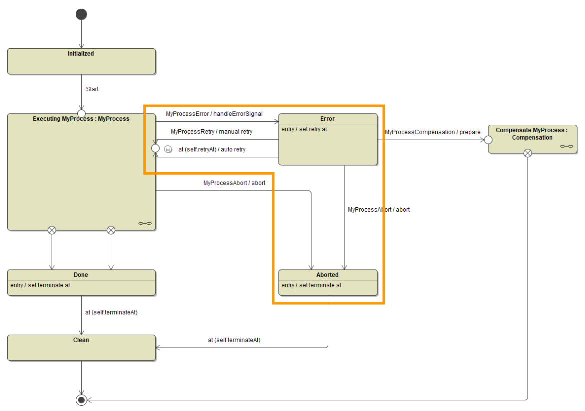
Error Handling of Root State Machine

The generated root state machine of Designer service comes with an integrated error handling.

All errors that occur on process execution are caught.



Executable processes work with the concept of **transactions**. Transactions are equivalent to units of work, that are committed at the end of the session. In case an error occurs during a transaction, the following happens:

1. The error is caught.
   When an exception occurs and is not explicitly caught in the execution diagram by the modeler, the current implementation (and persistent state transaction) skips to its end. All subsequent code is skipped. Then, the error is caught by the persistent state engine.
2. The error handler is invoked.
   The error handler sends an error signal (**handleErrorSignal**) to the root state machine.
3. The process switches into the error state at transaction end.
   The persistent state engine commits the transaction - and also the error signal. The process switches to the next process step but then goes to the error state.

The process instance is displayed as erroneous. You can now

- **Rectify the error and send a retry signal**
  You can now look into the error and apply fixes if necessary. The last process step the process has reached is marked by a history state, so it is possible to trigger a retry from this very step. You can either trigger a manual retry, or the process could have auto retry enabled and automatically performs retries in given intervals.
- **Abort the process instance**
  Alternatively, you can abort the process instance by sending an abort signal. In this case, the process instance will be terminated as aborted (not done).

Behavior of the State Machine

The behavior of the process is implemented to a sub state machine (**MyProcess** in the state machine diagram above). Depending on the BPMN Element you have added your execution implementation to, this process state machine shows a different behavior.

Error In	State Machine Element	BPMN Element	Behavior on Error
On Event	**Transition** from state **Initialized** to start event of sub state machine	<ul><li>Start Event</li><li>Message Start Event</li><li>Timer Start Event</li></ul>	1. The process state engine goes to next process step. 2. The process state engine goes to the error state. All subsequent code after the exception is skipped.

On this Page:

Related Pages:

- xUML Service State Machines

Related Documentation:

- Trigger a Retry
  - Persistent States of Containerized xUML Services
  - Stalled Persistent State Objects

	Transition from previous state to current state	• Plain Event • Message Event • Timer Event • End Event • Receive Task • User Task	
On Exit	**Entry** of Executed <name of service task>	Service Task	
Get Data		User Task	• Implementations in the **Get Data** execution do not have any effect on the persistent state engine (no error state). • On error, the form loads nevertheless but the form will not be initialized in any way.
Decision	**Guard expression on state transition** <name of the decision flow>	Exclusive Gateway	• The process instance does **not** go to error state. • The process is rolled back to the previous state. • A log message is logged to the service log: `Allowing guard functions to throw exceptions is considered a very bad practise, so, even though error handler will be executed, transaction will be rolled back. You probably should rethink your design.`

Behavior on Retry

In case of error, when the process is in error state, you can trigger a retry of the process execution. Refer to Persistent States of Containerized xUML Services (PAS Administration) or Stalled Persistent State Objects (Integration) for more details on how to do this.

When implementing you process, you should pay attention to the particularities listed below to allow for a smooth retry behavior.

Persisted Variables

A retry resumes the process starting with the behavior the error has occurred in. All persisted variables of the process are saved to the error state, and the retry will be performed with the saved values.

- Either set the persisted variables to valid starting values at the beginning of each behavior to avoid data corruption
- or keep track of the processing to be able to not start from the beginning but from the exact point where the error occurred.

Please also consider the hint regarding persisted variables at Order of Execution in Execution Diagram below.

Example

Processed data items are saved to a persisted array **anArray**. As the process switches into error state, all values within **anArray** are saved. On retry, the processing of data items is restarted from the beginning. If you do not initialize the array properly, you may end-up with duplicate array elements when data items are processed for the second time. In this case, it would be advised to initialize the array like e.g. `set anArray = NULL`.

Order of Execution in Execution Diagram

In this context, the order of execution in the execution diagram is important. The implementations in the execution diagram are executed in the following order:

1. **Persisted to Local**
   At first, all object flows from persisted variables to local variables are executed.
2. **Operation Calls**
   Next, all operation calls are performed in the defined order - including assigning the respective output values to local and persisted variables.
3. **Local to Persisted**
   Finally, all object flows from local to persisted variables are executed.

Regarding the retry hints concerning Persisted Variables above, please note that persisted variables are updated

- from operation calls directly after that call
- from local variables at the very end of the execution

Backend Processing

The same behavior as explained for persisted variables (see above) also applies to backend processing: If your backend does not support transaction handling, you should keep track of your processing to avoid duplicated processing in case of retry. This e.g. may concern file uploads or API calls.

For non-transactional backend processing, keep track of the processing to be able to retry the process from the exact point where the error occurred. You can save this data to persisted variables as these are saved to the error state and are available on retry.