# Exercise 8: Forward error recovery

As a starting point you can either use your code from Exercise 7.

Last time we looked at backward error recovery, where upon error all participants in a transaction reverted to a previous consistent state, and started over. In some situations this behaviour is unacceptable, because an error may happen many times in a row, which means there will be no upper bound on the number of retries required. Instead, we have to compensate for the error by finding a new (but probably sub-optimal) consistent state for the system. This is known as forward error recovery.

Regardless of whether we are using backward or forward error recovery, there may still be participants that are doing work that will never be used, since another participant has already signalled that the transaction should be aborted. The Ada language provides a mechanism for notifying tasks asynchronously, so they can be aborted immediately when a certain event triggers. This event can either be a delay, or the condition of a protected object entry becoming true. This mechanism is called "Asynchronous Transfer of Control", or ATC.

In our case we will use an entry in the transaction manager, where the condition becomes true when the transaction has been aborted by any of the participants.

## Desired functionality:

As before, we call Unreliable\_Slow\_Add. If this fails (raises an exception), all participants instead add 5, and commit this value (adding 5 never fails, because of reasons)[.](http://threewordphrase.com/pardonme.htm) When one participant fails, all participants that are not yet done should be aborted immediately.

### Select-Then-Abort

The structure for Asynchronous Transfer of Control is the "Select-then-abort" statement:

select

triggering\_alternative -- eg. X.Entry\_Call;

-- code that is run when the triggering\_alternative has triggered

-- (forward ER code goes here)

then abort

abortable\_part

-- code that is run when nothing has triggered

-- (main functionality)

end select;

The triggering alternative will be an entry call in the Manager, which we will call Wait\_Until\_Aborted. It has not been implemented yet (again, trust your future selves to do a competent job). After this trigger happens, compensate for the error by adding 5 (instead of 10), and committing that value instead.

Since we know that 1) the triggering alternative occurs simultaneously and asynchronously in all participating tasks, and 2) that we have an exit protocol that requires all participants to show up, we do not need to ask the manager if we should commit: We commit if all show up in the exit protocol (Manager.Finished;), or we do forward error recovery if the trigger triggers (As before, we use Signal\_Abort to notify the manager). Move or remove code to achieve this.

### Create the Wait\_Until\_Aborted entry

This is the trigger for the asynchronous select-statement. The manager has a boolean Aborted, that is set to true by the first participant to call Signal\_Abort. This entry will be run asynchronously the moment its condition is true (because of its placement in the select-then-abort statement), which means we can use the aforementioned boolean as the entry condition. Remember to reset this value when the last of the participants have passed through the entry.

### Modify the Finished entry

Since we do not need to ask the manager if we commit, we can get rid of the Should\_Commit boolean. And since we reset the Aborted boolean in the Wait\_Until\_Aborted entry, it does not need to be changed in the Finished entry. Remove the unnecessary code.

## Approval

* Show the student assistants that your program produce the expected results: All participants either add 10, or all participants compensate for failure by adding 5 instead. If an error occurs in any one of the participants, the recovery should happen immediately and asynchronously in all participants (make sure your delays in the Unreliable\_Slow\_Addfunction are set high enough that you can easily verify this).