Test Document

Version 1.8, 18 October 2008

Document Number: 07-A072E-07.08

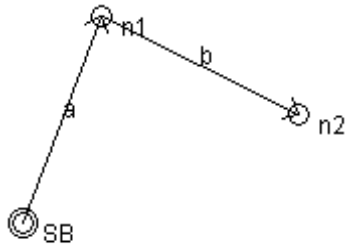
Class LTSNode	
This class represents a single node in an LTS. Currently it is only a wrapper around a string with get/set methods. It is mainly for future use.	
Test Name	***Description***
setGetName	Tests both the setName and getName methods as both are required to work to test just one of them. This test just sets the name to "*Name*" and checks that "*Name*" is returned by getName.
ToString	This method converts the node to its string representation. Currently it does the same as getName and so works in the same way as *setGetName*
Equals	This checks that the equals function returns true for equal nodes and false for unequal nodes.

Class LTSTransition	
This class represents a transition in an LTS. It holds two *LTSNodes* and a string for the label	
Test Name	***Description***
SetGetName	This test is functionally identical to *SetGetName* for *LTSNode*
SetGetNode1	This tests that getNode1 returns what ever was set by setNode1
SetGetNode2	This is functionally identical to *SetGetNode1*
Equals	This tests the custom equals function used to compare LTSTransitions. It creates two transitions with the same content (same labels, node names, etc) and checks they are equal. It then changes one of the transitions and ensures the equals function is correct in identifying them as unequal.
ToString	Constructs a transition and checks to see if the result of the toString method is valid for that transition. As some transitions may have a null value for their second node the test sets the seocnd node name to null and checks that the result is still correct. The LTSTransition class may have null as its second node when it is really representing an isolated node with no transitions.

Class LTS	
This class, being the largest and most complex class, has quite a few tests. Because many of the tests require a great deal of data to operate the test loads their data from a file allowing the load function to do all the hard work. Creating the internal data structures manually would take a great deal of time. Because of this dependence on the load function, if the load function fails its tests many other functions will also fail their tests because the data used in the test cant be loaded. If the load function fails it should be the first one to be fixed – any other failures may be a result of it. The data files are created in the tests setUp function and are removed in the tearDown function. Should the tearDown function fail to execute these data files may be left over in the current working directory.	
Test Name	***Description***
Load	This test loads a plain-text file representing an LTS and checks that the load function created the appropriate data structures. It also checks that the LTS class correctly fails to load invalid LTS files. These contain unquoted symbols " " (space), "-", "=" and "|".
Save	This test attempts to • Load an LTS, change it and save it with a specified filename. • Open the saved file for input to check that the save was successful • Modify the LTS by adding an isolated node transition (one with no label and a second node of null) • Overwrite the saved file with the new LTS • Load the LTS and ensure that the isolated node was saved correctly.
Composition	There are four composition tests which ensure parallel composition functions correctly. Each one of these tests implements one of the example LTS on the second page of the assignment 1 sheet. The last test includes a synchronization set. When one of these tests fail the contents of the calculated LTS and the correct LTS are dumped to the system console for debugging purposes. The first three tests which do not involve synchronization also test that Parallel Reach is able to do unsynchronized composition fine. For a visual guide of the composition tests see the last section.
testReach	This test checks that Parallel Reach works properly for synchronization (unsynchronized tests are done at the same time as the compose tests). This test is the same as the 4th composition test but with the isolated transitions removed.
Equals	This test constructs two equal LTS and checks that the equals() function claims they are equal. One of them is then modified so they are no longer equal. Their equality is then checked again to ensure

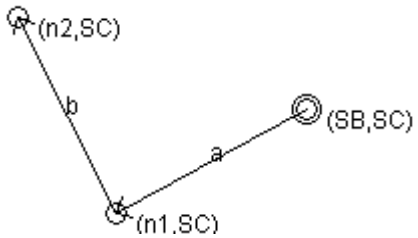
	that the equals() function can properly identify LTS that aren't equal.
getFileName	This test checks that the filename is always correctly returned after a file load and a save with a different file name.
SetGetStartNode	This tests the setStartNode getStartNode functions. When operating correctly the getStartNode method should always return what ever was passed into the setStartNode function
GetNodes	This test creates a few transitions containing 6 nodes of which only 2 are unique. The getNodes function should only return the 2 unique nodes, not 3 of each. A null node is then added and the node count is checked to ensure the null node is not returned. If the start node is not a member of a transition it will not be returned by getNodes(). To get the start node in this list even if it isn't a member of a transition, getNodes(true) should be used. Tests for this functionality are implemented.
TransitionOps	This checks the various functions that operate on transitions. This includes: • void addTransition(LTSTransition) • boolean transitionExists(LTSTransition) • Collectin<LTSTransition> getTransitions() • void removeTransition(LTSTransition) • void killTransitions() These are all implemented in one test because they mostly rely on each other to be tested. It does the following 1. Adds 2 transitions 2. Checks that one of the transitions added is present 3. gets a list of all transitions 4. checks that there are 2 transitions present in the list 5. checks that one of the added transitions is actually in the list 6. removes a transition 7. checks that transitionExists no longer claims the transition is present 8. Adds another transition 9. Updates the transition list 10. Attempts to remove all transitions (killTransitions function) 11. gets a list of transitions and checks that it is empty 12. checks that the list of transitions fetched in step 9 is *not* empty (if the getTransitions() function doesn't clone the list clearing the original list would end up clearing all lists returned by getTransitions())

CompositionPass1 test



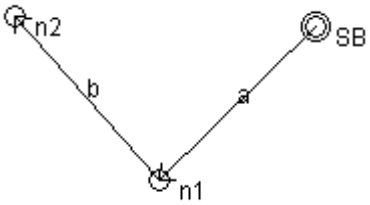
LTS A

LTS B

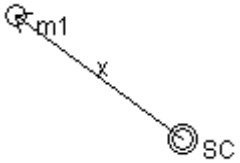


Expected Result

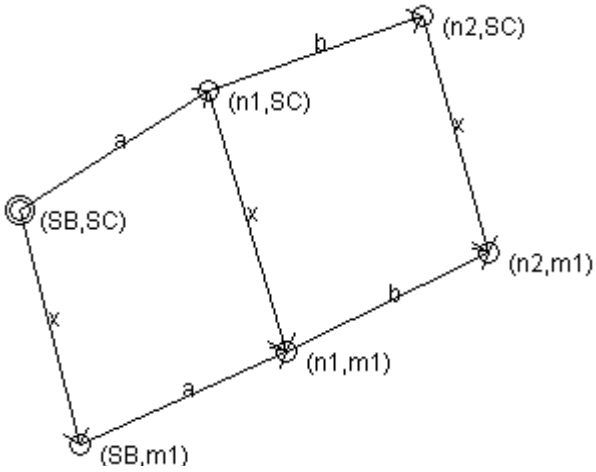
CompositionPass2 test



LTS A

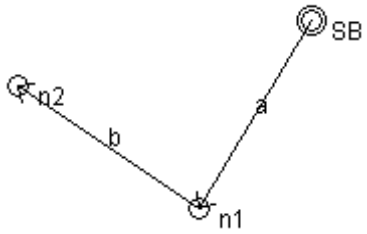


LTS B

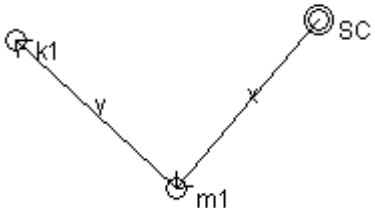


Expected Result

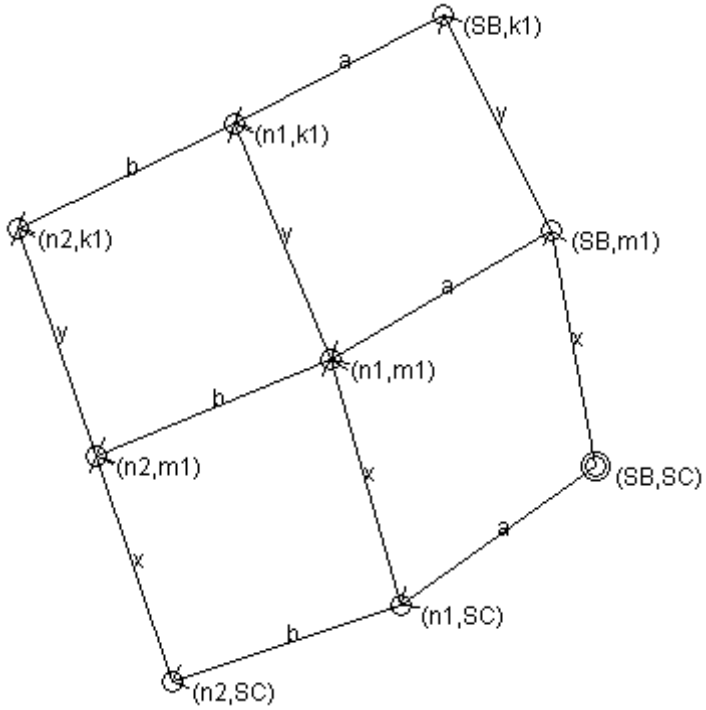
CompositionPass3 test



LTS A

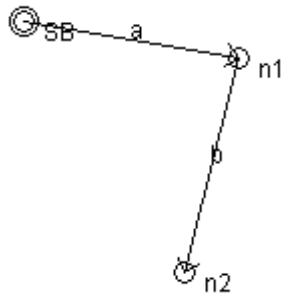


LTS B

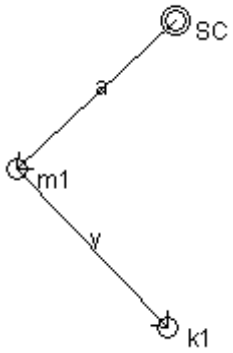


Expected Result

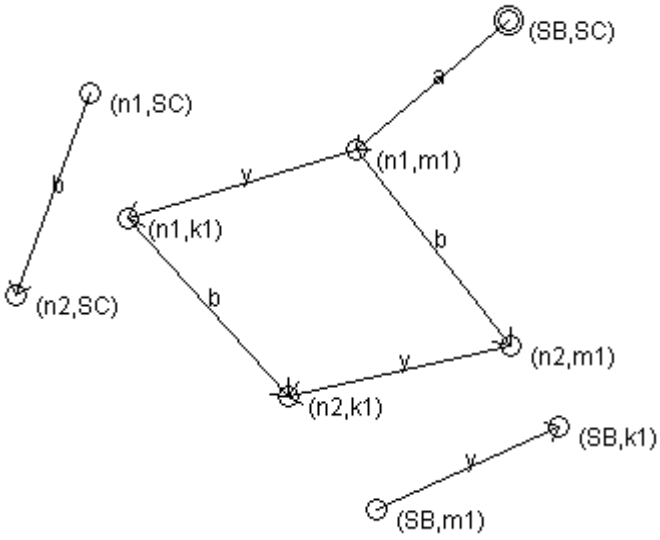
CompositionPass4 test



LTS A

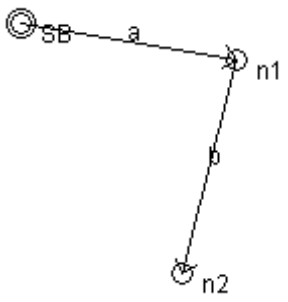


LTS B

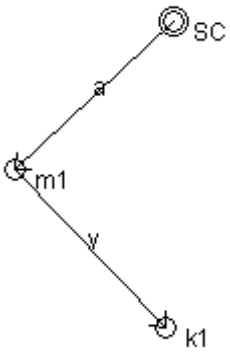


Expected Result

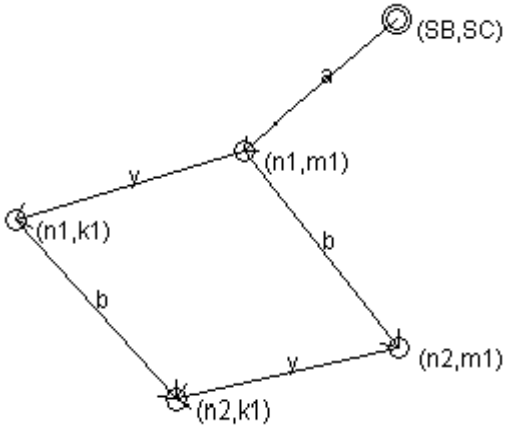
Reach test



LTS A



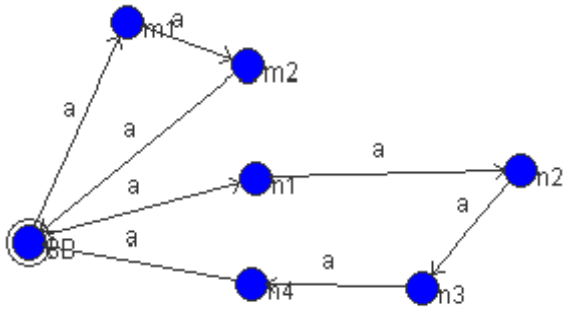
LTS B



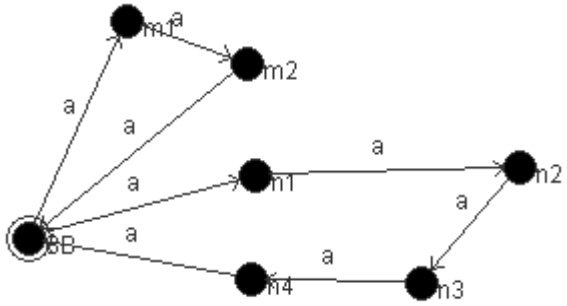
Expected result

Coloring 1 test

From assignment 3, figure 3



LTS A

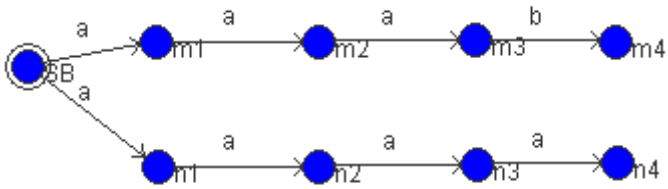


Expected Result

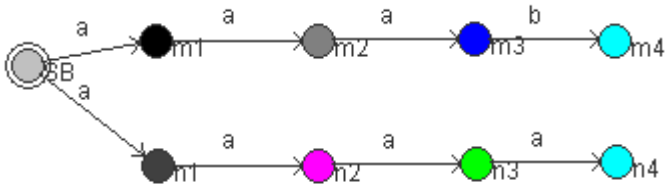
All nodes should have the same color.

Coloring 2 test

From assignment 3, figure 2



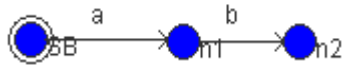
LTS A



Expected Result

Only nodes n4 and m4 should have the same color.

Bisimulation test 1



LTS A & B

As both LTS are identical the Bisimulation Equality function should consider them equivalent.

Bisimulation test 2

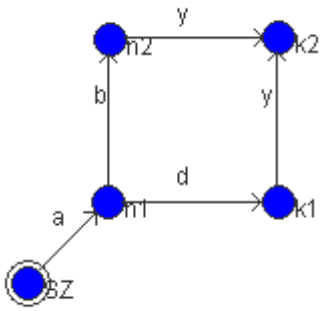
LTS A

LTS B

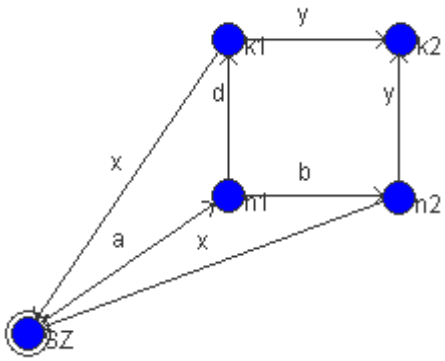
The Bisimulation equality function should not consider these two LTS to be equal or equivalent.

Bisimulation test 3

From assignment 3, figure 1



LTS A

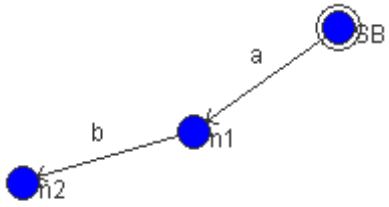


LTS B

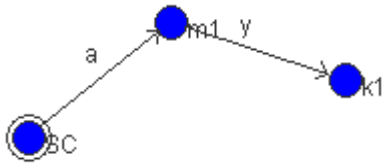
The bisimulation equality function should consider these LTS to be equivalent.

BisimCompose test

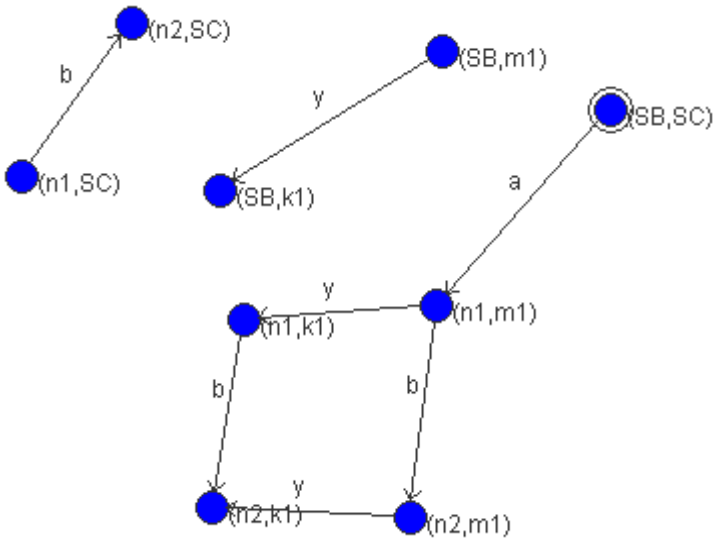
This test attempts to check that the LTS produced by Parallel Compose and Parallel Reach are equivalent. This is achieved by comparing the result of both LTS.



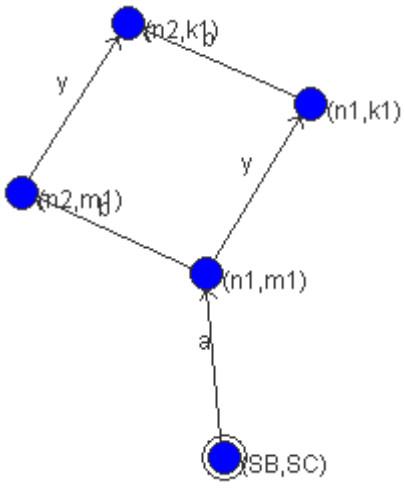
Input LTS A



Input LTS B



Parallel Compose result



Parallel Reach result

The results from parallel compose and parallel reach should be equivalent.