throw any exception type (like C++)?
 - subVI must not throw any exceptions?
- Need concept of *checked* and *unchecked* exceptions like Java?

Throwing an Exception

- Code where error occurred is responsible for instantiating and initializing an Exception object.
- Exceptions thrown by subVIs would need a new LabVIEW primitive to pass the exception object to the Runtime Engine. 
 - Throw primitives could be located anywhere, not just in a *try* frame.
 - Runtime Engine is responsible for invoking the appropriate exception handler.
 - Would be great if LabVIEW could automatically populate the *Throws Exceptions* array based on the use of the *Throw* primitive within the VI (and ignored exceptions in subVIs).

Nate Moehring
nate@themoehrings.com

20

Exception Propagation

- Exceptions propagate up the call chain until a compatible *catch* is found or an AEH is invoked. AEH VI would require LVException input.
- Exception Handlers could choose to rethrow a caught exception, or simply mark the exception handler frame as *ignore*, to propagate the exception up the call chain.
 - Ignored exceptions should look disabled.
 - Marking a handler as ignored tells the compiler you've thought about this error condition, don't give compiler error.



Nate Moehring
nate@themoehrings.com

21

Even More Benefits

- Callers of a subVI can **catch errors the subVI may not have anticipated** without modifying the behavior of the subVI, avoiding adding risk to **unknown callers**. AEH, *Catch* frames, or perhaps a new Call Option to specify an AEH for a specific subVI invocation.
- **SubVIs can be updated to throw new exceptions** that can be caught by existing callers **without updating the callers** to look for new errorcodes.
 - Catch ancestor exception classes, dynamic dispatch to type-specific exception methods.

Nate Moehring
nate@themoehrings.com

22

I often use subVIs that have unwired error output terminals, particularly on primitives such as Format Into String, a frequently omitted primitive. I'd like my VI to be able to catch these exceptions (either through the AEH or TryCatch Structure) without modifying the subVI for fear of changing the behavior of existing callers.

Conclusion

- Exceptions add a new dimension to error handling in LabVIEW.
- Improve software quality by **enforcing** callers to implement a **minimum** level of **error handling**.
- **Respond** to errors **immediately**, not downstream.
- Use inheritance to **group errors** (and response) by **type of error**, not just *what* error occurred.
- Add **new error checking without updates** to callers/callees.
- Reduce the gap between LabVIEW and other popular high level programming languages.

If we want LabVIEW to gain popularity in the Software Engineering/Computer Science engineering/academic worlds as a general purpose programming language, LabVIEW needs to provide some of the tools commonly found in other languages. Remove all barriers possible for a hardcore textual programmer to consider adopting LabVIEW. By broadening the developer base, NI would also broaden it's sales potential.

References

- Jenkov, Jacob. "Introduction to Java Exception Handling". <http://tutorials.jenkov.com/java-exception-handling/index.html>
- Sun Microsystems. "Advantages of Exceptions". <http://java.sun.com/docs/books/tutorial/essential/exceptions/advantages.html>
- Wikipedia. "Exception Handling". http://en.wikipedia.org/wiki/Exception_handling