Tests for the FAIR Datamaster

Martin Skorsky

Last change: 2024-10-02

Contents

1 Overview - What is tested 3

2 The Tests 6
2.1 dmPerformance 6
2.2 test_Cpu0Cpu1.py 6
2.3 test_RunningThreads.py 11
2.4 test_add.py 11
2.5 test_addDownloadCompare.py 11
2.6 test_altDestinations.py 11
2.7 test_async.py 13
2.8 test_basic.py 13
2.9 test_blink.py 14
2.10 test_boosterStartthread.py 15
2.11 test_bpcStart.py 24
2.12 test_coupling.py 24
2.13 test_dmCmd.py 26
2.14 test_dmCmdAbort.py 26
2.15 test_dmCmdAsyncclear.py 28
2.16 test_dmCmdClearcpudiag.py 28
2.17 test_dmCmdCursor.py 28
2.18 test_dmCmdDeadlinePreptimeStarttime.py 28
2.19 test_dmCmdForce.py 29
2.20 test_dmCmdHeap.py 29
2.21 test_dmCmdHex.py 29
2.22 test_dmCmdNoop.py 29
2.23 test_dmCmdOrigin.py 30
2.24 test_dmSched.py 30
2.25 test_dmTestbench.py 30
2.26 test_dmThreads.py 31
2.27 test_environment.py 31
2.28 test_fid.py 31

2.29 test_flow.py . . . 31
2.30 test_flowpattern.py . . . 32
2.31 test_flush.py . . . 33
2.32 test_loop.py . . . 34
2.33 test_lzma.py . . . 35
2.34 test_memory.py . . . 35
2.35 test_originStartthread.py . . . 36
2.36 test_originTwothreads.py . . . 36
2.37 test_overwrite.py . . . 39
2.38 test_overwriteQueue.py . . . 39
2.39 test_parallelBranch.py . . . 45
2.40 test_patternStartStop.py . . . 45
2.41 test_pps.py . . . 45
2.42 test_prioAndType.py . . . 48
2.43 test_priorityQueue.py . . . 59
2.44 test_referenceEdges.py . . . 59
2.45 test_remove.py . . . 61
2.46 test_runAllSingle.py . . . 64
2.47 test_runCpu0Single.py . . . 64
2.48 test_safe2remove.py . . . 70
2.49 test_schedules.py . . . 71
2.50 test_simultaneousThreads.py . . . 71
2.51 singleEdgeTest . . . 73
2.52 test_startStopAbort.py . . . 77
2.53 test_switch.py . . . 78
2.54 test_unilac.py . . . 78
2.55 test_waitloopFlush.py . . . 79
2.56 test_zzzFinish.py . . . 79

3 Common Components - The Testbench 80

3.1 dm_testbench.py . . . 80
3.2 Structure of Description . . . 87

Chapter 1

Overview - What is tested

The tests for the datamaster are written with Python and the pytest framework. This implies that tests can be started by name and also with a name pattern to select a group of tests. The tests use the datamaster tools `dm-cmd` and `dm-sched`.

The tests use the instance of the current build folder of the datamaster tools and `libcarpedm`.

All tests are on branch `dm-fallout-tests`. The tests run with `make` or `make all` in folder `modules/ftm/tests`. To compile `libcarpedm` use `make prepare`. This runs `make clean` and `make` in folder `modules/ftm/ftmx86`.

Important: The tests need exclusive access to the datamaster and the timing receiver. Otherwise the schedules and timing messages may be not the ones to test.

Examples:

`OPTIONS='--runslow' make`

Run all tests against the local datamaster, even those marked with –runslow. These test take longer than usual tests.

`OPTIONS='--runslow -k test_threadsStartStop' make`

Run tests with test_threadsStartStop in the file name, the class name or the test method name. In this case it is a file name.

These two markers are allowed to skip some of the tests.

1. `--runslow` Tests marked with this marker are long running tests which slow down the over all execution time. If you add this marker to the `OPTIONS`, these tests will run. Do not use it during test development.

2. `--development` Tests marked with this marker are not ready for automated testing. These are under development. If you add this marker to the `OPTIONS`, these tests will run.

Some test need both markers to run.

Table 1.1: Which test tests what (Part 1)
x means: this test uses the component, T means: this test tests this component. For some tests the result is not checked. These are considered as OK if no exception occurs.

Test	number of tests	Tools	libcarpedm	firmware	uses Python	checks result
dmPerformance		x	T	x	-	-
test_Cpu0Cpu1.py	4	T	T	T	x	x
test_RunningThreads.py	1	-	-	-	x	x
test_add.py	1	T	T	T	x	x
test_addDownloadCompare.py	238	T	T	T	x	x
test_altDestinations.py	20	T	T	T	x	x
test_async.py	1	x	T	x	x	x
test_basic.py	1	x	T	x	x	x
test_blink.py	1	T	T	T	x	x
test_boosterStartthread.py	8	x	T	T	x	x
test_bpcStart.py	1	x	T	x	x	x
test_coupling.py	1	x	T	x	x	x
test_dmCmd.py	31	T	x	x	x	x
test_dmCmdAbort.py	3	T	T	T	x	x
test_dmCmdAsyncclear.py	1	T	T	T	x	x
test_dmCmdClearcpudiag.py	1	T	T	T	x	x
test_dmCmdCursor.py	1	T	T	T	x	x
test_dmCmdDeadlinePreptimeStarttime.py	18	T	T	T	x	x
test_dmCmdForce.py	1	T	T	T	x	x
test_dmCmdHeap.py	3	T	T	T	x	x
test_dmCmdHex.py	1	T	T	T	x	x
test_dmCmdNoop.py	1	T	T	T	x	x
test_dmCmdOrigin.py	1	T	T	T	x	x
test_dmSched.py	8	T	x	x	x	x
test_dmTestbench.py	19	T	x	x	x	x
test_dmThreads.py	33	x	x	T	x	x
test_environment.py	3	-	-	-	x	x
test_fid.py	3	x	x	T	x	x
test_flow.py	5	x	T	x	x	x
test_flowpattern.py	4	x	T	x	x	x
test_flush.py	24	x	T	T	x	x

Table 1.2: Which test tests what (Part 2)
x means: this test uses the component, T means: this test tests this component. For some tests the result is not checked. These are considered as OK if no exception occurs.

Test	number of tests	Tools	libcarpedm	firmware	uses Python	checks result
test_loop.py	1	x	T	x	x	x
test_lzma.py	4	x	T	x	x	x
test_memory.py	15	T	T	T	x	x
test_originStartthread.py	4	x	T	x	x	x
test_originTwothreads.py	1	x	T	x	x	x
test_overwrite.py	3	x	T	x	x	x
test_overwriteQueue.py	1	x	T	x	x	x
test_parallelBranch.py	4	x	T	x	x	x
test_patternStartStop.py	1	x	T	x	x	x
test_pps.py	10	x	T	x	x	x
test_pps10Hz.py	3	x	T	x	x	x
test_prioAndType.py	2	x	T	x	x	x
test_priorityQueue.py	2	x	T	x	x	x
test_referenceEdges.py	4	x	T	x	x	x
test_remove.py	6	x	T	x	x	x
test_runAllSingle.py	4	x	T	x	x	x
test_runCpu0Single.py	1	x	T	x	x	x
test_safe2remove.py	19	x	T	T	x	x
test_schedules.py	6	x	x	T	x	-
test_simultaneousThreads.py	3	x	x	T	x	-
test_singleEdgeTest.py	1	-	T	-	-	x
test_startStopAbort.py	2	x	T	x	x	x
test_switch.py	1	x	T	x	x	x
test_unilac.py	3	x	x	T	x	x
test_waitloopFlush.py	1	x	x	T	x	x
test_zzzFinish.py	1	x	T	x	x	x

Chapter 2

The Tests

2.1 dmPerformance

`dmPerformance` tests the performance improvements in libcarpedm. The test starts a schedule on a clean datamaster, checks if some part of the schedule is removable, removes it and then adds another schedule. This is done for a small schedule and a larger schedule. The test is ok if all commands work. There is no check for this.

2.2 test_Cpu0Cpu1.py

`test_Cpu0Cpu1.py` tests three cases where an edge connects nodes on two CPUs.

In the test `testTwoCpusFlow`, a flow node on CPU 0 connects with a target edge and a flowdst edge to CPU 1. The test starts patterns, snoops the messages and checks that more than 35 messages were sent after one second. See Figure 2.1.

In the test `testTwoCpusFlush`, a target edge and a flushovr edge connect CPU 0 and CPU 1. The test starts patterns, snoops the messages and checks the messages after one second. See Figure 2.2.

In the test `testTwoCpusSwitch`, a switchdst edge and a target edge connect from CPU 0 to CPU 1. The test starts patterns, snoops the messages and checks the messages after one second. See Figure 2.3.

In the test `testTwoCpusOrigin`, a origindst edge connects from CPU 0 to CPU 1. The test tries to add the schedules, but this fails because such an edge is not allowed (neigbourhood validation forbids this). See Figure 2.4.

Only for some node types and edge types it is allowed that source and target node run on different CPUs.

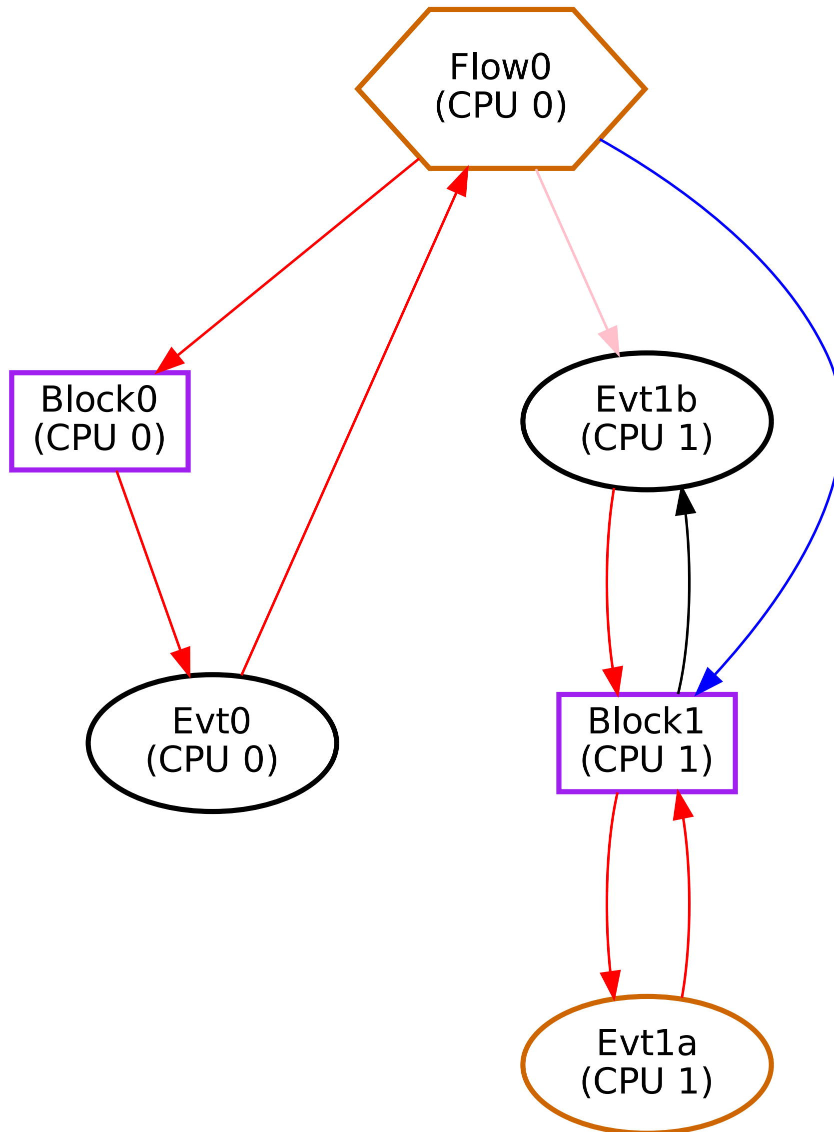


Figure 2.1: A flow node on CPU 0 connects with a target edge to a block node on CPU 1

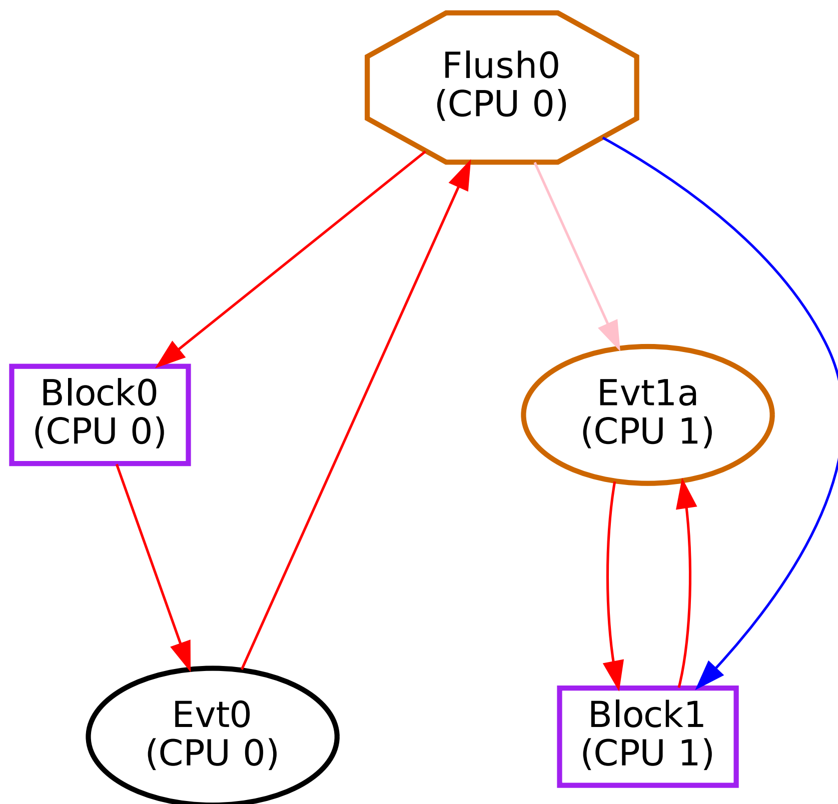


Figure 2.2: A flush node on CPU 0 connects with a flushovr edge to a block node on CPU 1

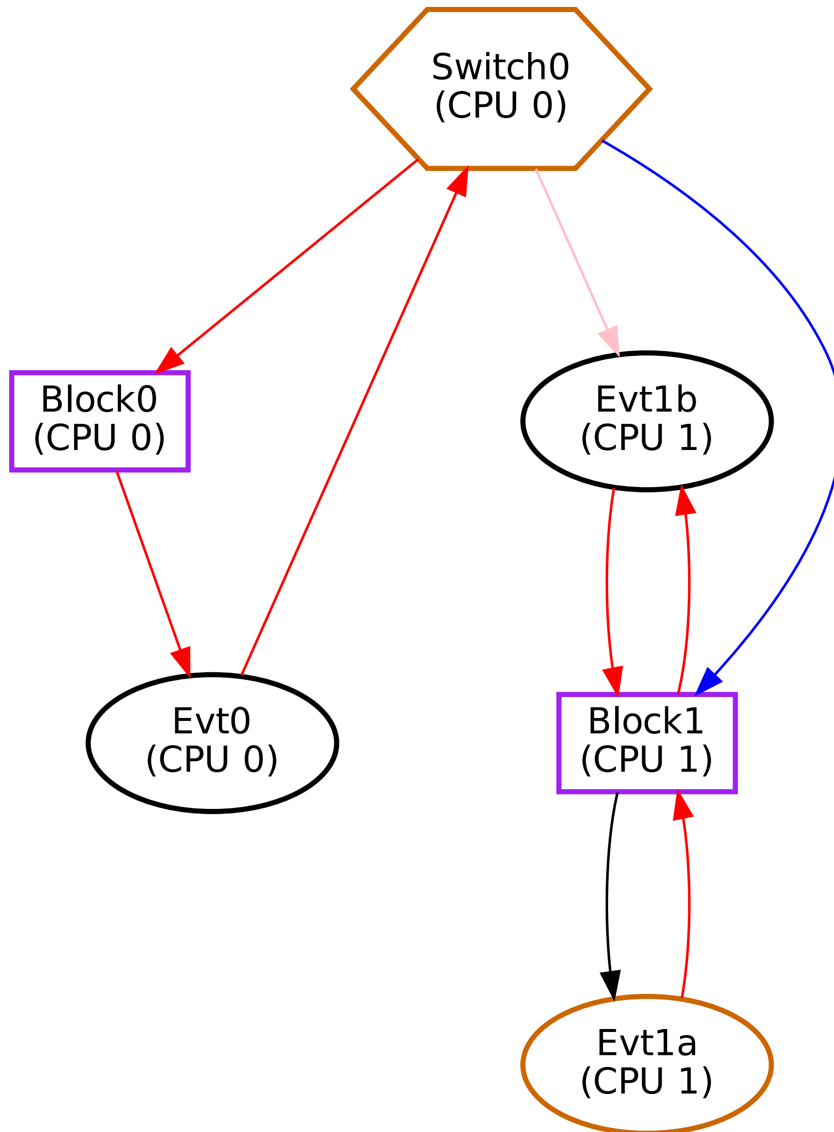


Figure 2.3: A switch node on CPU 0 connects with a target edge to a block node on CPU 1

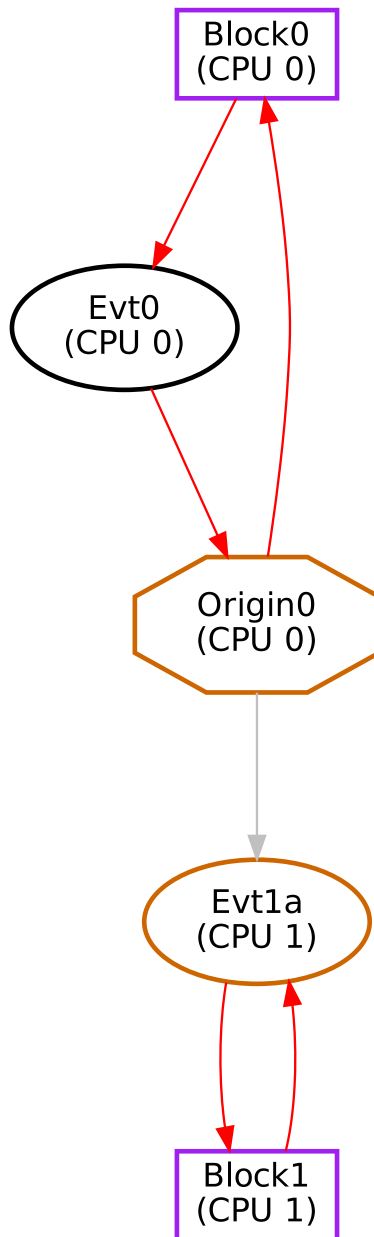


Figure 2.4: A origin node on CPU 0 connects with an origindst edge to an event node on CPU 1

2.3 test_RunningThreads.py

`test_RunningThreads.py` is a test for development. The aim is to improve `prepareRunThreads`. There were sporadic failures in the check that no thread is running. The command `dm-cmd <datamaster> running` shows running threads, which is unexpected. The test runs `prepareRunThreads` for 1000 times. The test adds some error handling and statistics in `tearDown`. It is marked with @pytest.mark.development, since this test is not needed in test runs.

2.4 test_add.py

`test_add.py` contains one test that adds a schedule, adds a second schedule. Then download the resulting schedule and compare this to an expected dot file. See Figure 2.5.

2.5 test_addDownloadCompare.py

`test_addDownloadCompare.py` test that a schedule is equivalent to the schedule which is downloaded form the datamaster firmware. The test steps are: clear the datamaster, add a schedule, start all pattern, download the schedule, compare both schedules with `scheduleCompare`. `scheduleCompare` should find no difference between the original and the downloaded schedule. Each test case uses a different schedule.

This test requires `scheduleCompare` to be installed. This tool checks that two dot-files represent the same schedule. The tool is build with make in folder modules/ftm/analysis/scheduleCompare/main/. It is installed with sudo make install in the same folder.

2.6 test_altDestinations.py

`test_altDestinations` tests for 9 edges and 10 edges of type `altdst` for a block node. The schedule `altdst-flow-9.dot` has one block, nine flow commands and nine messages. each flow command changes flow to another message. The test adds this schedule and starts the pattern in this schedule. The test checks that messages are produced.

The schedule `altdst-flow-10.dot` is similar to the one obove, but with 10 flow nodes and 10 messages. Adding this schedule is OK. The test checks for the correct response.

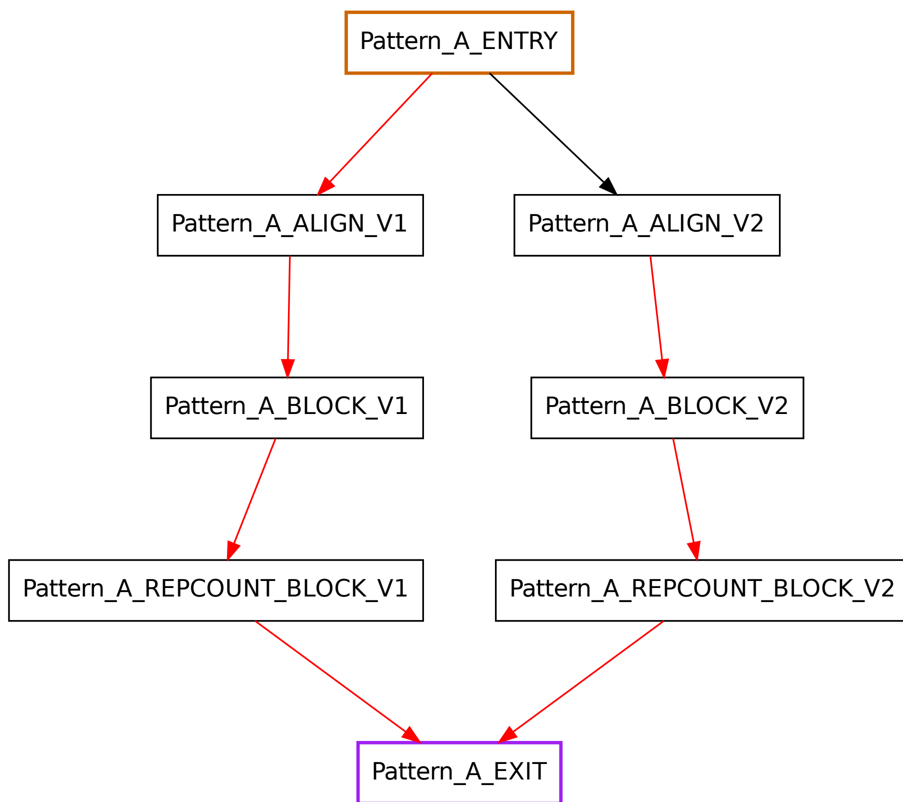


Figure 2.5: The resulting schedule which is added in two steps.

The schedule `altdst-9.dot` has one block and nine messages. The test switches through all messages and checks with snoop that the correct messages are send.

The schedule `altdst-10.dot` has one block and 10 messages. This schedule can be added to the datamaster. The test checks for the correct response.

The schedule `test-altdst-missing-node.dot` has one block with 9 altdst edges and one defdst edge. The test checks that a switch command via CMD-schedule works.

2.7 test_async.py

This test was `full_test/dynamic/async`.

1. Purpose of Test

   For a schedule asynchronous clear a block, change the destination to a timing message and check that all nodes are visited.

   See Figure 2.6 for the test pattern.

2. Test Actions

   Upload test schedule and start pattern *LOOP*. Check with `dm-cmd rawvisited`. The nodes 'BLOCK_B', 'BLOCK_LOOP', 'CMD_LOOP' are visited. Lock pattern 'B' (this locks 'BLOCK_B'), check this with `showlocks`. Clear pattern 'B' which clears the queues of 'BLOCK_B', check this with `rawqueue`. Change schedule with flow command. Destination of 'BLOCK_B' is now 'MSG_A' for one message. Check this with `rawqueue`. Unlock pattern 'B', wait for 1.2 seconds and then check with `rawvisited` that all nodes are visited and no blocks are locked (with `showlocks`).

3. Success Criteria

   Check that node 'MSG_A' is visited after changing the destination of the flow.

2.8 test_basic.py

This test was `full_test/static/basic`.

1. Purpose of Test

   This test uses `dm-sched add` and `dm-sched remove`. The pattern Figure 2.7 is loaded into datamaster and removed afterwards.

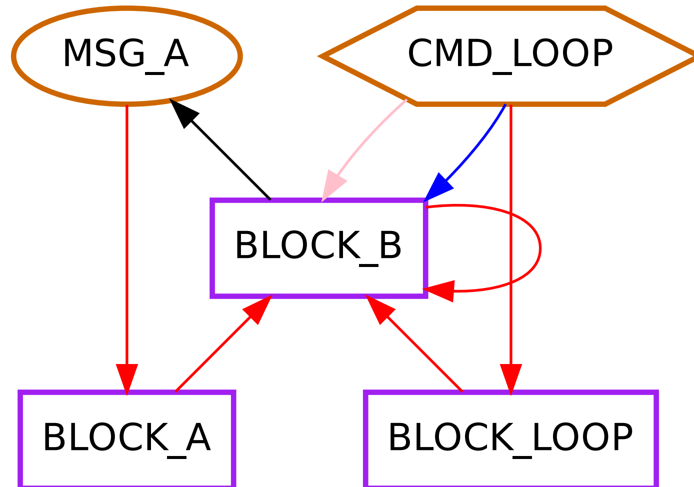


Figure 2.6: Pattern for the dynamic async test

2. Test Actions

   On a cleared datamaster the test pattern is added with `dm-sched add`. With `dm-sched status` it is checked that 24 nodes with the expected names are available. The test pattern is removed with `dm-sched remove`. At the end `dem-sched status` is used to check that no pattern is present on the datamaster.

3. Success Criteria

   The test is successful if no schedule is loaded. Checked with `dm-sched status`

2.9 test_blink.py

1. Purpose of Test

   Test flow nodes in connection with origin, preptime and two threads.

2. Test Actions

   Start pattern `ping` for a single timing message which indicates the start of the test. Set origin, preptime, starttime for threads 0, 1 of CPU 0.

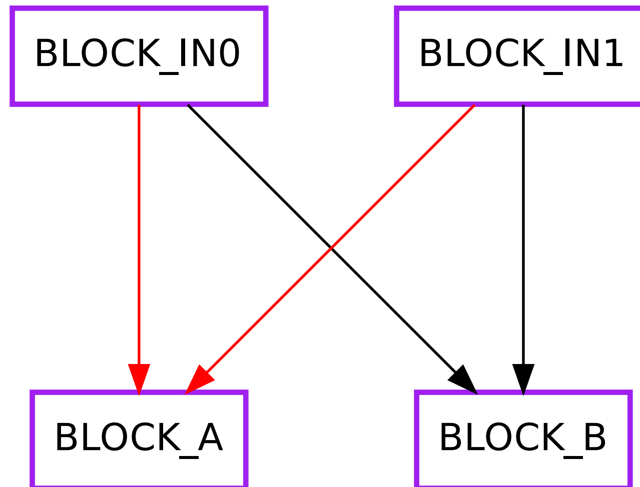


Figure 2.7: Pattern for the static basic test

Read these values. Start thread 0, 1 of CPU 0 with a single command. Snoop timing messages for 10 seconds and check the frequency of the messages.

3. Success Criteria

   The test is successful when snoop of timing messages receives all four types of messages, each marked by the parameter.

2.10 test_boosterStartthread.py

`test_boosterStartthread.py` tests schedules with nodes of type `origin` and type `startthread`.

1. `test_threeThreads0` Use schedule `booster_startthread.dot` (see 2.9), which has three pattern (MAIN, A, B). The schedule runs on CPU 1, thread 0. It starts threads 1, 2, and 3. Thread 1 and 2 trigger a message with parameter 2, while thread 3 triggers a message with paramter 3. The test result is checked with a snoop for 1 second. The correct timeline of the messages is not checked.

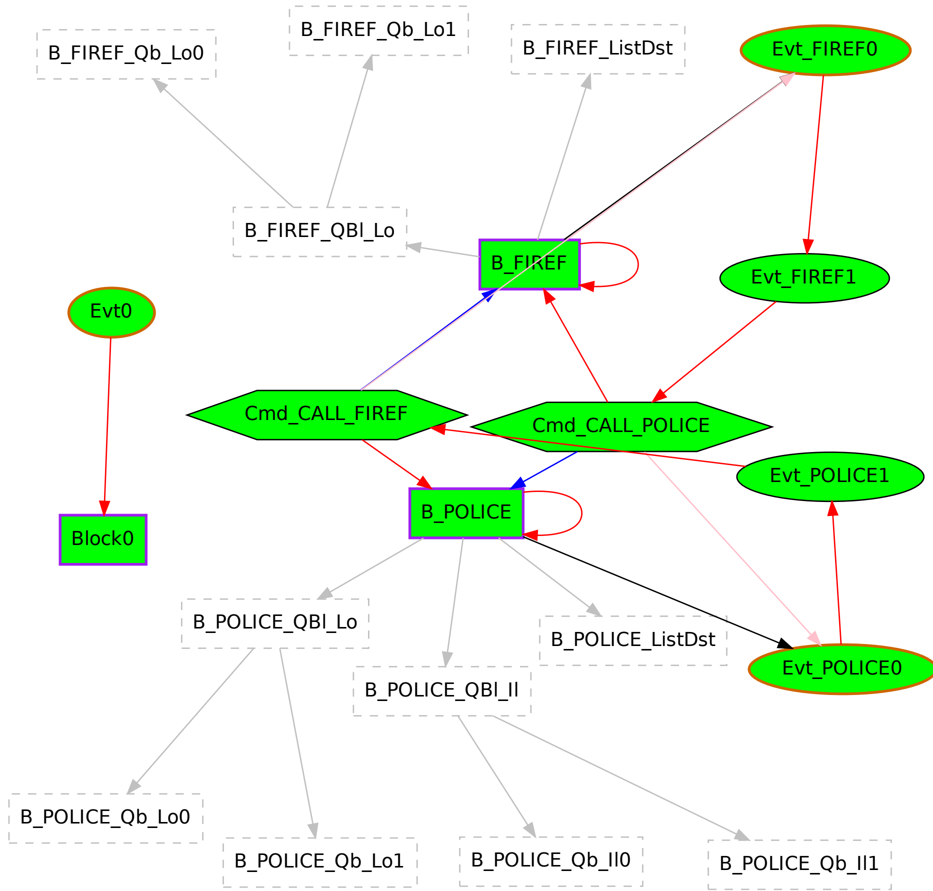


Figure 2.8: Flip between two sequences of timing events

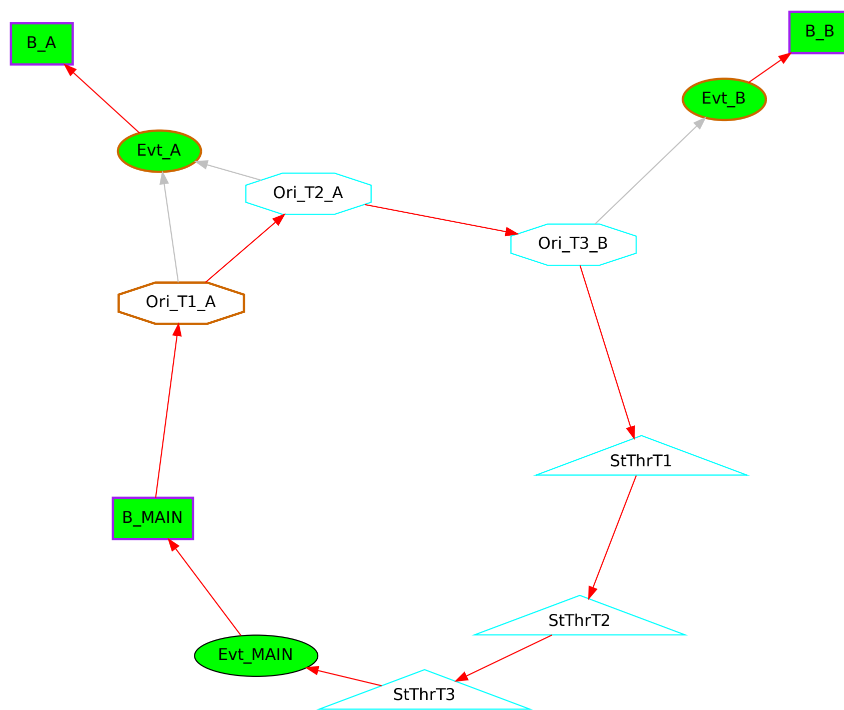


Figure 2.9: Schedule for test of origin and startthread with three threads, originated by three origin nodes

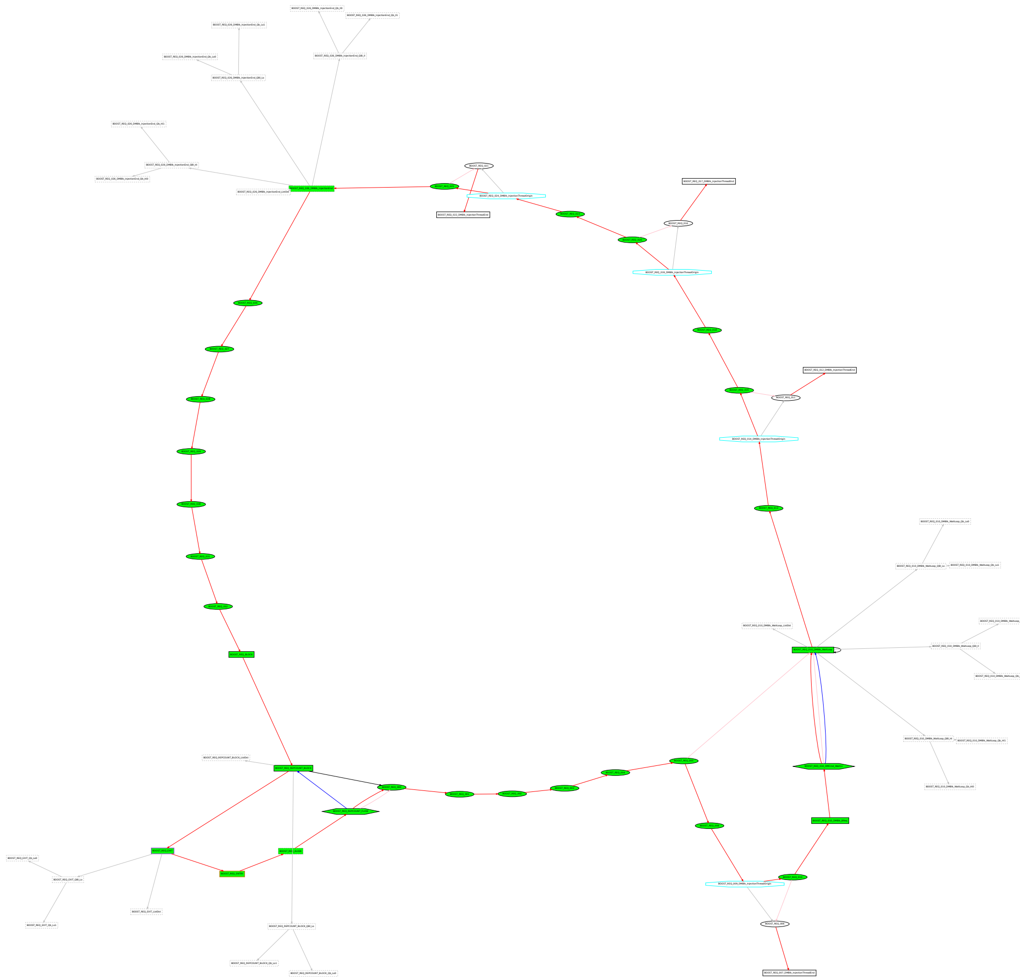


Figure 2.10: Schedule for test of origin with thread 1

2. `test_threeThreads1` Use schedule `booster_startthread-1.dot` (see 2.10) to test origin nodes. The test starts pattern BOOST_REQ and snoops timing messages for two seconds. The result is checked for the event numbers 0x0100 (more than 7), 0x0200 (at least one), 0x0102 (exactly one), 0x0103 (exactly one), 0x0160 (at least one).

3. `test_threeThreads2` Use schedule `booster_startthread-2.dot` (see 2.11) to test origin nodes. The test starts pattern BOOST_REQ and snoops timing messages for three seconds. The result is checked for the event numbers 0x0100 (more than 7), 0x0200 (at least one), 0x0102 (exactly one), 0x0103 (exactly one), 0x0160 (at least one).

4. `test_threeThreads3` Use schedule `booster_startthread-3.dot`

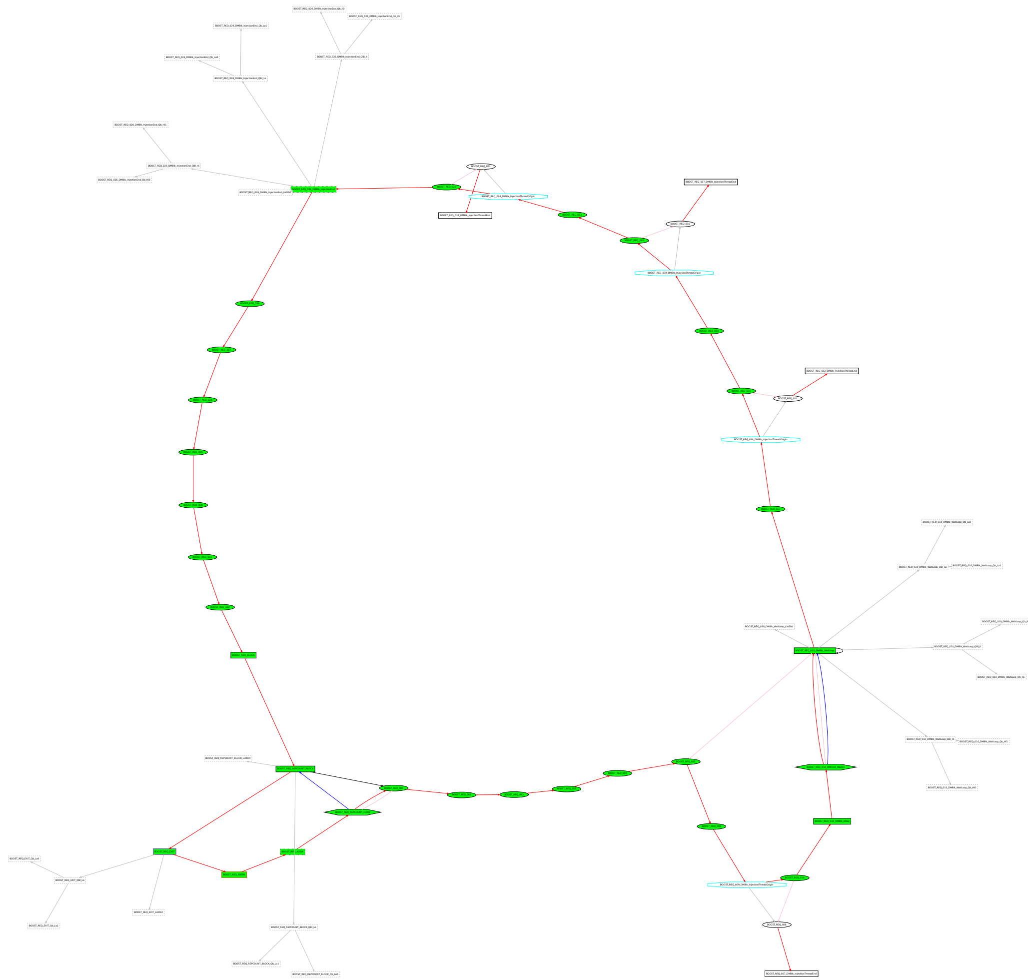


Figure 2.11: Schedule for test of origin with thread 1

(see 2.12), to test origin and startthread nodes. The test starts pattern MAIN and snoops timing messages for three seconds. The result is checked for the event numbers 0x0001 (at least one), 0x0002 (at least one), and 0x0003 (at least one). Event number 0x0001 indicates thread 1, event number 0x0002 indicates thread 2, Event number 0x0003 indicates thread 3. With saft-ctl snoop there are 4 timing messages each second after starting pattern MAIN. First a message with evtno=0x0003, then two messages with evtno=0x0002 1µs later, one of these is delayed. 20µs later we get a message with evtno=0x0001. tperiod=1s in block B_MAIN defines the tact of one second. tperiod of the blocks B_A and B_B are not relevant. The delay of evtno=0x0002 with 1µs is defined by toffs=1000. The delay of evtno=0x0001 is defined by toffs=20000. What is the effect of toffs and startoffs defined in the three startthread nodes?

5. `test_booster_all_threads` Use schedule `booster-all-threads.dot` (see 2.13), which has one pattern MAIN. The pattern runs on CPU 0, thread 0 and starts threads 1 to 7 on CPU 0. MAIN has a timing message and a block in a loop. One timing message every 0.1 sec.

6. `test_booster_thread_0_loop` Use schedule `booster-thread-0-loop.dot` (see 2.14), which has one pattern MAIN and starts thread 0 in a loop. This is a theoretical setup to test that starting a running thread again does not crash.

7. `test_booster_thread_0` The schedule consists of three nodes: a startthread for thread 0 (CPU 0), a timing message with EVTNO 1, and a block with length 0.1 seconds. The thread 0 starts every 0.001 seconds again, thus producing timing messages with 1kHz. The test asserts that there are more than 2990 tmsg in 3 seconds with EVTNO 0x0001. Use schedule `booster-thread-0.dot` (see 2.15), which has one pattern MAIN.

8. `test_booster_8_loops` The schedule consists mainly in a loop with origin nodes and startthread nodes for threads 1 to 7. This loop runs in thread 0 with a block of length 0.1 seconds and a timing message with EVTNO 0, which helps to control that the test is working. Each of the origin nodes is the origin of a sequence of a block, a timing message, and a block (no loop). The startthread nodes start threads 1 to 7, repectively. Since the EVTNO of the timing messages correspond to the thread numbers, the test can check that the thread is started. Since there is no loop for thread 1 to 7 each of these threads runs to

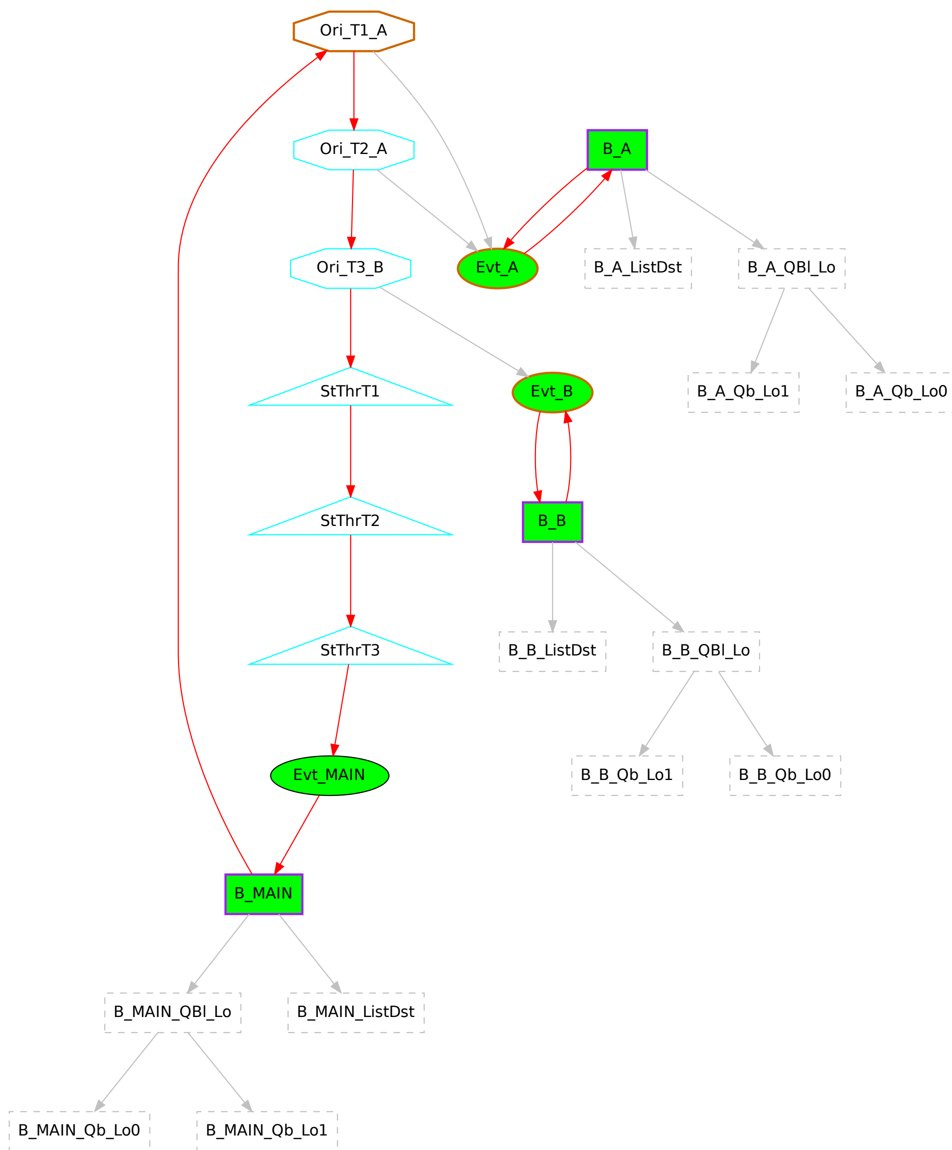


Figure 2.12: Schedule for test of origin and startthread

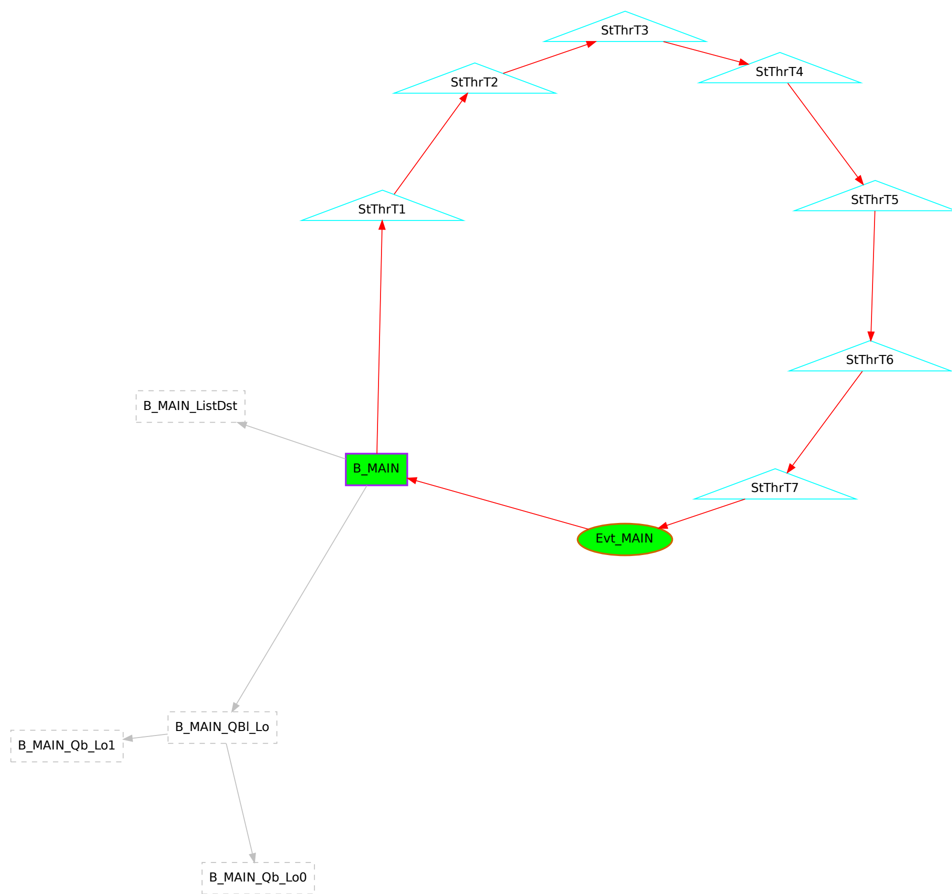


Figure 2.13: Schedule for test of origin and startthread with all 8 threads of CPU 0

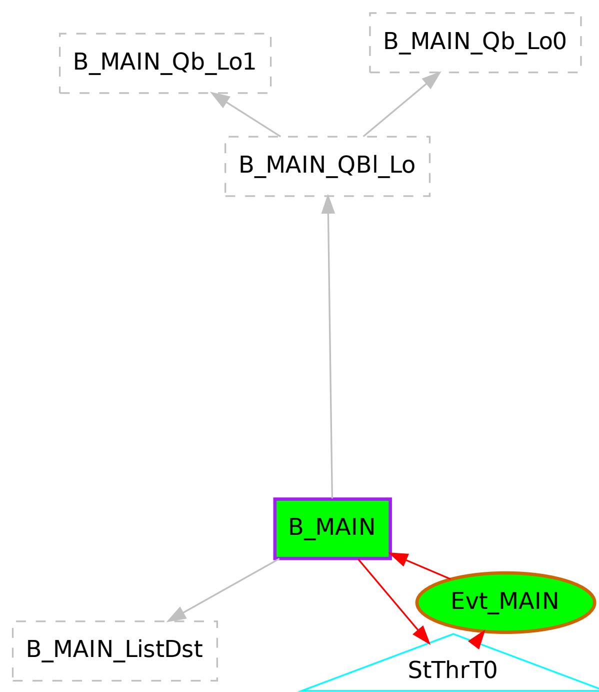


Figure 2.14: Schedule for startthread in a loop

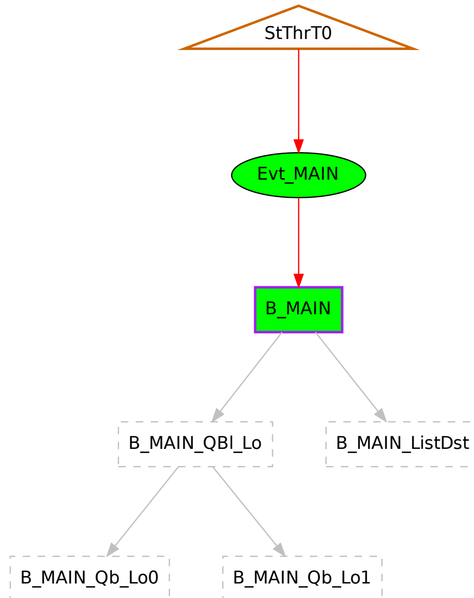


Figure 2.15: Schedule for starting thread 0 once every 0.001 seconds

idle. The threads are started once every 0.1 second from the main loop in thread 0. Thus we have a frequency of 10Hz for the messages of each thread. This is checked with a snoop for 2 seconds. Use schedule `booster-8-loops.dot` (see 2.16).

2.11 test_bpcStart.py

`test_bpcStart.py` tests the implementation of the beam process chain start flag in libcarpedm. The test schedule sends two timing messages with `bpcstart=True` and `bpcstart=1`. With `saft-ctl snoop` it is checked that the timing messages contain the correct setting. In addition with `dm-sched` the dumped schedule is checked for the bpcstart flag.

2.12 test_coupling.py

This test was `full_test/dynamic/coupling`.

1. Purpose of Test

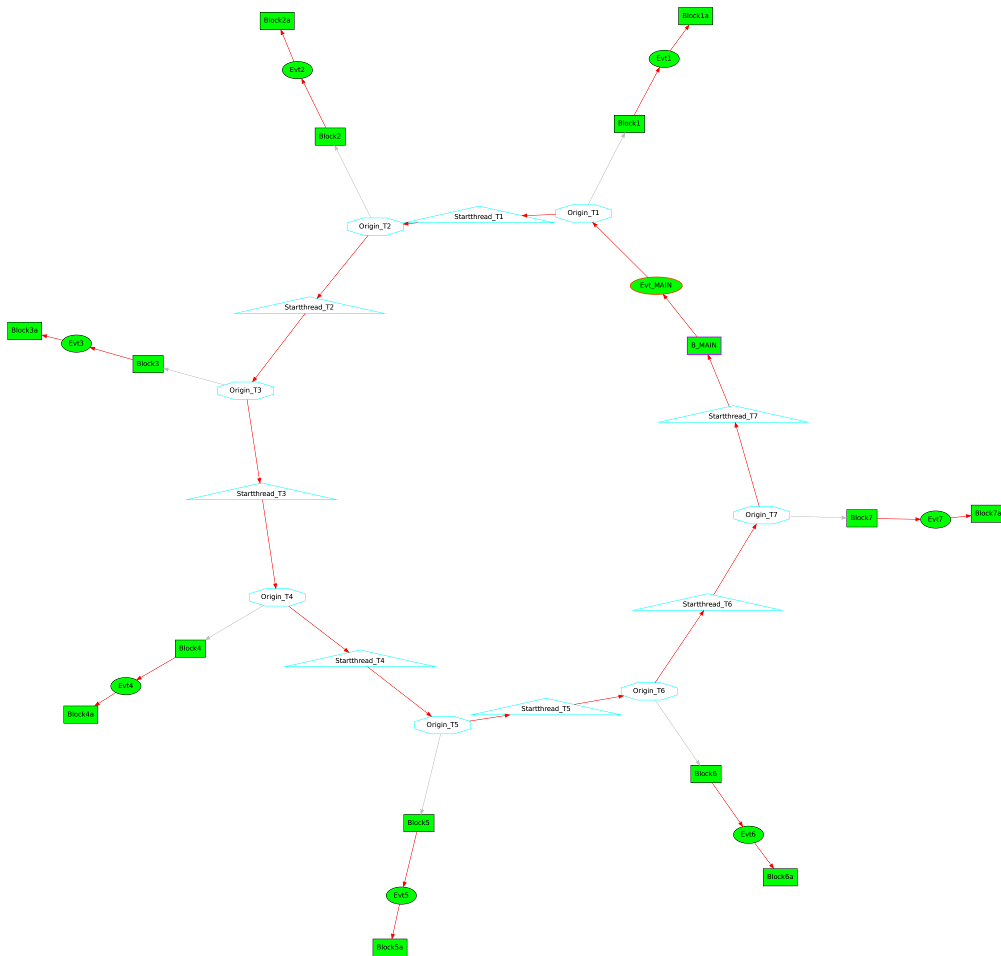


Figure 2.16: Schedule for starting threads 0 to 7 in a main loop

This test enlarges an existing pattern with a second pattern with edges into the first pattern. See Figure 2.17 for the test patterns.

2. Test Actions

   First, a pattern with three nodes is added. In a second step a pattern with additional three nodes is added. This pattern contains edges into the first pattern.

3. Success Criteria

   After adding the two patterns the status is checked with `dm-sched status`. The resulting `download.dot` is compared to an expected dot-file.

2.13 test_dmCmd.py

`test_dmCmd.py` contains Python unit tests for the tool dm-cmd. Each unit test calls dm-cmd with commands and options and checks the result with the output on stdout and stderr. There are also negative tests with an invalid command line. These tests are successful when the response is the correct error message and not a core dump.

There are five tests for the stop command. The test cases are

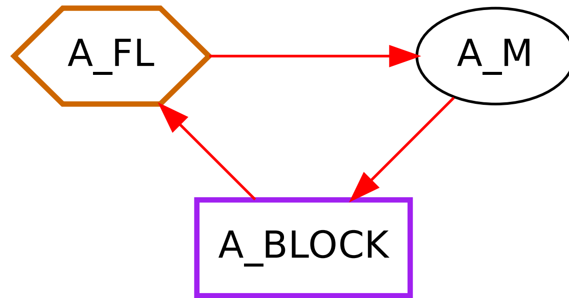
1. Test with existing block with low prio queue: result ok.
2. Test with existing block without low prio queue: result fail.
3. Test with existing timing event: result fail.
4. Test with non-existing block: result fail.
5. Test with no target name: result fail.

There are tests for `dm-cmd reset` and for `dm-cmd reset all`. These two tests check for return code 0, but not the status of the datamaster.

2.14 test_dmCmdAbort.py

`test_dmCmdAbort.py` tests `dm-cmd <datamaster> abort` for different CPUs and threads. Mainly tests the options `-c` and `-t`.

1. testAbortSingleThreadDecimal tests this for each CPU and each thread, given as decimal numbers.



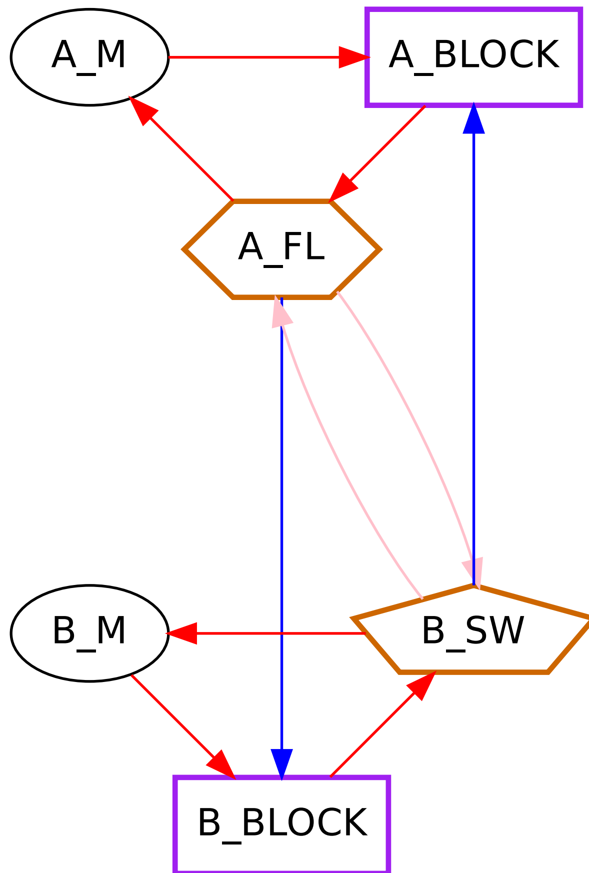


Figure 2.17: Pattern for the static coupling test before and after coupling

2. testAbortSingleThreadHex tests this for each CPU and each thread, given as hexadecimal numbers.

3. testAbortRunningThreads runs schedules on all threads and `abort`s for CPU 0 and 1 the threads 1, 3, 5, 7.

2.15 test_dmCmdAsyncclear.py

`test_dmCmdAsyncclear.py` tests `dm-cmd <datamaster> asyncclear` for different CPUs and threads. Send the command `noop` to Block0b. Since the schedules send a message every second, this command is in the queue for a second. The `asyncclear` clears this command in the queue. This is checked with `dm-cmd <datamaster> queue Block0b`. Before `asyncclear` the command is in the queue, afterwards the queue is empty.

2.16 test_dmCmdClearcpudiag.py

`test_dmCmdClearcpudiag.py` tests the command `dm-cmd <datamaster> clearcpudiag` for different CPUs and threads. Run `clearCPUdiag` for CPU 0 and 1 and for the threads 1, 3, 5, 7. Check the result with `dm-cmd <datamaster> diag`.

2.17 test_dmCmdCursor.py

`test_dmCmdCursor.py` tests `dm-cmd <datamaster> cursor` for two CPUs and four threads. Run `dm-cmd <datamaster> cursor` for CPU 0 and 1 and for the threads 1, 3, 5, 7. Check for the correct number of output lines.

2.18 test_dmCmdDeadlinePreptimeStarttime.py

`test_dmCmdDeadlinePreptimeStarttime.py` tests the five commands
`dm-cmd <datamaster> origin`,
`dm-cmd <datamaster> cursor`,
`dm-cmd <datamaster> deadline`,
`dm-cmd <datamaster> preptime`, and
`dm-cmd <datamaster> startline`
for different CPUs and threads. Tests the commands `origin, cursor, starttime, preptime`, and `deadline` for different combinations of threads. With `preptime`, all combinations for the threadbits between 0 and 255 ist

tested. This is every subset of a set of 8 threads. In addition, invalid thread masks and thread numbers out of range are tested.

2.19 test_dmCmdForce.py

`test_dmCmdForce.py` tests the command `dm-cmd <datamaster> force` for different CPUs and threads. Run `force` for CPU 0 and 1 and for the threads 1, 3, 5, 7. Check for the correct number of output lines.

2.20 test_dmCmdHeap.py

`test_dmCmdHeap.py` tests `dm-cmd <datamaster> heap` for different CPUs and threads.

1. testHeapSingleThreadDecimal tests this for each CPU and each thread, given as decimal numbers.
2. testHeapSingleThreadHex tests this for each CPU and each thread, given as hexadecimal numbers.
3. testInspectHeap run `heap` for CPU 0 and 1 and for the threads 1, 3, 5, 7. Check for the correct number of output lines. Run `heap` for each thread on each CPU.

2.21 test_dmCmdHex.py

`test_dmCmdHex.py` tests `dm-cmd <datamaster> hex` for the nodes `Block0a` and `Block0a_ListDst_0`. This is done on a pps pattern on CPU0. The output is different for 8 and 32 threads. The output is checked.

2.22 test_dmCmdNoop.py

`test_dmCmdNoop.py` tests `dm-cmd <datamaster> noop` for different CPUs and threads. Check with `dm-cmd <datamaster> queue Block0b` that the queue of Block0b is empty. Send the command `noop` to Block0b. Check with `dm-cmd <datamaster> queue Block0b` that the command `noop` is in the queue of Block0b.

2.23 test_dmCmdOrigin.py

`test_dmCmdOrigin.py` tests `dm-cmd <datamaster> origin` for two CPUs and four threads. Run `origin` for CPU 0 and 1 and for the threads 1, 3, 5, 7. Check for the correct number of output lines.

2.24 test_dmSched.py

`test_dmSched.py` contains 8 unit tests for the tool dm-sched. Each unit test calls dm-sched with commands and options and checks the result with the output on stdout and stderr.

1. `dm-sched <datamaster> -h`: check for correct usage message.
2. `dm-sched <datamaster>`: check for default operation (same as status).
3. `dm-sched <datamaster> status`: check for printig the datamaster status.
4. `dm-sched <datamaster> dump`: check that the current schedule is dumped to file download.dot (the default file name).
5. `dm-sched <datamaster> dump -o dump.dot`: check that the current schedule is dumped to file dump.dot.
6. `dm-sched <datamaster> clear`: check the clear command.
7. `dm-sched <datamaster> rawvisited`: check the rawvisited command.
8. `dm-sched <datamaster> add pps.dot`: check for adding a schedule.

2.25 test_dmTestbench.py

1. Purpose of Test

   Tests the method analyseFrequencyFromCsv which is in the module dm_testbench.py.

2. Test Actions

   Run 8 tests for the different check syntax of this method with the file `other/snoop_test_analyseFrequencyFromCsv.csv`.

3. Success Criteria

   Use the assertions in method analyseFrequencyFromCsv.

2.26 test_dmThreads.py

`test_dmThreads.py` tests the firmware with up to 8 threads and up to 4 CPUs. For each thread a pattern with one block and one timing message per second is loaded and started. To check that the thread is running, `dm-cmd` is used two times with a delay of one second. The number of messages is extracted from stdout and this number must increase for test success.

There are 8 tests with 1 to 8 threads on CPU 0. In addition, one test runs the schedules one 4 CPUs, each with 8 threads. This produces 32 messages per second.

2.27 test_environment.py

`test_environment.py` checks the test environment. The test is ok if and only if dm-cmd and dm-sched are from folder ../bin and libcarpedm is loaded from ../lib. This is checked with `ldd`.

2.28 test_fid.py

`test_fid.py` tests timing messages with different values of fid.

1. `test_fid7` tests the fix for the format id 7 bug. This bug was: under some conditions an illegal format id 7 was in the message. The test checks that ony format id 1 is in the snooped messages.

2. `test_fid0` tests with fid 0, which results in 19 fields in the timing messages.

3. `test_fid1` tests the standard format of the timing messages.

2.29 test_flow.py

This test is based on `full_test/dynamic/branch/single`, using flow instead of flowcommand.

1. Purpose of Test

   Test that the flow command switches from one block to another.

   See Figure 2.18 for the test pattern.

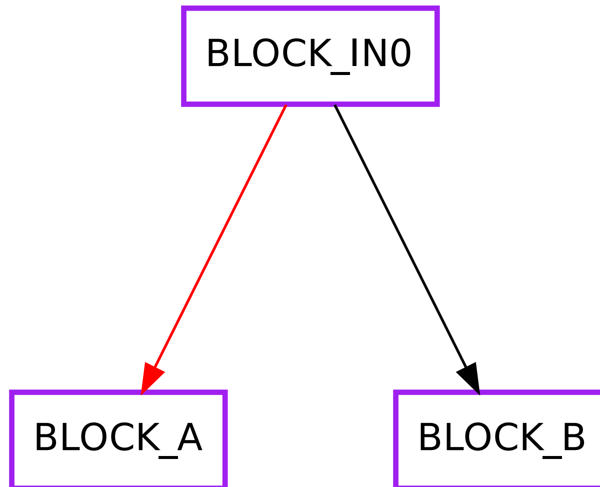


Figure 2.18: Pattern for the dynamic branch single test

2. Test Actions

   Add a schedule, start the pattern 'IN_C0'. After checking that nodes 'BLOCK_IN0' and 'BLOCK_A' are visited, change the flow with the flow command at pattern 'IN_C0' from 'A' to 'B'. Check that the flow command is in the low priority queue and then start the pattern 'IN_C0'. Check that the flow command is processed in the low priority queue and node 'BLOCK_B' is visited.

   Test the four combinations of relative VTIME, absolute VTIME and immediate vs. delayed (one second) execution.

3. Success Criteria

   Changing flow from 'BLOCK_A' to 'BLOCK_B' works.

2.30 test_flowpattern.py

This test is based on `full_test/dynamic/branch/single`.

1. Purpose of Test

Test that the flow command switches from one block to another.

See Figure 2.18 for the test pattern.

2. Test Actions

   Add a schedule, start the pattern 'IN_C0'. After checking that nodes 'BLOCK_IN0' and 'BLOCK_A' are visited, change the flow with the flowpattern command at pattern 'IN_C0' from 'A' to 'B'. Check that the flowpattern command is in the low priority queue and then start the pattern 'IN_C0'. Check that the flowpattern command is processed in the low priority queue and node 'BLOCK_B' is visited.

   Test the four combinations of relative VTIME, absolute VTIME and immediate vs. delayed (one second) execution.

3. Success Criteria

   Changing flow from 'BLOCK_A' to 'BLOCK_B' works.

2.31 test_flush.py

Tests various variants of the flush command. The schedules use timing messages, flow and flush commands, and blocks. The flow commands are used to fill the command queues of the block. They switch to different timing messages, which differ in the parameter value.

Naming of the tests: `test_flow_flushX_prioY`, where X are the queues to flush. Allowed values: None, 0, 1, 2, 01, 02, 12, 123. Y is the priority of the flush. Allowed values: 0, 1, 2. Naming of patterns: P-queueX-prioY, where X and Y as described above.

Schedules used:
schedules/flush-queue01-prio0.dot
schedules/flush-queue01-prio1.dot
schedules/flush-queue01-prio2.dot
schedules/flush-queue23-prio0.dot
schedules/flush-queue23-prio1.dot
schedules/flush-queue23-prio2.dot

Structure of each test: add the schedule, start the pattern twice. The second startpattern executes the commands which are written to the queues on first startpattern. Check for flushed queues and the executed flush command after a delay of 0.1 seconds. Four tests use the same schedule to minimize the numer of schedules. Each schedule uses 4 CPUs.

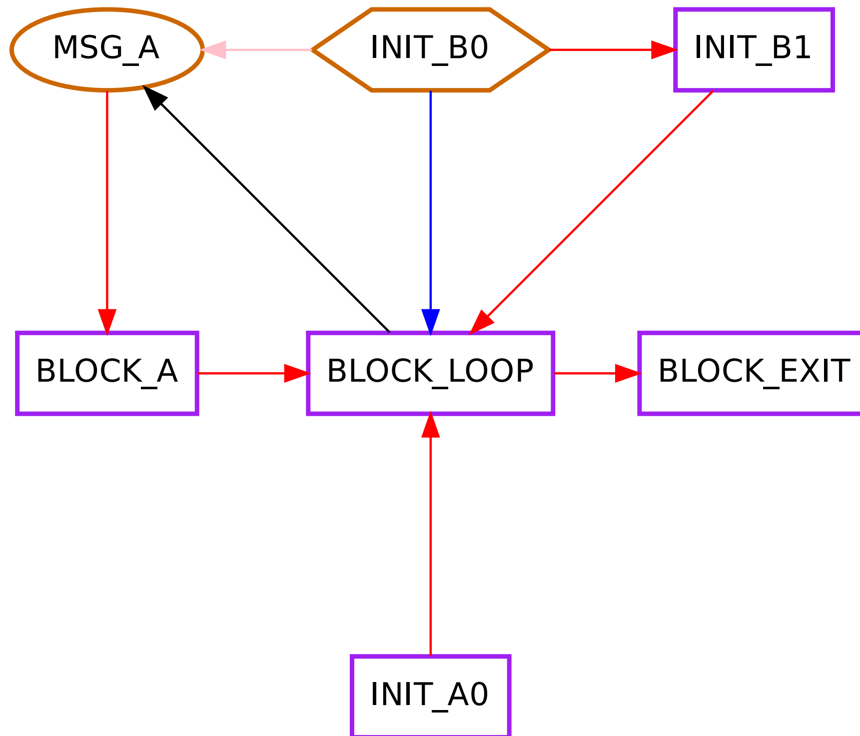


Figure 2.19: Pattern for the dynamic loop test

2.32 test_loop.py

This test was `full_test/dynamic/loop`.

1. Purpose of Test

   Test loop with flow initializer.

   See Figure 2.19 for the test pattern.

2. Test Actions

   Add a schedule and start pattern 'IN_A'. Check the visited nodes. Nodes 'INIT_A0', 'BLOCK_LOOP', and 'BLOCK_EXIT' should be visited. Start pattern 'IN_B' and check the visited nodes. All nodes should be visited.

3. Success Criteria

   In the end all blocks are visited.

2.33 test_lzma.py

There is a bug in the lzma decompression methods. The memory is not correctly allocated. This occurs with large schedules and long pattern names. The large schedule contains a loop of timing messages connected to one block.

OK test (`test_large_patternname_ok`): use a schedule with 862 messages and a pattern name of 30 chars. This works without an exception, including start of pattern.

Fail test: use a schedule with 1000 messages and a pattern name of 30 chars. This does not work. Ends with SEGV (return code -11).

2.34 test_memory.py

There are 15 tests to tet the correct memory consumption of the schedules. Oversized schedules schuld be rejected to prevent the datamaster being corrupted.

Four tests add two large schedule into the datamaster. This works fine. Then a third schedule is loaded, in each test for a different CPU. This should fail with return code 250.

Four additional tests check the memory limit more precisely. These tests use generated schedules just with a given nuber of blocks. Each block has its own pattern. A test with the theoretical limit of 1874 blocks fails. A similar test with 1869 blocks is Ok, while a test with one further block detects a failure. Also, a test with 1875 blocks detects a failure. These four tests use CPU 0.

Two tests check the memory limit for all 4 CPUs. The OK-test loads 1869 blocks into CPU 0, 1, 2 and 1675 blocks into CPU 3. The Fail-test uses one additional block on CPU 3.

A group of five tests (`test_memory_full_msg_*`) test the memory with schedules with timing messages in a loop. Due to validation checks there are 1000 messages in one loop allowed. The generator for the schedules splits the timing messages into two loops when there are more than 1000 nodes.
`test_memory_full_msg_small` uses 10 timing messages to test the generator.
`test_memory_full_msg_half` uses 900 timing messages for the test.
`test_memory_full_msg_ok` uses 1867 timing messages, which is the maximal number of nodes allowed.

`test_memory_full_msg_infinite_loop_ok` uses 1000 timing messages. This is added and started.
`test_memory_full_msg_infinite_loop_fail` uses 1001 timing messages. This schedule is rejected and not added.

2.35 test_originStartthread.py

1. `test_threadsStartStop`
   Thread 0 assigns Tmsg{1,2,3} and Block{1,2,3} to thread 1,2,3 and starts these threads. The test then starts pattern D (Tmsg4 and Block4) to show that this does not stop the other threads. The test stops pattern A which stops thread 1. The test stops pattern C which stops thread 3. The test stops node Block2 which stops thread 2. See Figure 2.20 for the schedule. For the Tmsg and the Blocks the diagram shows the threads. There is no command to stop a thread.

2. `test_nodeInTwoThreads`
   This test demonstrates that a node can exist in two threads. Tmsg1 ist in thread 1, Tmsg2 is in thread 2. Successor for both is Tmsg3. Thus we have a timing message from Tmsg3 for each timing message from Tmsg1 and Tmsg2. The loop in thread 0 (nodes Tmsg0, OriginN, StartthreadN, Block0) starts thread 1 and 2 every 10ms. Thread 1 and 2 end with Block3. See Figure 2.21 for the schedule.

3. `test_startStopAllThreads`
   Run a pps pattern on all threads and all CPUs. Halt all threads and check this state. Then start four threads 0,1,2,3 on all CPUs and check. For more than 8 threads: Then start four threads 8,9,10,11 on all CPUs and check. Then start four threads 16,17,18,19 on all CPUs and check. Then start four threads 24,25,26,27 on all CPUs and check.

4. `test_startStopBlocks`
   Run a pps pattern on all threads of CPU 0. Stop block Block0b four times and check the result. This block is on thread 1, and for more than 8 threads also on thread 9, 17, 25. Thus we check that one of these threads is stopped for more than 8 threads.

2.36 test_originTwothreads.py

1. Purpose of Test

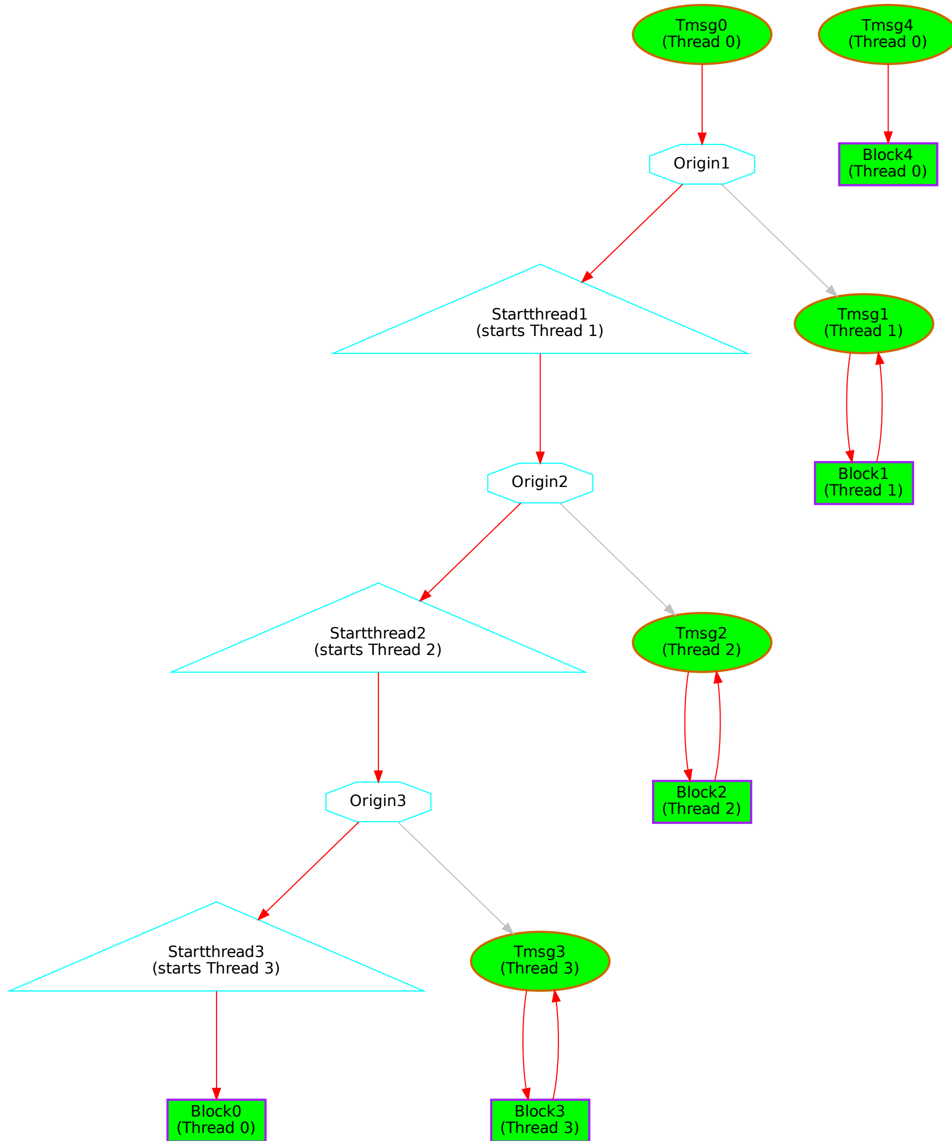


Figure 2.20: Schedule which assigns origins and then starts threads with these origins. This is used in test_originStartthread.py

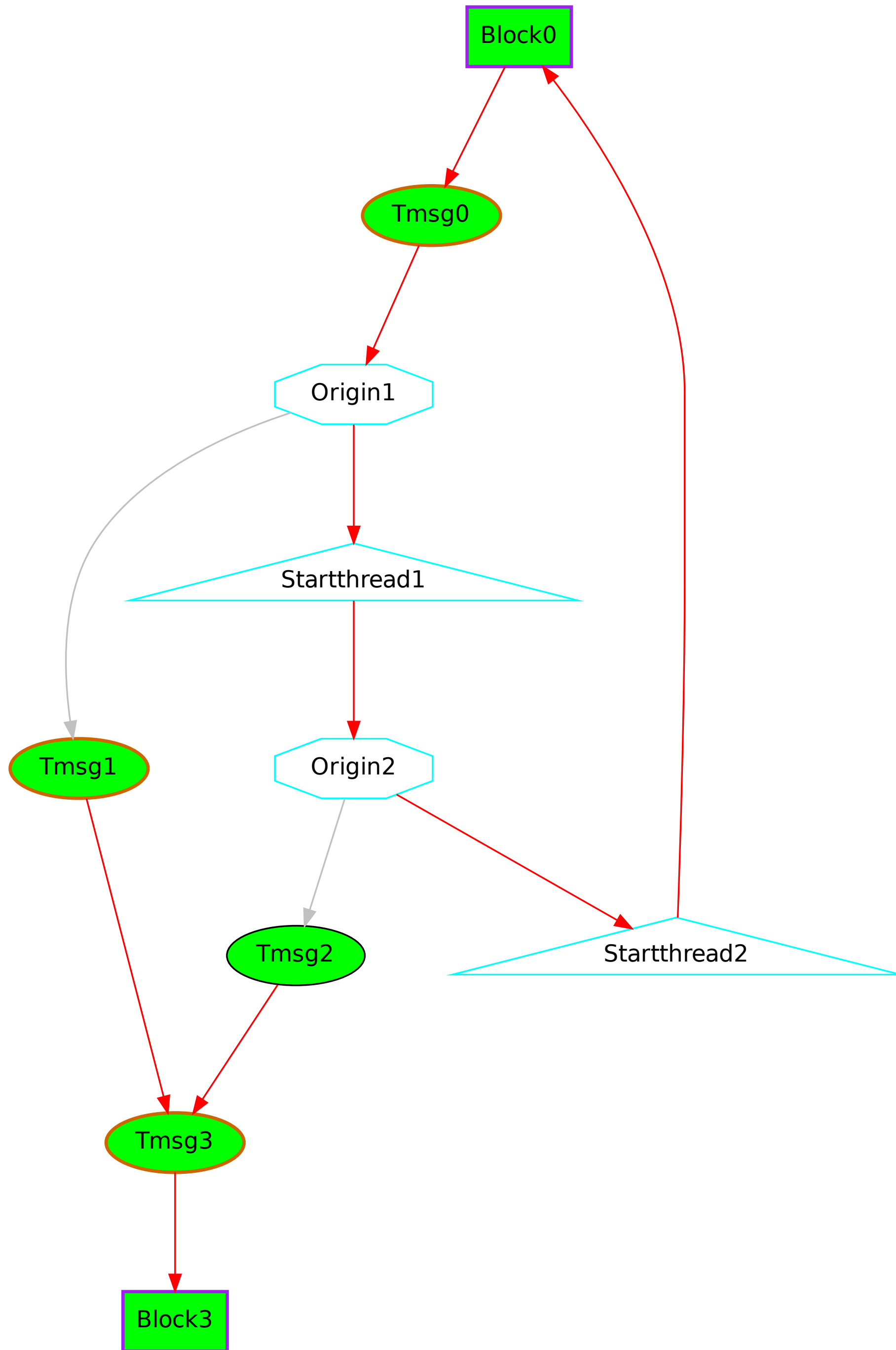


Figure 2.21: Schedule with a node in two threads. This is used in test_originStartthread.py

This test shows that an origin node and a startthread node work.

2. Test Actions

   Origin and Start are in pattern B. Pattern B is started and produces with Tmsg1 events with parameter 1. Tmsg2 produces events with parameter 2. The node Start starts thread 1 and is the origin for this thread. Thus, thread 1 is started every 50 ms by itself. There is a snoop action for 1 s.

3. Success Criteria

   The result of the snoop should get at least 15 events with parameter 1 and one message with parameter 2.

2.37 test_overwrite.py

1. testOverwrite1 Load and start pattern A, a loop of a message and a block. Stop the pattern and overwrite the block with a blockalign. Analyse the timing messages with the parameter field.
2. testOverwrite2 Load a schedule with a switch. Executing the switch interchanges the defdst to EvtA with the altdst to EvtB. Then the schedule loops over EvtB. Overwrite the schedule with the original schedule changes the edges back to defdst to EvtA and altdst to EvtB. Then the switch is executed again. Analyse the timing messages with the parameter field.
3. testOverwrite3 Load a schedule with a switch. Executing the switch interchanges the defdst to EvtA with the altdst to EvtB. Then the schedule loops over EvtB. Overwrite the schedule with a smiliar schedule which replaces the switch with a flow and the switchdst edge with a flowdst edge. Analyse the timing messages with the parameter field.

2.38 test_overwriteQueue.py

The test `test_overwriteSchedules` overwrites a block with a low priority queue with a block that has a low and a high priority queue. The low priority queue is always needed to stop the pattern PPS_Q. The test is successful when the downloaded schedules are isomorphic to the expected schedules with the meta nodes. The messages are checked for the appropriate number of messages.

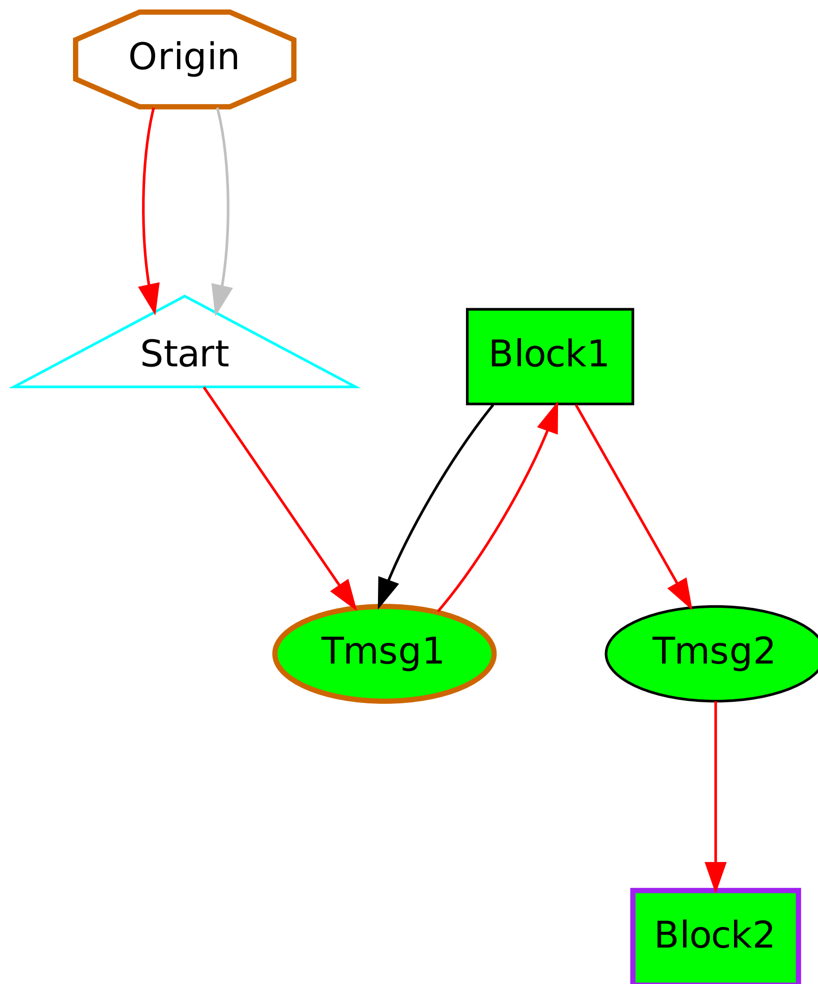


Figure 2.22: Schedule with two threads used in test_originTwothreads.py

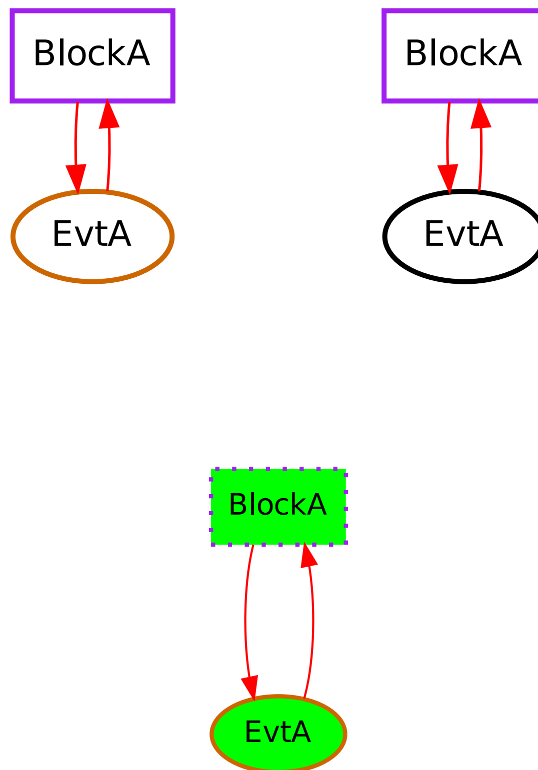


Figure 2.23: Schedules for testOverwrite1 (test_overwrite.py). The first schedule consists of a block and an event. The parameter of the messages is 1, which is checked in the snooped messages. This pattern runs for a second. It is overwritten with the second schedule containing a blockalign and an event. This event sends parameter 2, which is also checked in the snooped messages. The third schedule is the result.

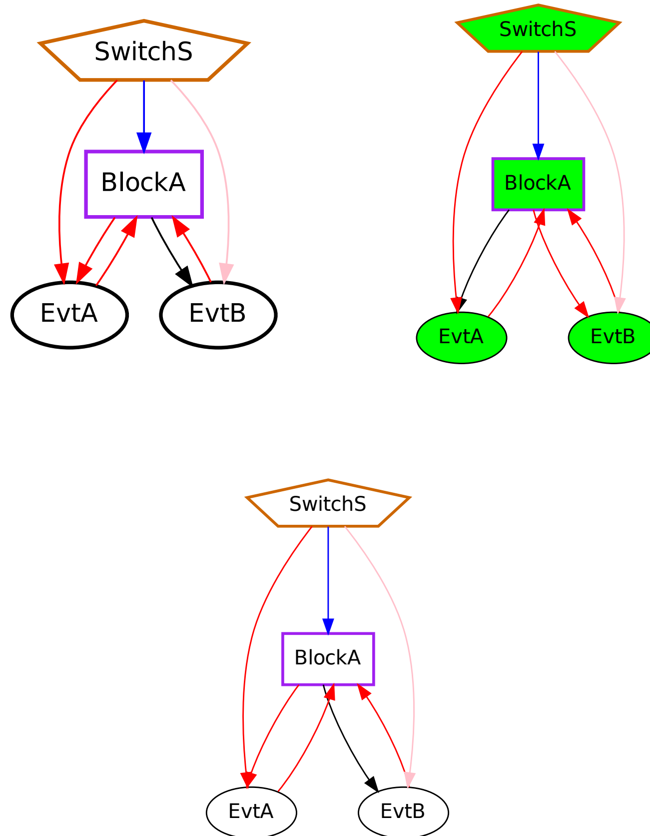


Figure 2.24: Schedules for testOverwrite2 (test_overwrite.py). The first schedule shows the defdst to EvtA and the altdst to EvtB. After executing the switch, the second schedule shows the defdst to EvtB and the altdst to EvtA. Overwritting with the original scheulde results in the third schedule.

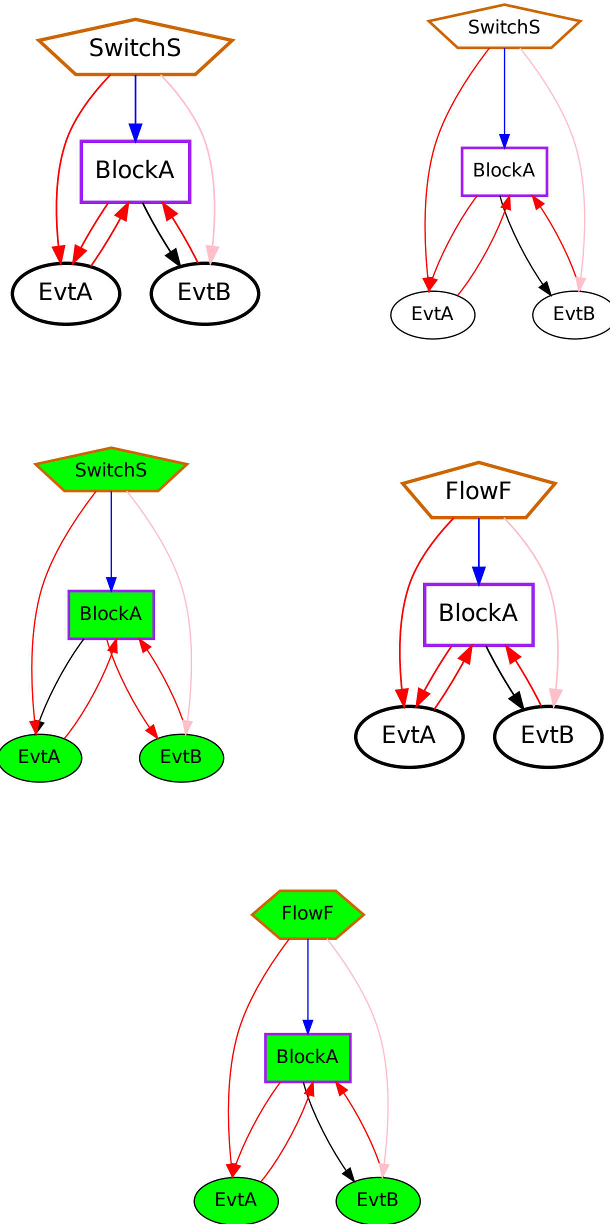


Figure 2.25: Schedules for testOverwrite3 (test_overwrite.py). The first schedule contains a switch. The second schedule is the check before executing the switch. The third schedule is the check after executing the switch. The fourth schedule overwrites the first with a flow node. The fifth schedule is the result.

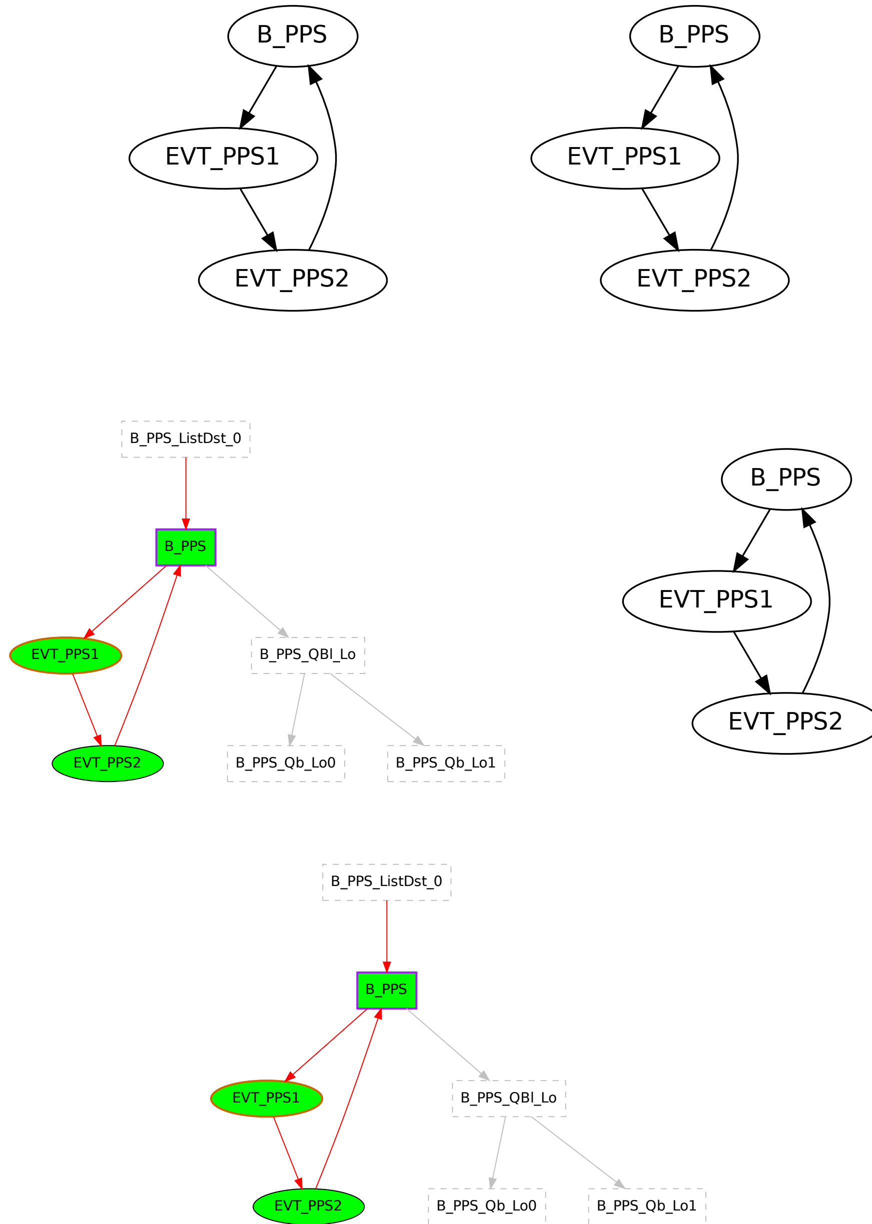


Figure 2.26: Schedules for test_overwriteSchedules (test_overwriteQueue.py). The first schedule contains a block with no queue. The second overwrites this with a low priority queue. The result is the third schedule. The fourth schedule overwrites this with a block with two queues. The result is the fifth schedule.

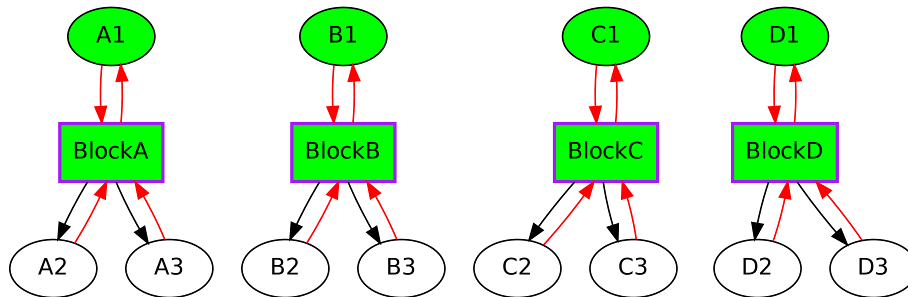


Figure 2.27: Schedule for test_parallelBranch.py.

2.39 test_parallelBranch.py

Use schedule `branch1.dot` to test branching with flow commands with absolute time offset. The checks use snoop for specific event numbers. The four tests are for CPU 0, CPU 0 and 1, CPU 0, 1, and 2, all 4 CPUs. Each of these pattern A, B, C, D runs on a different CPU.

2.40 test_patternStartStop.py

The test `test_startStopPattern` runs a pps pattern on all threads of CPU 0. Stop pattern Block0b which should fail, since not a pattern name. Stop pattern PPS0b which is OK. This pattern runs on threads 1, 9, 17, 25 (the last three for 32 threads). Check which thread is stopped. Start pattern Block0b which should fail, since not a pattern name. Start pattern PPS0b on the thread, which was stopped before. The start is OK. Check that all threads (8 or 32) are running.

2.41 test_pps.py

`test_pps.py` (pps: pulse per second) is a collection of tests based on schedules with a simple loop of a timing message and a block.

1. test_pps is a basic test with a schedule which sends two timing messages every second. The test checks with `saft-ctl snoop` the timing messages.

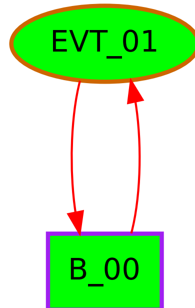


Figure 2.28: Pattern for test_ppsAdd.

2. test_ppsAdd is a basic test with a schedule which sends one timing message every second. The test checks with `saft-ctl snoop` the timing messages. Download the schedule with `dm-sched status` and compare with the uploaded schedule with `scheduleCompare`. The two schedules should be isomorphic.

3. testPpsAdd0 Add two schedules. The first schedule contains two nodes and an edge. The second adds some edges to the first schedule. Start pattern A and use a flow command to trigger messages. The status of the schedules is compared against known schedule files with `scheduleCompare`. The messages are snooped and checked. During snoop start pattern A. This produces 1 message. Pattern A finishes. Download the schedule for later compare. Add a schedule which contains two edges. Queue a flow command with quantity 10 to block B_A. Again start pattern A. The flow command triggers the next messages. At the end, 12 messages are produced and the pattern loops in block B_A.

4. testPpsAdd1 Add two schedules. The first schedule contains pattern A with two nodes and an edge. The second adds a similar pattern B. The messages are snooped and checked.

5. testPpsAdd2 Test that the validation for nodes connected by defdst or altdst on the same CPU works. Add a first schedule. Try to add a second schedule. This fails due to an edge from CPU 0 to CPU 1. Add a third schedule with a target edge from CPU 0 to CPU 1. This works.

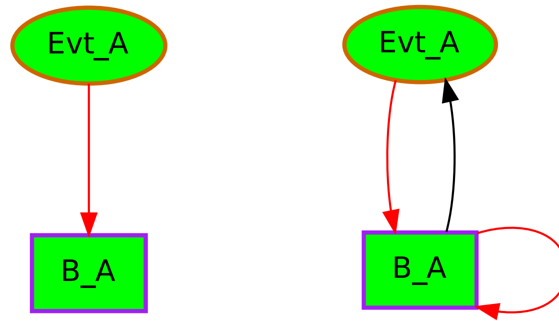


Figure 2.29: Schedules for testPpsAdd0 (test_pps.py). The first produces one timing message. The second is the result after adding more edges.

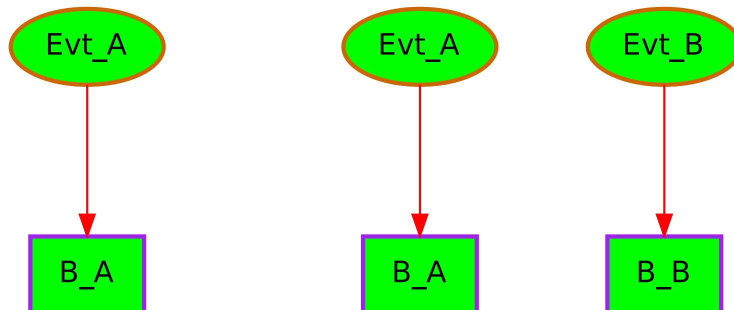


Figure 2.30: Schedules for testPpsAdd1 (test_pps.py).

6. testPpsAdd3 Test with five schedules. Add two schedules, remove the third, add the fourth, remove the fifth.

7. testPpsAdd4 Test removing a pattern which is running. This is rejected as expected. Then stop the pattern A and remove it. Check that the appropriate number of messages is produced.

8. testPpsAdd5 Test removing a pattern which is running. This is rejected as expected. When pattern A has finished, remove it. Check that the appropriate number of messages is produced.

9. testPpsAdd6 Test adding a node which changes the pattern entry. Check that the appropriate number of messages is produced.

10. testPpsAdd8 Test adding a schedule with a node with name ending in ListDst_3. This fails, because this name is generated during upload and a collision happens. Second attempt is to add a similar schedule with the name Block0_0_ListDst_6. This works. The generated nodes have the names Block0_0_ListDst_x with x from 0 to 5. Download the schedule including meta nodes and compare it to the expected one with `scheduleCompare`.

2.42 test_prioAndType.py

This test was `full_test/static/prio_and_type`.

1. Purpose of Test

   The test checks the relative and the absolute time values for two nodes in a four node pattern. See Figure 2.42 for the test pattern and 2.43 for the test pattern with meta nodes, displaying the priority queues.

2. Test Actions

   Add the pattern, check the relative time values. Clear the datamaster. Add the pattern again and check the absolute time values.

3. Success Criteria

   Two checks of the time values with `dm-cmd rawqueue`.

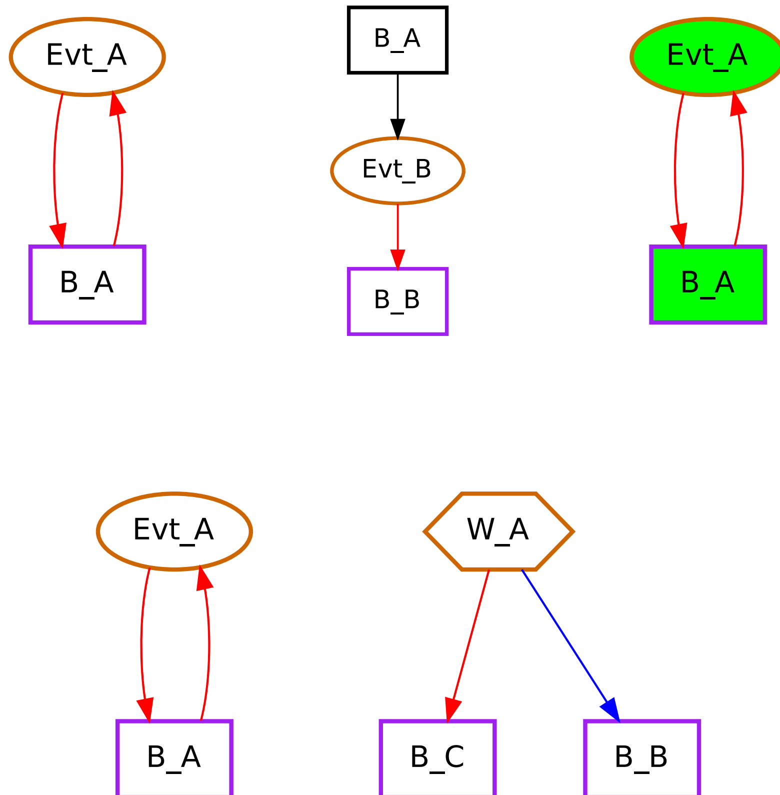


Figure 2.31: Schedules for testPpsAdd2 (test_pps.py). The upper part of the figure shows the first schedule on the left. Try to add the middle schedule. This fails, since block B_A is on CPU 0 but Evt_B and B_B are on CPU 1. The schedule on the right is the result. The lower part of the figure shows the result after adding a wait node W_A with two blocks B_B and B_C to the original schedule of Evt_A and B_A.

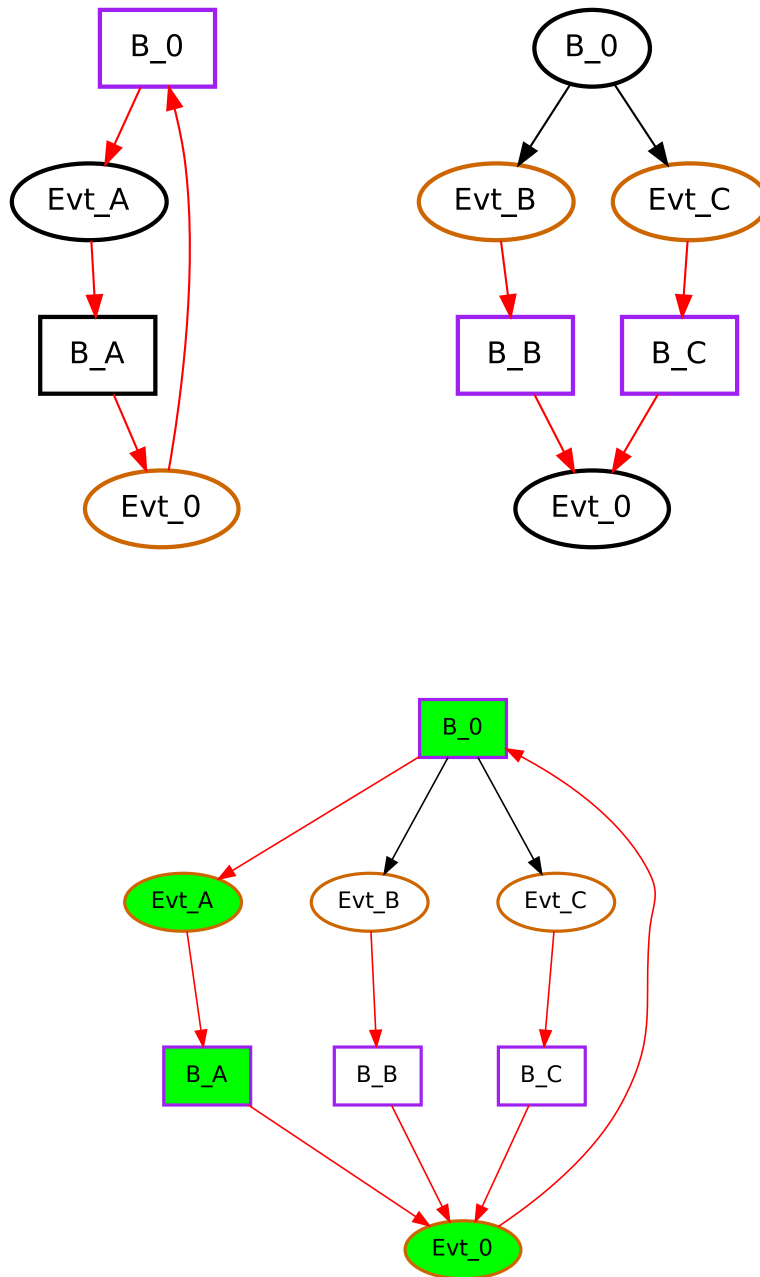
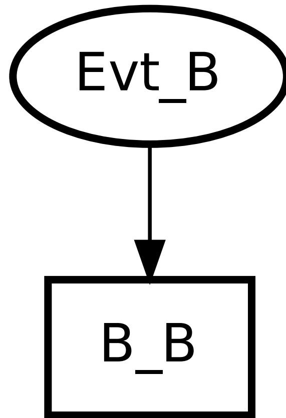


Figure 2.32: Schedules for testPpsAdd3 (test_pps.py), first step. Starting with the upper left schedule, adding the upper right schedule results in the schedule in the lower part of the figure.



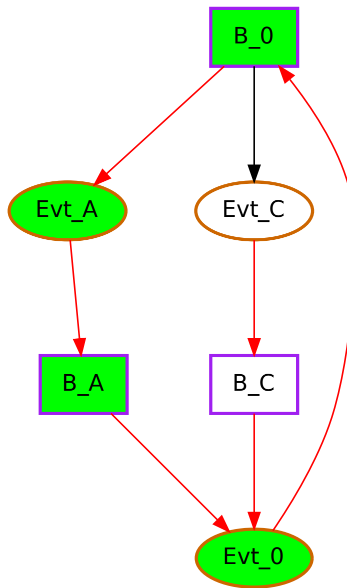


Figure 2.33: Schedules for testPpsAdd3 (test_pps.py), second step. Remove the schedule with Evt_B and B_B results in the second schedule.

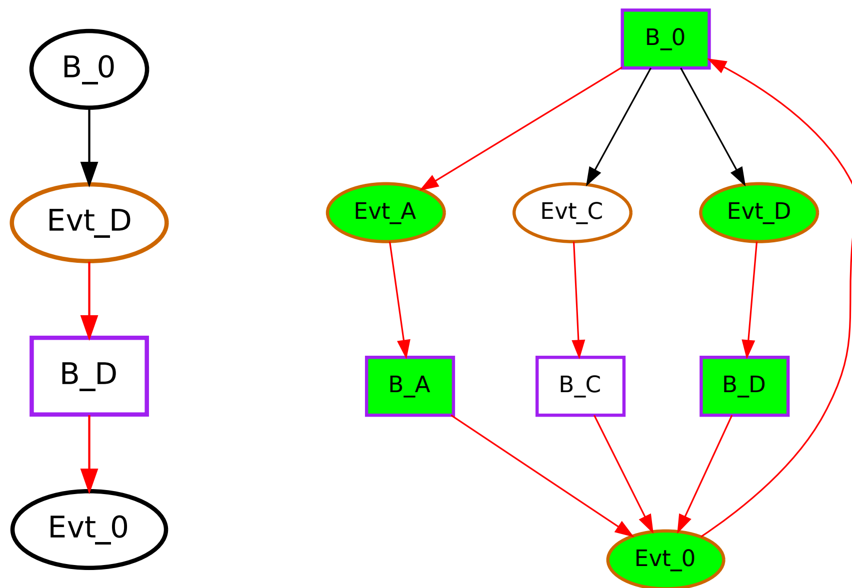
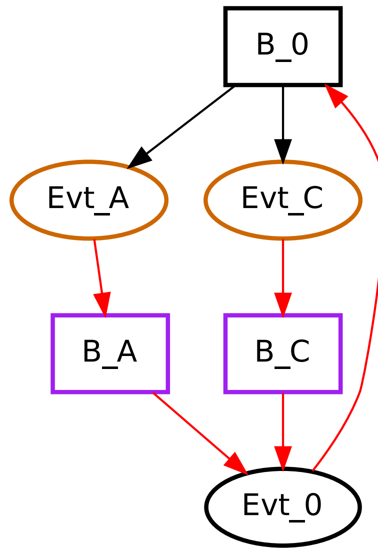


Figure 2.34: Schedules for testPpsAdd3 (test_pps.py), third step. Add the schedule with Evt_D and B_D results in the second schedule.



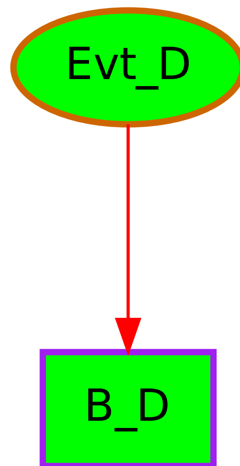


Figure 2.35: Schedules for testPpsAdd3 (test_pps.py), fourth step. Remove the first schedule results in the schedule with Evt_D and B_D.

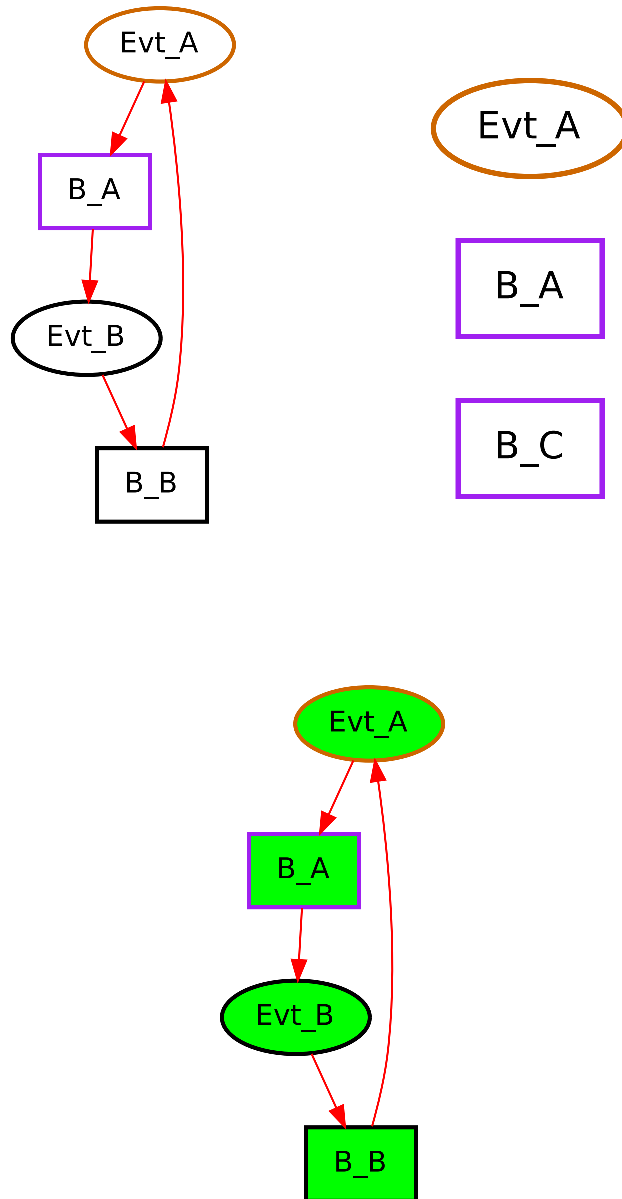
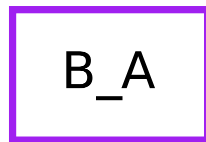


Figure 2.36: Schedules for testPpsAdd4 (test_pps.py), first part. Removing the three nodes Evt_A, B_A, and B_C from the first schedule fails since pattern A is running. The resulting schedule in the lower part is the same as the first schedule.



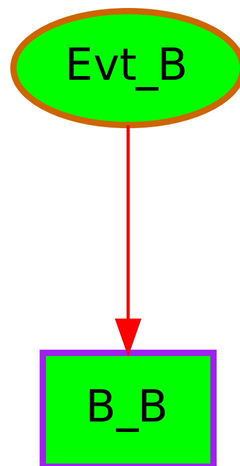


Figure 2.37: Schedules for testPpsAdd4 (test_pps.py), second part. After stopping pattern A, nodes Evt_A and B_A are removed and the result is the last schedule.

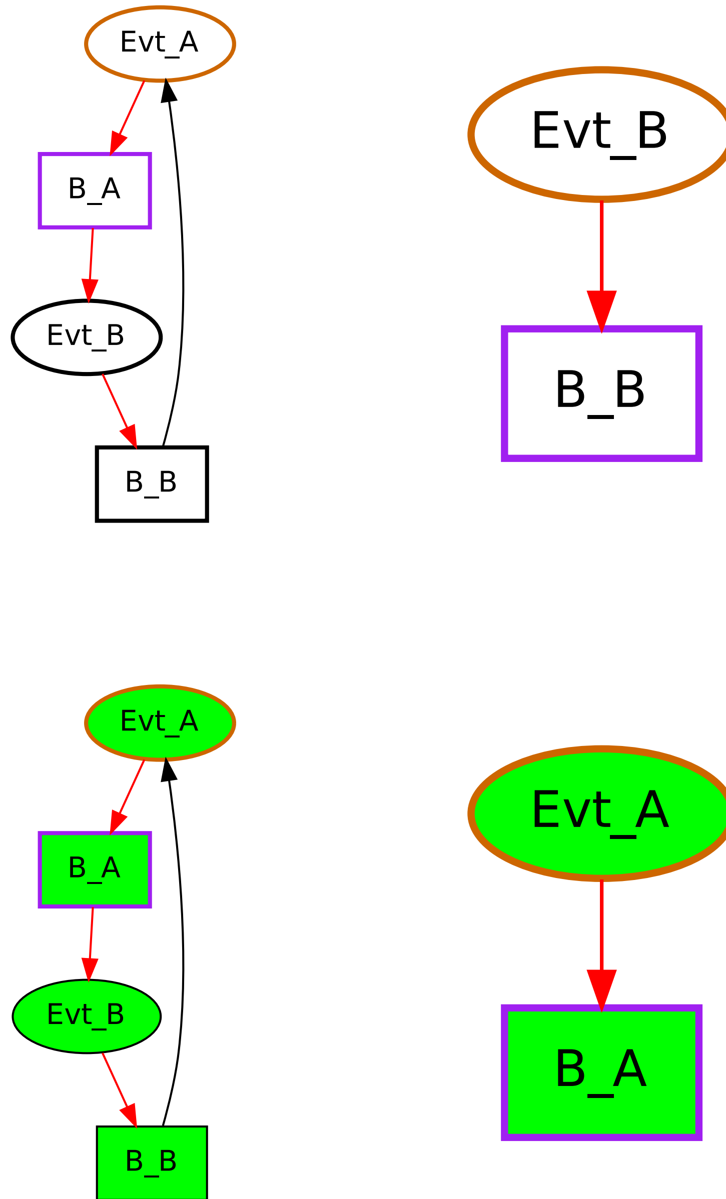


Figure 2.38: Schedules for testPpsAdd5 (test_pps.py). From the left upper schedule remove nodes Evt_B and B_B while pattern A is running. This fails with return code 250. Pattern A finishes. Trying to remove Evt_B and B_B again results in the lower right schedule with nodes Evt_A and B_A.

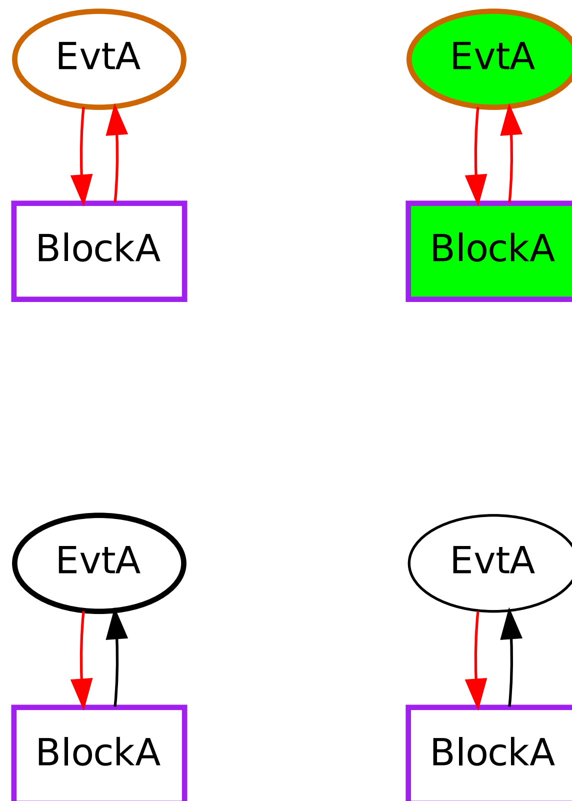


Figure 2.39: Schedules for testPpsAdd6 (test_pps.py), first part. Start pattern A in the upper left schedule. The result is the upper right schedule. Overwrite the schedule with an altdst edge from BlockA to EvtA and no pattern entry. Result is the lower right schedule. Cannot start pattern, because entry is missing.

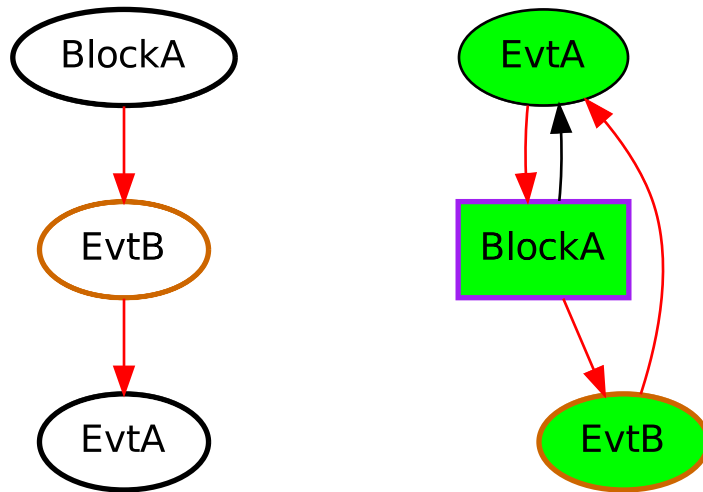


Figure 2.40: Schedules for testPpsAdd6 (test_pps.py), second part. Add the left schedule with EvtB as pattern entry. Then start pattern A. This results in the right schedule.

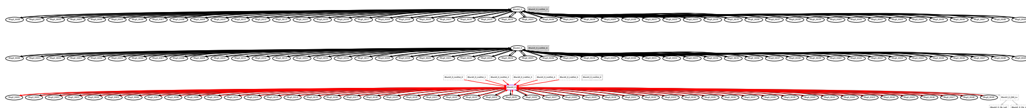


Figure 2.41: Schedules for testPpsAdd8 (test_pps.py). The first one fails to load. The second is added successfully. Compare it with the third schedule.

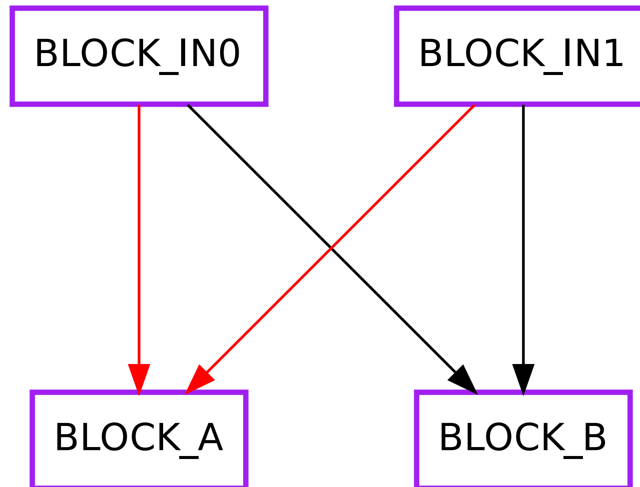


Figure 2.42: Pattern for the static priority and type test

2.43 test_priorityQueue.py

`test_priorityQueue.py` tests the execution of priority queues.

Generate a schedule with 5 or 6 altdst edges to a central block. These edges connect this central block with tmsg nodes. The tmsg nodes have a defdst edge to the central block.

With a generated schedule test altdst. Use a loop over all tmsg nodes to switch the destinations such that the schedule flow from the central block switches through all tmsg nodes. See Figure 2.44

2.44 test_referenceEdges.py

`test_referenceEdges.py` tests that reference edges work. At most three reference edges are allowed for a node. testReferenceEdgeLoop1 uses 1 reference edge. testReferenceEdgeLoop3 uses 3 reference edges. testReferenceEdgeLoop4 uses 4 reference edges, which is forbidden.

1. testReferenceEdgeSimple Use a schedule with an edge of type reference between two loops (a block and a tmsg). The loops run with 10Hz. Check for the correct parameter value when using the reference.

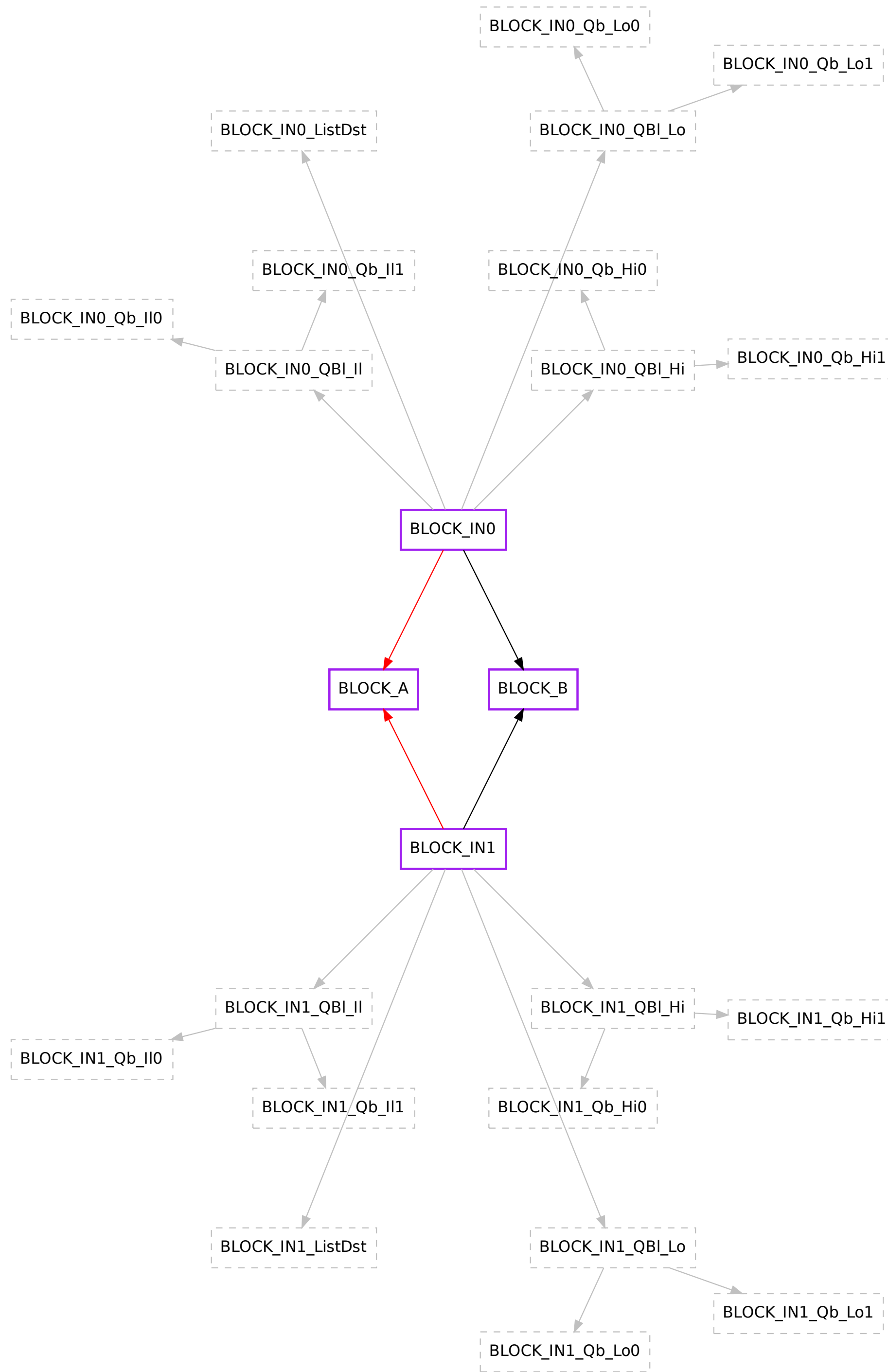


Figure 2.43: Pattern for the static priority and type test with meta nodes

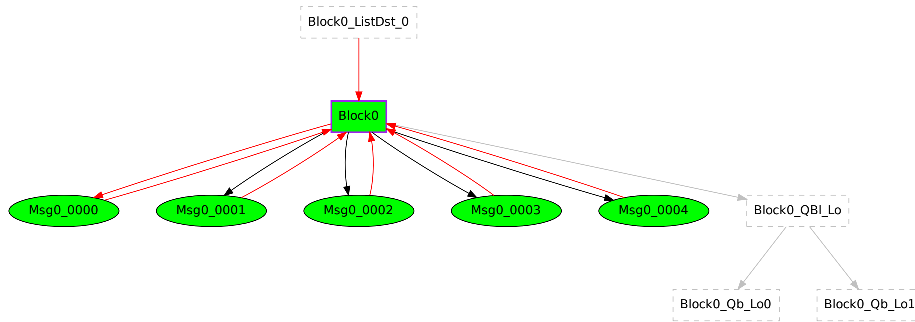


Figure 2.44: Pattern for the priority queue tests

2. testReferenceEdgeLoop1 Use a schedule with a loop of a block and two tmsg. The loops run with 1Hz. There is a reference between the two tmsg nodes. Check for the correct gid value in the timing messages. Check for the correct parameter value when using the reference.

3. testReferenceEdgeLoop3 Use a schedule with a loop of a block and four tmsg. The loops run with 1Hz. There is are references between the tmsg nodes. Check for the correct gid value in the timing messages. Check for the correct parameter value when using the reference. See Figure 2.45.

4. testReferenceEdgeLoop4 Use a schedule with a loop of a block and five tmsg and four references. This is not allowed. Adding the schedule fails.

2.45 test_remove.py

1. testRemove1 Load and during snoop start pattern A, a loop of a message and a block. This produces messages with 10Hz. Download the schedule for later compare. Stop the pattern and remove the pattern which is nearly the same, but the block has no type. The result is a schedule with only the block.

2. testRemove2 Load and start pattern A, a loop of a message and a block. Stop the pattern and remove parts of the pattern which is nearly the same, but the block has no type and a different event node EvtC. Remove fails, because EvtC is unknown in the existing schedule.

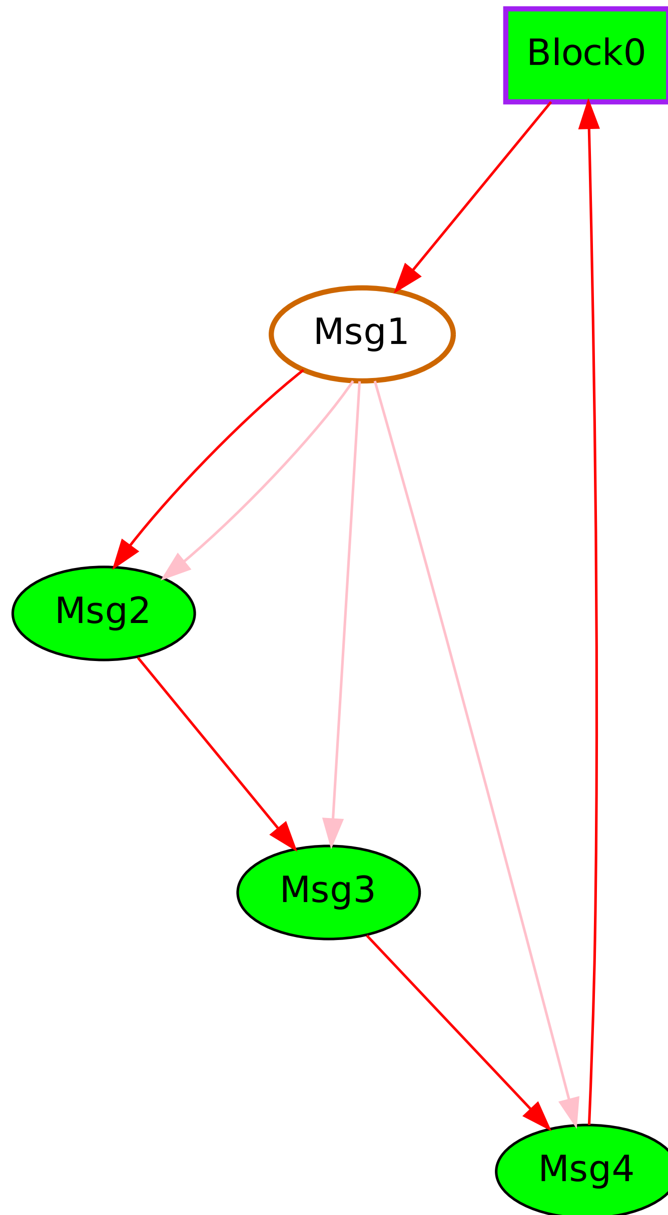


Figure 2.45: Schedule for testReferenceEdgeLoop3 (test_referenceEdges.py).

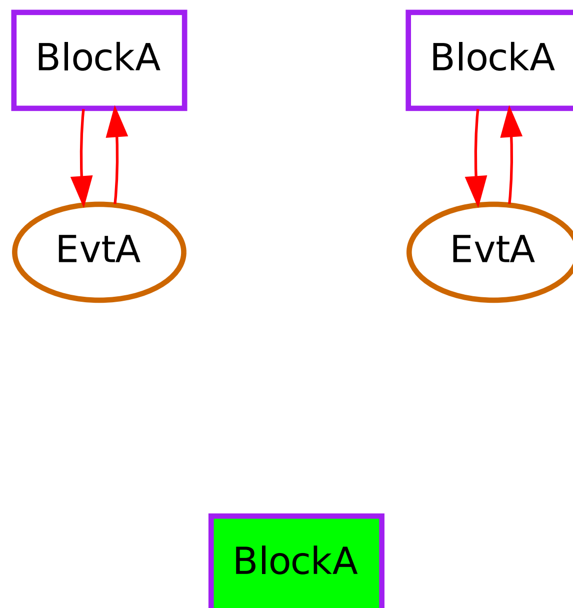


Figure 2.46: Schedules for testRemove1 (test_remove.py). The first schedule is the loop. The second schedule is removed from the first, resulting in the third schedule.

The difference to testRemove1 is that the schedule for removal contains EvtC instead of EvtA.

3. testRemove3 Load and start pattern A, a loop of message EvtA and block BlockA. Stop the pattern and remove the pattern which is nearly the same, but the block BlockA has no type. The remove command fails.

   The difference to testRemove1 is that the schedule for removal contains EvtA without a type. Thus, the remove command tries to remove BlockA, which would leave EvtA childless.

4. testRemove4 Load and start pattern A, a loop of message EvtA and block BlockA. Stop the pattern and remove EvtA. The result is a schedule with only the block. Then add node EvtA of type switch and block BlockB. Thus, EvtA changes the type.

2.46 test_runAllSingle.py

This test was `full_test/dynamic/basics/run_all_single`.

1. Purpose of Test

   Run a very basic schedule on all four CPUs.

   See Figure 2.51 for the test pattern.

2. Test Actions

   Add a schedule with four blocks to the datamaster. For each CPU, check that no block is visited. Then start a pattern for one CPU and check that the block specific for this pattern is visited.

3. Success Criteria

   For each pattern the correct CPU is used.

2.47 test_runCpu0Single.py

This test was `full_test/dynamic/basics/run_cpu0_single`.

1. Purpose of Test

   Add the test schedule to the datamaster, start patterns and check which nodes were visited.

   See Figure 2.52 for the test pattern.

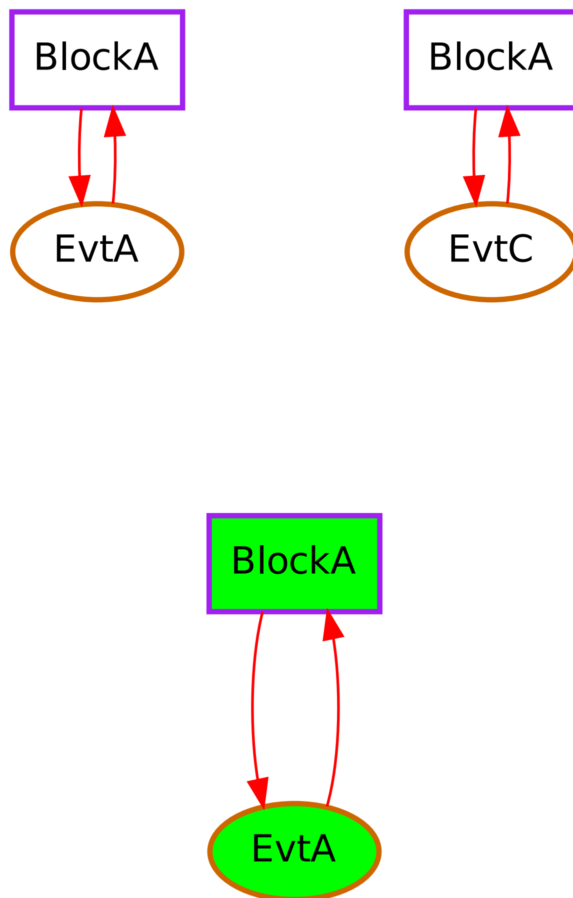


Figure 2.47: Schedules for testRemove2 (test_remove.py). The first schedule is the loop. The second schedule is removed from the first, resulting in the third schedule.

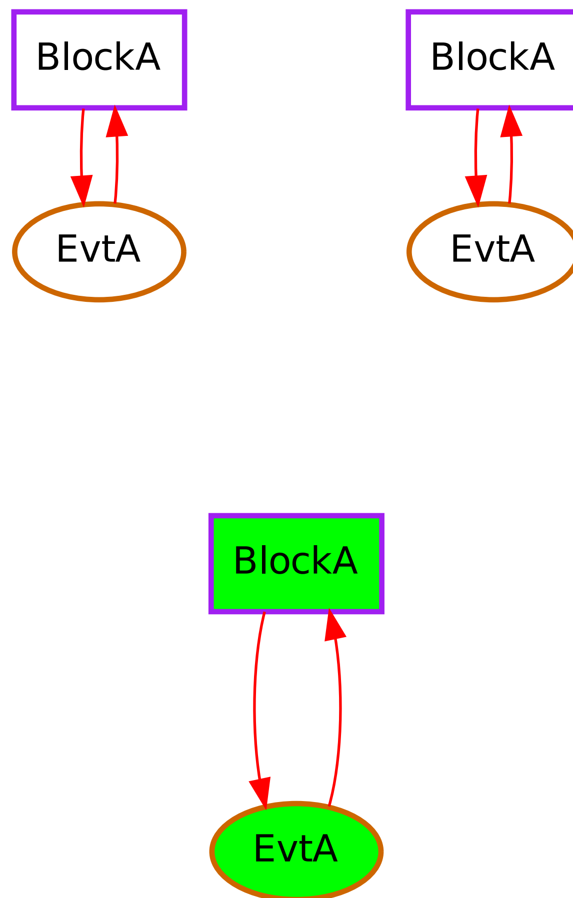


Figure 2.48: Schedules for testRemove3 (test_remove.py). The first schedule is the loop. The second schedule is removed from the first, resulting in the third schedule.

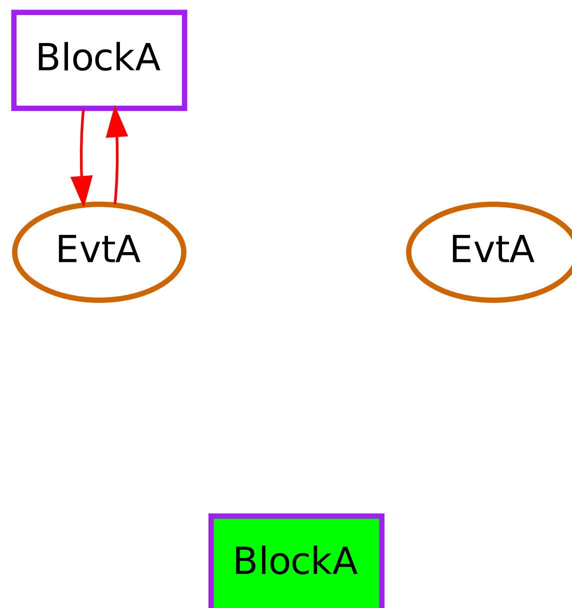
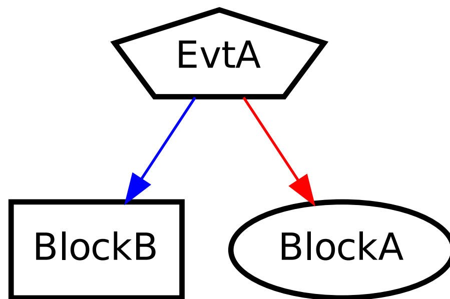


Figure 2.49: Schedules for testRemove4 (test_remove.py). First part. First schedule consists of EvtA and BlockA. Remove the second schedule and get the third schedule as result.



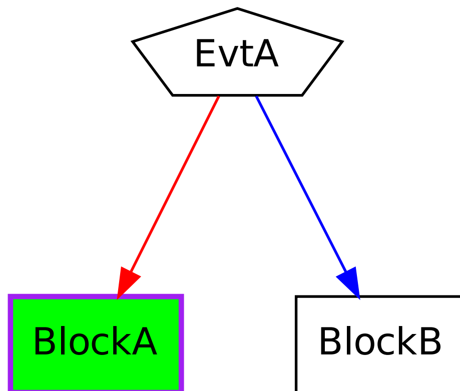


Figure 2.50: Schedules for testRemove4 (test_remove.py). Second part. Add the first schedule with EvtA as a switch node. The result is the second schedule.

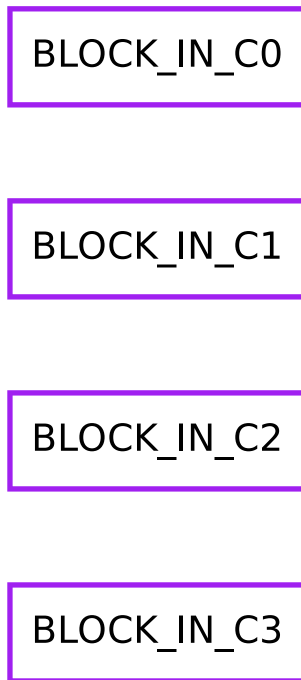


Figure 2.51: Pattern for the dynamic run all test

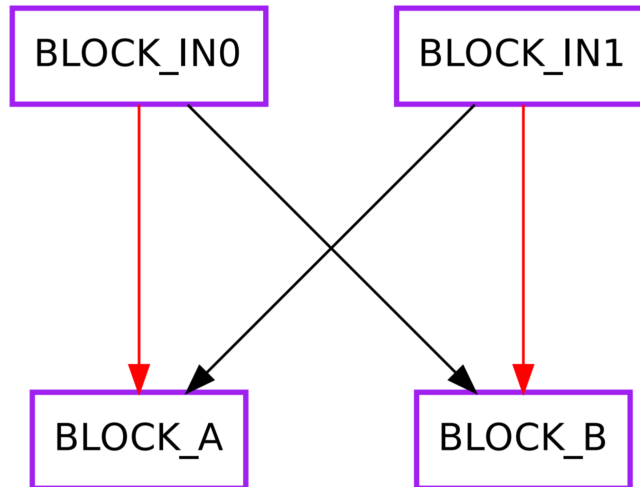


Figure 2.52: Pattern for the dynamic run CPU 0 single test

2. Test Actions

   First check that no block is visited. Start pattern 'IN0'. Check that 'BLOCK_A' and 'BLOCK_IN0' are visited. Start pattern 'IN1'. Check that in addition 'BLOCK_B' and 'BLOCK_IN1' are visited.

3. Success Criteria

   In the end all blocks are visited.

2.48 test_safe2remove.py

`test_safe2remove.py` tests to remove a pattern from a running schedule. Test steps:

1. Clear datamaster
2. Add schedule
3. Start pattern 'G1_P1'
4. Check removal of one pattern, should fail while pattern is running.

5. Abort pattern 'G1_P1'
6. Check removal of this pattern, should be valid, since pattern is not running.
7. Remove this pattern.
8. Check status of remaining schedule.

These test steps are applied to a bunch of schedules. The schedules use 1 to 4 CPUs whith 1, 2, 4, and 9 pattern beside the default pattern. There are 1, 10, or 150 blocks per pattern.

2.49 test_schedules.py

`test_schedules.py` collects schedules from INT or PROD. All patterns are started. The tests need 4 CPUs. Tests that the schedules are compiled and loaded. This includes two schedules from INT and PROD as of 2024-10-02.

2.50 test_simultaneousThreads.py

`test_simultaneousThreads.py` tests that startthread nodes can start a number of threads simultaneously. In Figure 2.53 StartThread (runs in thread 0) starts threads 1, 2, and 3. To check that the threads are running, each threads produces timing messages with the thread number as parameter. The tests run for two threads (0 and 1), four threads (0, 1, 2, 3) and all threads (8 or 32).

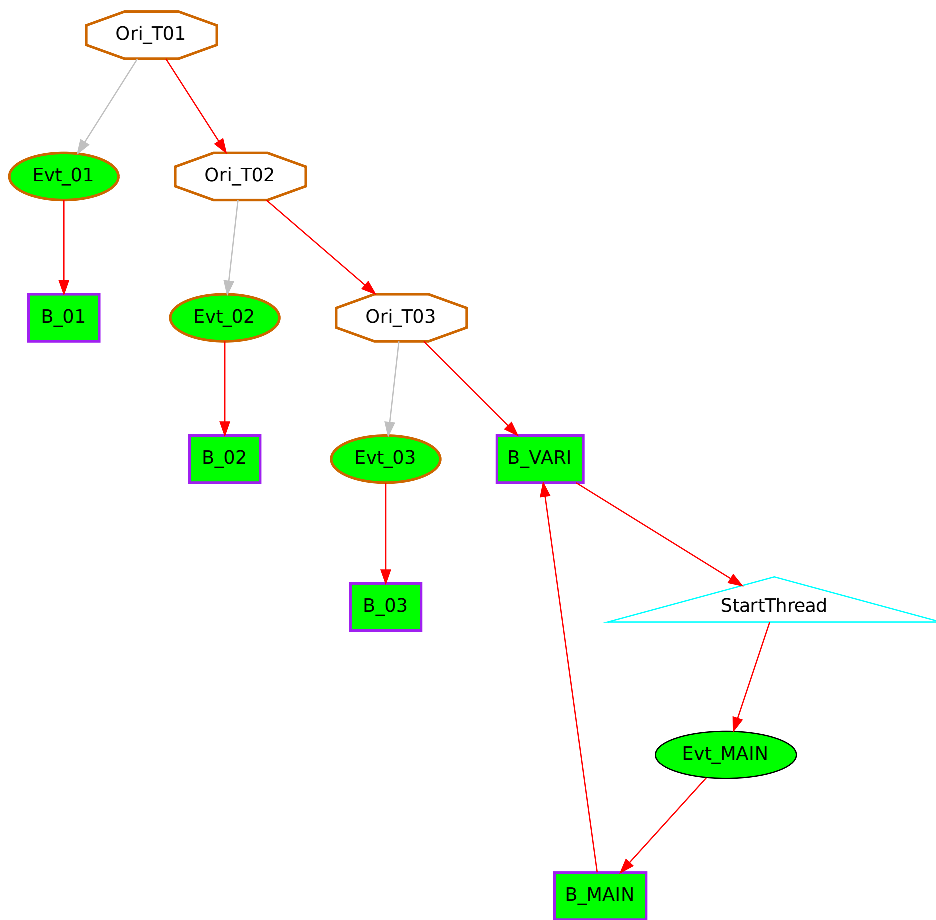


Figure 2.53: Pattern for the simultaneous start of threads

2.51 singleEdgeTest

`singleEdgeTest` tests the validation of combinations of node types and edge types. This uses libcarpedm in a special version, called dmtest. There are all private methods of libcarpedm accessible in dmtest. There are 2873 schedules with at least one edge and two nodes tested. Some schedule are enlarged with meta nodes. 209 schedules of the 2873 schedules are valid. All other schedules include a forbidden edge type or a forbidden child type.

Build process: when building the test object libcarpedm / dmtest, start with `make clean` to build a consitent test object.

`singleEdgeTest` has a mode which does not generate meta nodes. This is used to generate schedules for the pytest tests. The meta nodes are needed for the validation tests, but they are not allowed for tests which upload schedule to the lm32 firmware.

For this, the combinations of a node and an outgoing edge and, respectively, a node and an incoming edge.

The tables are compiled from source validation.cpp.

Classification of nodes:

1. Meta nodes: qinfo, qbuf, listdst
2. Event nodes: tmsg, switch, flow, flush, noop, waits
3. Command nodes: flow, flush, noop, wait

The last row combines all edge types and show the maximal number of outgoing edges.

1. the check for childless nodes (events and qinfo must have childs),
2. the sum of min and max cardinalities for the detailed edge types.

Open questions:

1. Is the list of node types complete?
2. Is the list of edge types complete?
3. Node type listdst: no rules for in- or out-edges defined. Is this correct? With release 9.0.0 of the firmware, listdst nodes have exactly one edge of type defdst to a block.
4. Edge types dynid x, dyntef, dynres: are these edge types used?
5. How to distinguish edge types target (Switch) and target (Command)?

Table 2.1: Schedule – Valid edge types per node type

Edge Type	Node Type - Out-Edge, first node												
	block	blockalign	flow	flush	listdst	noop	origin	qbuf	qinfo	startthread	switch	tmsg	wait
defdst	0..1	0..1	0..1	1	–	0..1	0..1	–	–	0..1	0..1	1	0..1
altdst	0..9	0..9	–	–	–	–	–	–	–	–	–	–	–
listdst	0..1	0..1	–	–	–	–	–	–	–	–	–	–	–
baddefdst	–	–	–	–	–	–	–	–	–	–	–	–	–
target (Switch)	–	–	–	–	–	–	–	–	–	–	0..1	–	–
target (Command)	–	–	0..1	0..1	–	0..1	–	–	–	–	–	–	0..1
flowdst	–	–	0..1	–	–	–	–	–	–	–	–	–	–
flushovr	–	–	–	0..1	–	–	–	–	–	–	–	–	–
switchdst	–	–	–	–	–	–	–	–	–	–	0..1	–	–
meta	–	–	–	–	–	–	–	–	2	–	–	–	–
origindst	–	–	–	–	–	–	1	–	–	–	–	–	–
priolo	0..1	0..1	–	–	–	–	–	–	–	–	–	–	–
priohi	0..1	0..1	–	–	–	–	–	–	–	–	–	–	–
prioil	0..1	0..1	–	–	–	–	–	–	–	–	–	–	–
dynid x	–	–	–	–	–	–	–	–	–	–	–	–	–
dynpar0	–	–	–	–	–	–	–	–	–	–	–	0..1	–
dynpar1	–	–	–	–	–	–	–	–	–	–	–	0..1	–
dyntef	–	–	–	–	–	–	–	–	–	–	–	–	–
dynres	–	–	–	–	–	–	–	–	–	–	–	–	–
sum	0..14	0..14	1..3	1..3	0..	1..2	1..2	0..	2	0..1	1..3	1..3	1..2

Table 2.2: Schedule – Valid edge types per node type

Edge Type	Node Type - In-Edge, second node												
	block	blockalign	flow	flush	listdst	noop	origin	qbuf	qinfo	startthread	switch	tmsg	wait
defdst	0..	0..	0..	0..	–	0..	0..	–	–	0..	0..	0..	0..
altdst	0..	0..	0..	0..	–	0..	0..	–	–	0..	0..	0..	0..
listdst	0..1	0..1	–	–	–	–	–	–	–	–	–	–	–
baddefdst	–	–	–	–	–	–	–	–	–	–	–	–	–
target (Switch)	0..	0..	–	–	–	–	–	–	–	–	–	–	–
target (Command)	0..	0..	–	–	–	–	–	–	–	–	–	–	–
flowdst	0..	0..	0..	0..	–	0..	0..	–	–	0..	0..	0..	0..
flushovr	0..	0..	0..	0..	–	0..	0..	–	–	0..	0..	0..	0..
switchdst	0..	0..	0..	0..	–	0..	0..	–	–	0..	0..	0..	0..
meta	–	–	–	–	–	–	–	1	–	–	–	–	–
origindst	0..	0..	0..	0..	–	0..	0..	–	–	0..	0..	0..	0..
priolo	–	–	–	–	–	–	–	–	0..1	–	–	–	–
priohi	–	–	–	–	–	–	–	–	0..1	–	–	–	–
prioil	–	–	–	–	–	–	–	–	0..1	–	–	–	–
dynid x	–	–	–	–	–	–	–	–	–	–	–	–	–
dynpar0	0..1	0..1	0..1	0..1	–	0..1	0..1	–	–	0..1	0..1	0..1	0..1
dynpar1	0..1	0..1	0..1	0..1	–	0..1	0..1	–	–	0..1	0..1	0..1	0..1
dyntef	–	–	–	–	–	–	–	–	–	–	–	–	–
dynres	–	–	–	–	–	–	–	–	–	–	–	–	–
any edge	0..	0..	0..	0..	0	0..	0..	1	0..1	0..	0..	0..	0..

The upper bounds for the number of in-edges may be incorrect. In theory, there is no upper bound for the incomming edges.
The last row of the table is in most cases the sum of the lower bounds and a sum of the upper bounds. Exception is qinfo, where one of the edge types priolo, priohi, prioil comes in.

6. Node type flush and flow: these nodes may have priority queues (lo, hi, il). Why are the queues allowed? There are no meta nodes generated for these queues. Why?
7. The rules allow two edges from a block to a bufferlist with edge types priolo and priohi. Should the rules be fixed for this invalid schedule?
8. Is an edge of type defdst necessary for a node of type flow? For node type flush it is necessary.
9. There are no ConstellationRules for nodes of type listdst and qbuf. Is this correct? Yes! Node types must be childless.
10. Node type qbuf must be childless.
11. Node type listdst has one defdst to a block. This node type is automatically generated by libcarpedm.

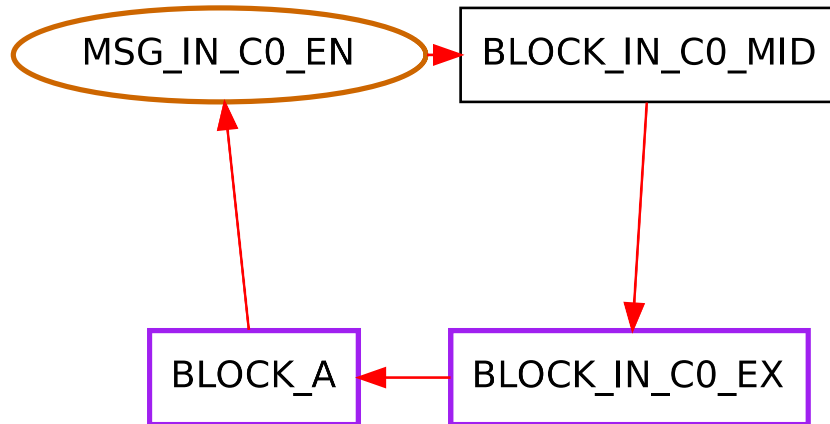


Figure 2.54: Pattern for the dynamic start stop abort test

2.52 test_startStopAbort.py

This test was `full_test/dynamic/basics/start_stop_abort`.

1. Purpose of Test

   First part: Start and abort a pattern. Second part: Start and stop a pattern.

   See Figure 2.54 for the test pattern.

2. Test Actions

   Add a schedule, check that CPU 0 is idle and then start pattern 'IN_C0'. Check that the pattern is running. Abort the pattern 'IN_C0'. Check the visited nodes for the pattern. The second part is similar. Add the same schedule, check that CPU 0 is idle and then start pattern 'IN_C0'. Check that the pattern is running. Stop the pattern 'IN_C0'. Check the visited nodes for the pattern.

3. Success Criteria

   Check that pattern is correctly aborted (rawstatus RUN is 0 immediately) or stopped (rawstatus RUN is 1 immediately, but 0 after 1.5 seconds).

2.53 test_switch.py

This test was `full_test/dynamic/switch`. The schedule `dynamic-switch-schedule.dot` is used.

1. Purpose of Test

   Test the switch command.

2. Test Actions

   After loading the schedule check with 'dm-sched rawvisited' that no node is visited. Then start pattern IN0. Check the visited nodes. Then start pattern IN1 and check the visited nodes. Use a command schedule to switch to destination pattern B and check again the visited nodes.

3. Success Criteria

   The correct nodes are visited after each step.

2.54 test_unilac.py

There are three tests with the same structure. The tests work with message rates of 5 kHz, 45kHz, 90 kHz.

1. Purpose of Test

   This test is used to ensure that the datamaster can handle the amount of timming messages needed to control UNILAC. This should be at least 600 messages in a 20 msec intervall. This is a message rate of 30 kHz.

2. Test Actions The schedule consists of two series of timing messages which are numbered by evtno and parameter. At the end of the first series a flow command directs the flow to the second series of timing messages. The block has a length of 10 msec, so it is executed with 100 Hz.

3. Success Criteria

   Mesages with parameter 1 are received by saft-ctl snoop with 50 Hz.

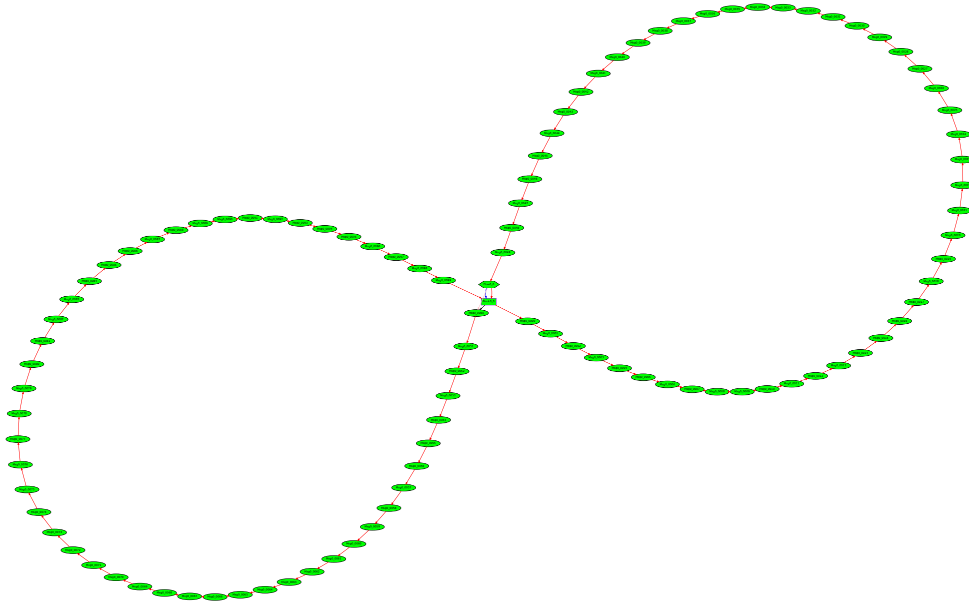


Figure 2.55: Schedule to test the message rate for the UNILAC

2.55 test_waitloopFlush.py

`test_waitloopFlush.py` uses a schedule with a wait loop to check the correctness of the flush command. The schedule contains BLOCK_LOOP with a low prio queue and a high prio queue. On pattern start, a flow command is queue in the low prio queue with quantity 10,000. This redirects the pattern to BLOCK_LOOP such that this block is executed 10,000 times in a loop. Also on start of pattern, a flush command is queued in the high prio queue with a relative valid time of 0.5 seconds. Since BLOCK_LOOP has a period of 100 µs, the flush command breaks the loop after 5011 executions.

The test checks after 0.1 seconds that the flush is not executed, but after 1 second it is executed. The timeline of the checks is not precise enough to check exactly after 0.5 seconds the execution.

2.56 test_zzzFinish.py

`test_pps.py` (pps: pulse per second) is a basic test with a schedule which sends two timing messages every second. The test checks with 'saft-ctl snoop' the timing messages. This test should be the last in the whole test suite to leave the datamaster with a defined schedule.

Chapter 3

Common Components - The Testbench

3.1 dm_testbench.py

`dm_testbench.py` is a collection of Python functions for use in other test scripts.

1. setUpClass(self)

   Read environment variables for the datamaster and the test binaries.

   - TEST_BINARY_DM_CMD sets the binary for dm-cmd. Default is just `dm-cmd`, the installed tool.
   - TEST_BINARY_DM_SCHED sets the binary for dm-sched. Default is just `dm-sched`, the installed tool.
   - DATAMASTER set the datamaster device. Example: `dev/wbm0` or `tcp/fel0069.acc`. There is no default. This method stops with a KeyError if the environment variable DATAMASTER is not set.
   - TEST_SCHEDULES set the folder for schedules. Default is `schedules/`.
   - SNOOP_COMMAND set the snoop command. Default is `saft-ctl tr0 -xv snoop 0 0 0`.

   Usually the binaries for dm-cmd and dm-sched together with the libcarpedm are used from the build of the test repository in folders `modules/ftm/bin` and `modules/ftm/lib`. This is configured in the Makefile.

Set the thread quantity (read from the lm32 firmware with eb-info) and set the CPU quantity to 4.

2. setUp(self)

   Call `self.initDatamaster()`.

3. initDatamaster(self)

   The datamaster is halted, cleared, and statistics is reset with `dm-cmd reset all`.

4. addSchedule(self, scheduleFile)

   Add the `scheduleFile` to the datamaster with the command `dm-sched DATAMASTER add scheduleFile`. Does not start a pattern.

5. startPattern(self, scheduleFile)

   Connect to the given datamaster and load the schedule file (dot format). Search for the first pattern in the datamaster with 'dm-sched' and start it.

6. startPattern(self, scheduleFile, pattern='')

   Add the `scheduleFile` to the datamaster with the command `dm-sched DATAMASTER add scheduleFile`. Start the given pattern. The default for the pattern name is empty.

7. startAllPattern(self, scheduleFile, pattern='', onePattern=False, start=True)

   Connect to the given datamaster and load the schedule file (dot format). If a pattern is given, start this pattern. Otherwise, scan the output of dm-sched for pattern names and start the first pattern (onePattern = True) or all pattern (onePattern = False). All calls to dm-cmd and dm-sched are checked for the return code. The method stops of the return code is not 0.

8. startAndCheckSubprocess(self, argumentsList, expectedReturnCode = [0], linesCout=-1, linesCerr=-1)

   Start a subprocess and check the return code and the lines of stdout and stderr. The output on stdout or stderr is lost.

   (a) argumentsList
       The command itself with arguments as a list.

(b) expectedReturnCode = [0]

The list of expected return codes. The default is 0 which is successful return.

(c) linesCout=-1

The expected lines of stdout. If linesCout = -1 (default), this is not checked.

(d) linesCerr=-1

The expected lines of stderr. If linesCerr = -1 (default), this is not checked.

Examples for calls of this method are in `dm_testbench.py`.

9. startAndGetSubprocessStdout(self, argumentsList, expectedReturnCode = [0], linesCout=-1, linesCerr=-1)

   Start a subprocess and check the return code and the lines of stdout and stderr. Return stdout as a list of lines.

   (a) argumentsList

   The command itself with arguments as a list.

   (b) expectedReturnCode = [0]

   The list of expected return codes. The default is 0 which is successful return.

   (c) linesCout=-1

   The expected lines of stdout. If linesCout = -1 (default), this is not checked.

   (d) linesCerr=-1

   The expected lines of stderr. If linesCerr = -1 (default), this is not checked.

10. startAndGetSubprocessOutput(self, argumentsList, expectedReturnCode = [-1], linesCout=-1, linesCerr=-1)

    Start a subprocess and check the return code and the lines of stdout and stderr. The output on stdout or stderr is return as a list of two lists of lines.

    (a) argumentsList

    The command itself with arguments as a list.

(b) expectedReturnCode = [0]

The list of expected return codes. The default is 0 which is successful return.

(c) linesCout=-1

The expected lines of stdout. If linesCout = -1 (default), this is not checked.

(d) linesCerr=-1

The expected lines of stderr. If linesCerr = -1 (default), this is not checked.

11. removePaintedFlags(self, dotLines)

    From the dotLines remove all which indicates that a node is visited. This ist needed for the comparison with expected output. In the flags attribute, flags="0xnnnnnn1nn" is replaced by flags="0xnnnnnn0nn". In addition, fillcolor green ist replaced by white and other layout attributes are deleted. dotLines is created from 'dm-sched -o output-file'. Method is used by compareExpectedResult and compareExpectedOutput.

12. compareExpectedResult(self, fileCurrent, fileExpected, exclude='')

    Compare two dot files. fileCurrent is the file from the current test run. fileExpected is the file with the expected result. exclude may contain a word. If this word occures in a line of the current file, this line is removed before the comparison. Next step is to remove the 'painted flag' from the current file. Assert that a unified diff has no lines.

13. compareExpectedOutput(self, output, fileExpected, exclude='', excludeField='', delete=[])

    Compare the output from the current test run with the fileExpected which is the file with the expected result. exclude may contain a word. If this word occures in a line of the current file, this line is removed before the comparison. Next step is to remove the lines numbered in the list 'delete' from the current output and the expected file. Next step is to remove the rest of the line when 'excludeField' is found in a line. This is used to remove timestamps from the current output and the expected file. Last step is to remove the 'painted flag' from the current output. Assert that a unified diff has no lines.

14. getSnoopCommand(self, duration)

    Get the snoop command with the duration given in seconds.

15. getResetCommand(self)

    Get the eb-reset command. If eb-reset exists in the repository or workspace, this file name is returned. Otherwise 'eb-reset' is returned, which assumes that eb-reset is installed.

16. snoopToCsv(self, csvFileName, duration=1)

    Run the snoop command for duration seconds and store the output in csvFileName. Since the duration of saft-ctl snoop is measured in seconds, but not fractions of a second, the duration is an integer.

17. snoopToCsvWithAction(self, csvFileName, action, duration=1)

    Run the snoop command for duration seconds and store the output in csvFileName. Since the duration of saft-ctl snoop is measured in seconds, but not fractions of a second, the duration is an integer.

    While snoop runs in a separate thread, the method 'action' is called. This method should return before snoop ends.

18. analyseFrequencyFromCsv(self, csvFileName, column = 20, printTable = True, checkValues = dict())

    Analyse a file output of 'saft-ctl snoop' as a csv file.

    (a) csvFileName
        The file name of the csv file.
    (b) column
        The number of the column to analyse. Default column is 20 (parameter of the timing message). Column for EVTNO is 8.
    (c) printTable
        If True, the occurrences of all values in 'column' are printed.
    (d) checkValues
        checkValues is a dictionary of key-value pairs to check. Key is a value in the column and value is the required frequency. The value can be '>n', '<n', '=n', 'n' (which is the same as '=n'), '=0'. The syntax '<n' fails if there are no occurrences. Checks for intevalls are not possible since checkValues is a dictionary and keys occur at most once. Example: column=8 and checkValues=('0x0001', 62) checks that EVTNO 0x0001 occurs in 62 lines of the file to analyse. Example: column=8 and checkValues=('0x0002', '>0') checks that EVTNO 0x0002 occurs at least once in the file to analyse. Example: column=4 and checkValues='0x7': '=0' checks that FID 0x7 does NOT occur in the file to analyse.

19. analyseDmCmdOutput(self, threadsToCheck=0)

    Run 'dm-cmd' with default action which shows the running threads and the message counts. threadsToCheck is a string of 0 and 1. For 8 threads and 4 CPUs, the string is 32 chars long. For 32 threads and 4 CPUs, the string is 128 chars long. A thread is considered running if the output line for that thread contains 'yes'. A list of (key, message counts) is returned. Keys of this list are numbers xy, where x is the CPU number, y is the thread number, both single digits. If a thread hangs, dm-cmd may show 'yes' for this thread running, but the thread is in undefined status. To avoid this, use the method 'checkRunningThreadsCmd'.

20. checkRunningThreadsCmd(self, messageInterval=1.0)

    Use analyseDmCmdOutput twice with a delay of messageInterval in between. Assert that the message counts increase with the second call and are greater than 0 on the first call.

21. getQuantity(self, line) Get the quantity of command executions from the output line of 'dm-cmd <datamaster> queue -v <block name>'. Search for 'Qty: ' in the line and parse the number.

22. checkQueueFlushed(self, queuesToFlush, blockName, flushPrio, checkFlush=True) Check that a flush command is executed and the defined queues are flushed.

    (a) queuesToFlush: binary value between 0 and 7 for the queues to flush.
    (b) blockName: name of the block with the queues to check.
    (c) flushPrio: priority of the flush command. Defines the queue where to look for the flush command. Priority queues higher than the flushPrio are not affected by the flush command since these queues are empty when a flush command with lower priority is executed. Therefore queuesToFlush is changed for
    (d) flushprio = 0, 1. The check for the executed flush command and the checks for the flushed queues are independant. All checks are done in one parse run of the output of 'dm-cmd <datamaster> -v queue <blockName>'. Flags and counters are used to signal the section inside the output.
    (e) checkFlush = False means: check that no flush is executed.

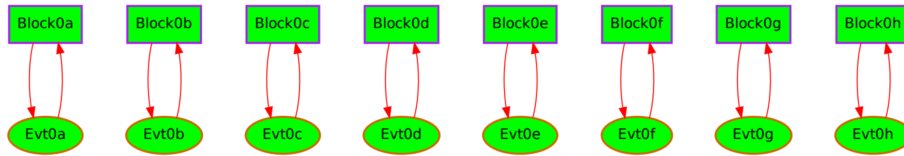


Figure 3.1: Schedule with 8 PPS pattern to run 8 threads on one CPU used in method prepareRunThreads

23. delay(self, duration)

    Sleep for 'duration' seconds. 'duration' is a float.

24. runThreadXCommand(self, cpu, thread, command)

    Test for one CPU and one thread with 'command'. Check the return code of 0 for success and one line of output on stdout.

25. prepareRunThreads(self, cpus=15):

    Check that no thread runs on the CPUs given by 'cpus' (bit mask). Load schedules (see 3.1) into datamaster, one for each CPU and start all threads. Check that these are running. By default cpus=15, all CPUs are used.

26. deleteFile(self, fileName)

    Delete the file with name 'fileName'.

27. resetAllCpus(self):

    Reset each CPU (loop over all lm32 CPUs). Use eb-reset <datamaster> cpureset <cpu>.

28. listFromBits(self, bits, quantity) − > list

    Convert 'bits', given as a string or an int, into a list of int items. quantity is the maximal int + 1.

29. bitCount(self, bits, quantity) − > int

    Count how many bits are 1 in the number 'bits'. This is the number of items enabled in 'bits'.

30. printStdOutStdErr(self, lines)

    Print the lines of stdout and stderr. This is given as a list of two lists of lines.

31. getThreadQuantityFromFirmware(self) − > int

    This class method uses 'eb-info -w' to get the thread quantity from the lm32 firmware. This method is used once in the set up of a class by setUpClass.

32. logToFile(self, text, fileName)

    This class method logs a text to a fileName. The text is prefixed by the current test name. This is appended to the file.

33. checkRunningThreads(self, lines, masks) − > str

    Check the hex numbers describing the running threads. Since in some cases it is not determined which thread is used for a command, we have to check the lines against multiple masks.

    (a) lines is a list of lines, describes the running threads.

    (b) masks is a list (not a set!) of hex numbers as strings.

    Each number describes an allowed pattern of running threads.

3.2 Structure of Description

1. Purpose of Test

   What is the objective of this test?

2. Prerequisites of Test

   What is the setting of the test?

3. Test Actions

   List the actions of the test. This includes the graphs of the test pattern.

4. Success Criteria

   What is checked to state a successful test?