Automata and Behavioral Equivalences

(Manual Guide)

Gomez, Renz Rallion T.
Bermudez, Christopher M.
Cachero, Villy Kayle D.

02/15/2020
CCS23 - 2H (TF 5:30-7:30)

Week 1 - Introduction to Automata Theory

Terminologies

Automata Theory - The study of abstract machines and automata, as well as the computational problems that can be solved using them. It is a theory in theoretical computer science and discrete mathematics (a subject of study in both mathematics and computer science). The word automata (the plural of automaton) comes from the Greek word αὐτόματα, which means "self-making".

Input - A string fed to a machine which the machine will determine whether it is part of the language that the machine was designed for.

Return/Output - The results of running the machine on a given input. Initially, this will either be accepted or rejected, indicating whether the input string is respectively part of the language or not

State - A resting place while the machine reads more input, if more input is available. States are typically named Canonical names for states (which we will get to later) commonly consist of the lowercase letter 'q' followed by a number.

Start/Initial State - this is known as the *program entry point*. It is the state that the machine naturally starts in before it reads any input**.**

Accepting/Final State - A set of states which the machine may halt in, provided it has no input left, in order to accept the string as part of the language.

Rejecting/Trap State - Any state in the machine which is not denoted as an accepting state. The string is only rejected if the machine halts in a rejecting state with no more input left.

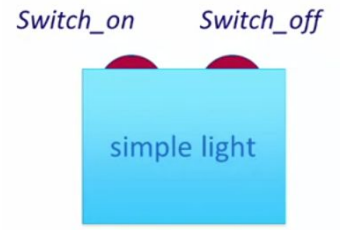
Deadlock State - An unmarked state where no events are possible. A rejecting state that is essentially a dead end. Once the machine enters a dead state, there is no way for it to reach an accepting state, so we already know that the string is going to be rejected.

Transition - A transition is represented as an arrow pointing from one state to another, labelled with the symbol or symbols that it can read in order to move the machine from the state at the tail of the arrow to the tip of the arrow. Transitions may even point back to the same state that they came from, which is called a loop.

Finite State Machine - A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, or simply a state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time.

Transition System - A transition system or transition graph is a finite directed labelled graph in which each vertex represents a state and the directed edges indicate of a state and the edges are labelled with input/output. a transition system is a concept used in the study of computation. It is used to describe the potential behavior of discrete systems.

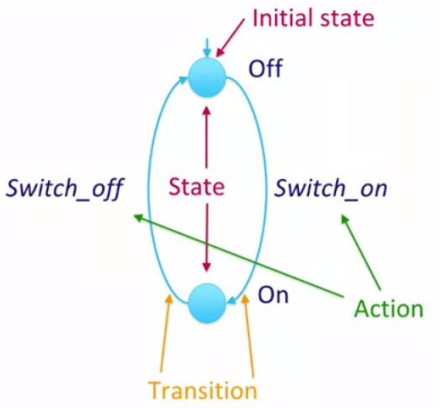
Systems are always interacting with each other, input comes and out comes for short it goes all together. - *Robert Milner*



Examples of System Behavior: (Simple Light)

A simple light can be switched on and switched off, and there are two actions namely switch_on and switch_off.

Modeling a system's behavior mathematically *(Simple Light)*

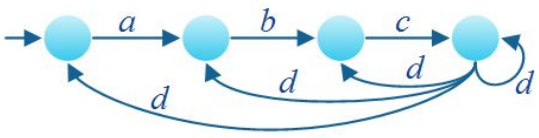


Take all actions of the system or behavior and reconsider them as being atomic.

State - Light
Actions - State on or State off. = labeled transition system or state machine or an automata.
Transitions - State on or off is a trace of the system, *Trace* is simply switch sequence of actions and that shows what you can do with the system.
Initial State - It is where the whole system starts

Non - deterministic behavior

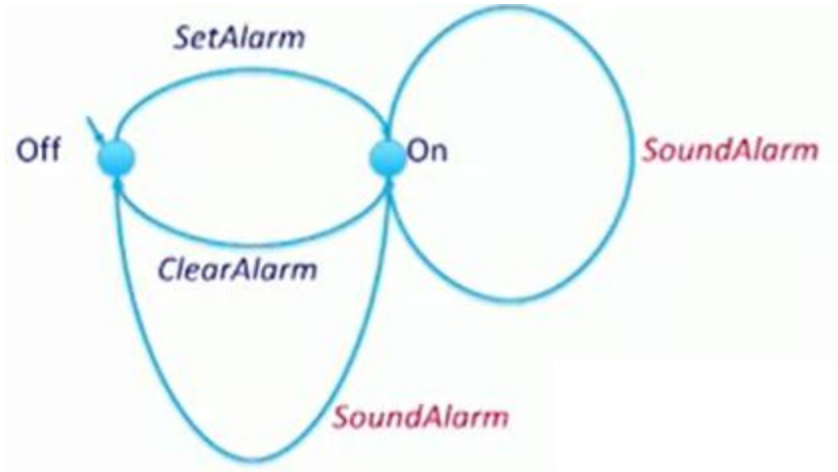
A *State* is *non-deterministic* if and only if it has at least two outgoing transitions with the same label.
A labelled transition system is non-deterministic if it has at least one non-deterministic state.
Example of Non-deterministic behavior:



The Automaton:
A finite automaton is represented formally by a 5-tuple <states, actions, transitions, initial state, terminating state> or it is represented by:

- S is a set of *states*.
- Act is a set of actions, possibly multi-actions.
- $\longrightarrow \subseteq S \times Act \times S$ is a *transition relation*.
- $s \in S$ is the *initial state*.
- $T \subseteq S$ is the set of *terminating states*.

This is called the Labeled Transition System (LTS)
Example:

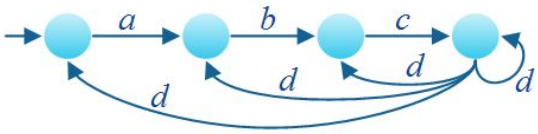


X = {{Off, On}, > States
{SetAlarm, ClearAlarm, SoundAlarm}, > Actions
{Off -> SetAlarm -> On, > Transitions
Off -> ClearAlarm -> On,
On -> SoundAlarm -> On,
On -> SoundAlarm -> Off},
Off, > Terminating State
{Off}} > Initial State

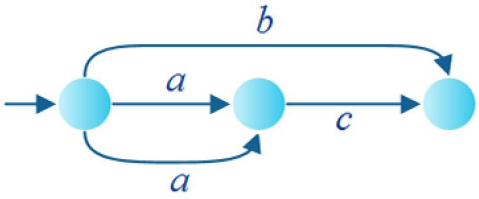
Activities:

1. Which of the following labelled transition systems is non-deterministic?

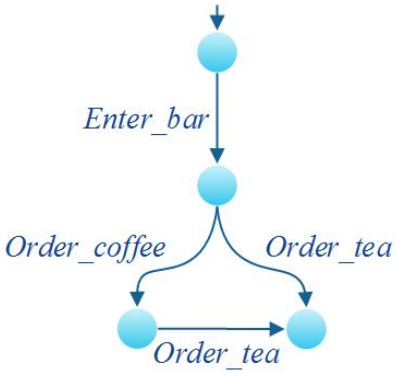
A.



B.



C.



2. Must a transition system always have a finite number of states? Why?

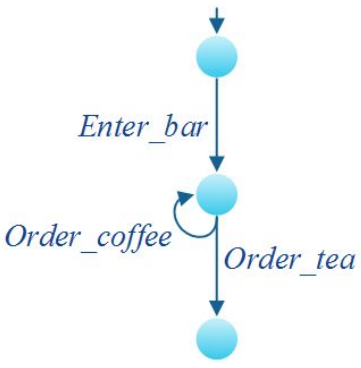
3. Which is not the wrong sequence of the 5 tuple labeled transition system?
   A. X = (s, Actions, ----->, S, T)
   B. X = (Initial State, Actions, transitions, States, Terminating States)
   C. X = (S, Actions, ----->, s, T)
   D. X = (Terminating States, Actions, transitions, States, Initial State)

4. What is a Deadlock State in Automata?
   A. It is a state that has been quarantined for some cases
   B. An unmarked state where no events are possible
   C. It is a condition where 2 or more processes are waiting for another to release a resource
   D. An marked state where no events are possible

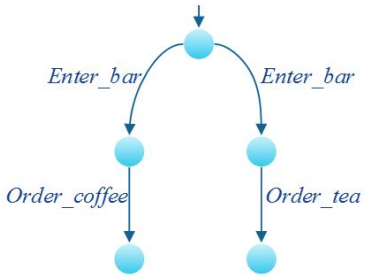
5. Select the transition systems that correctly describes the process of entering a bar where it is only possible to order coffee first and then repeatedly order tea followed by coffee, followed by tea, etc.

A. None of the Above

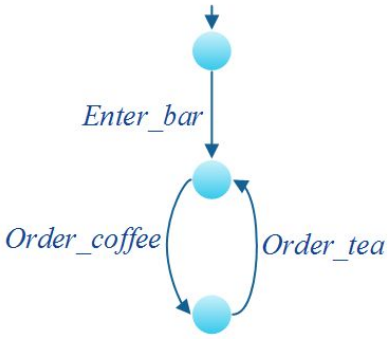
B.



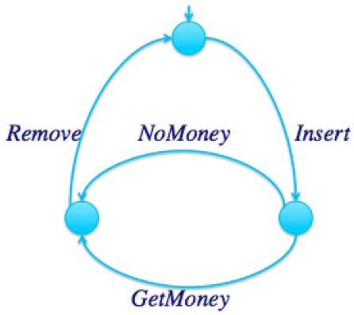
C.



D.

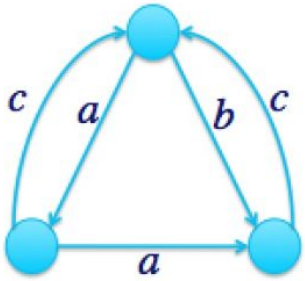


6. What is the correct trace of this system behavior



A. Remove, Insert, GetMoney, Remove, Insert, GetMoney, Remove, Insert, NoMoney
B. No Money, Remove, Insert, GetMoney, Remove
C. Insert, GetMoney, Remove, Insert, GetMoney, Remove. Insert, No Money, …
D. None of the Above

7. Which of the following are mostly true

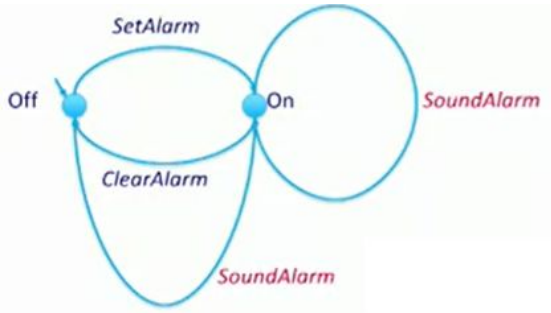


A. This is an example of a deterministic transition system
B. This is an example of a non-deterministic transition system.
C. This is neither deterministic nor non-deterministic transition system
D. This is not a valid transition system.

8. What is the correct definition of a transition system?

A. A transition system not used in the study of computation
B. A transition system is used to describe the potential behavior of discrete systems
C. A transition system is useful in coding
D. None of all the choices are

9. What is the correct Labeled Transition System (LTS) of this graph:



A. x = {{Off, On},{SetAlarm, ClearAlarm, SoundAlarm},{Off -> SetAlarm -> On,Off -> ClearAlarm -> On,On -> SoundAlarm -> On,On -> SoundAlarm -> Off},Off,{Off}}

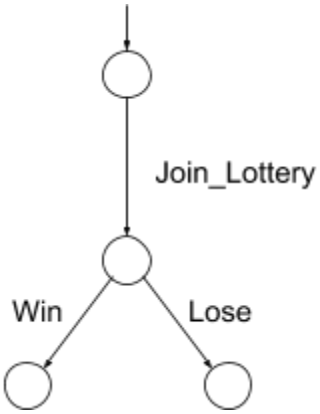
B. x = {{SetAlarm, ClearAlarm, SoundAlarm},{Off, On},{Off -> SetAlarm -> On,Off -> ClearAlarm -> On,On -> SoundAlarm -> On,On -> SoundAlarm -> Off},{Off},Off}

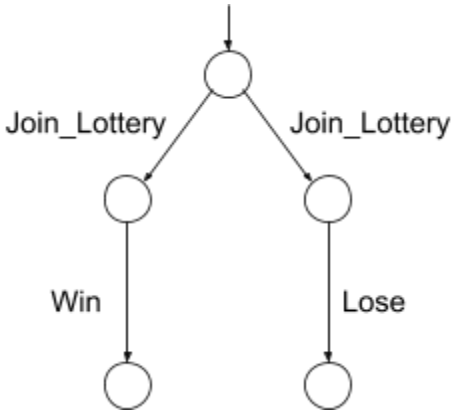
Activities Key Answers:

1. A
2. No, because a set of actions can also be finite or infinite.
3. C
4. B
5. D
6. C
7. A
8. B
9. A

Week 2 - Behavioral Equivalences pt. 1

Some people believe that we make our destinies as we go along. Some believe that our destinies have been written from the moment we are born. I say, what is the difference? I say, how can something so abstract can even be proven when there are infinite destinies we can come up with? We are intrigued everyday to predict rather unexpected events. And how we define and react to these events is even more interesting.





But let's say that we play god and we command lives to show us their fates. We dress up as fortune tellers to help people decide on their actions based on the corresponding consequences. How can we tell people their future? And how can we advise them on which course of life to take?

Behavioral Equivalence - describes the labelled transition systems (LTS) to have the same possible sequences of observable actions.

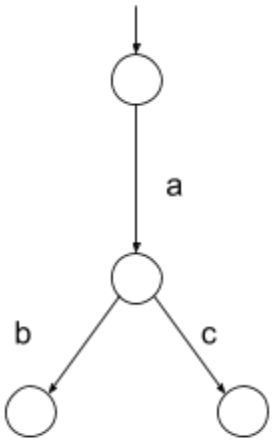
LTS may have finite or infinite sequences depending on the scenario.

Main Types of Equivalences

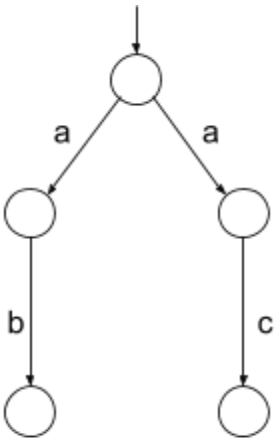
- Trace Equivalence
- Bisimulation Equivalence

Trace Equivalence

Robin Milner defines Trace Equivalence as when LTS have the same Trace Language such that for every LTS (S, s0, L, T) its Trace Language T is the set of sequences.



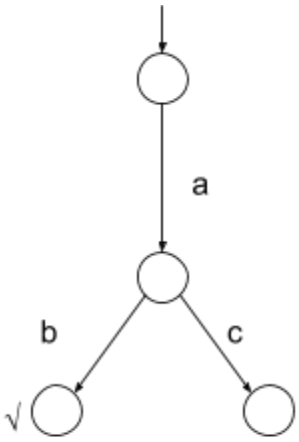
{ϵ, a, ab, ac}



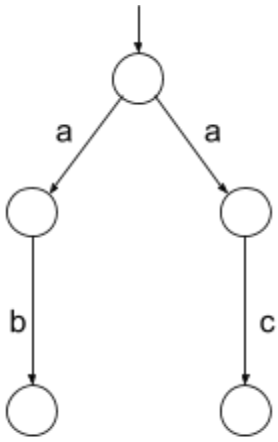
{ϵ, a, ab, ac}

The illustration above shows two LTS, the left and the right. In terms of visual representation, they are not equal. But the Trace Languages (sets of sequences located below the LTS) are found to be equal therefore describing these LTS as Trace Equivalent.

Any difference in Trace Languages will result in Trace Inequivalence.



{ϵ, a, ab, ab√, ac}



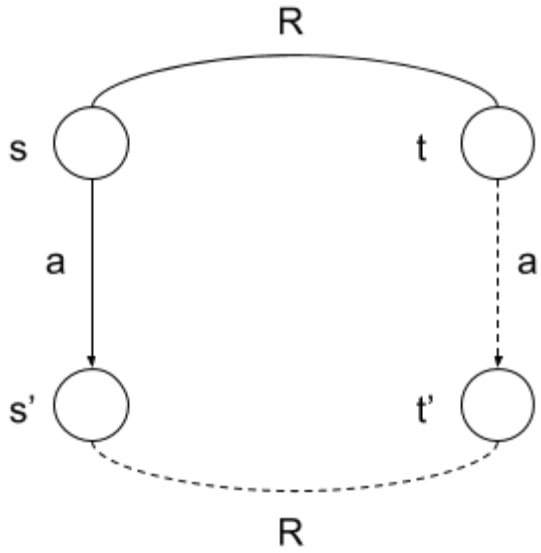
{ϵ, a, ab, ac}

The illustration above shows a termination after action b was made in the left LTS. A minor detail added but still resulted in Trace Inequivalence due to the requirement equal Trace Languages not being met.

This implies that context and our actions are more important when determining our destinies rather than our status. No matter if we are rich or not, we can only make a difference through our actions.

Bisimulation Equivalence

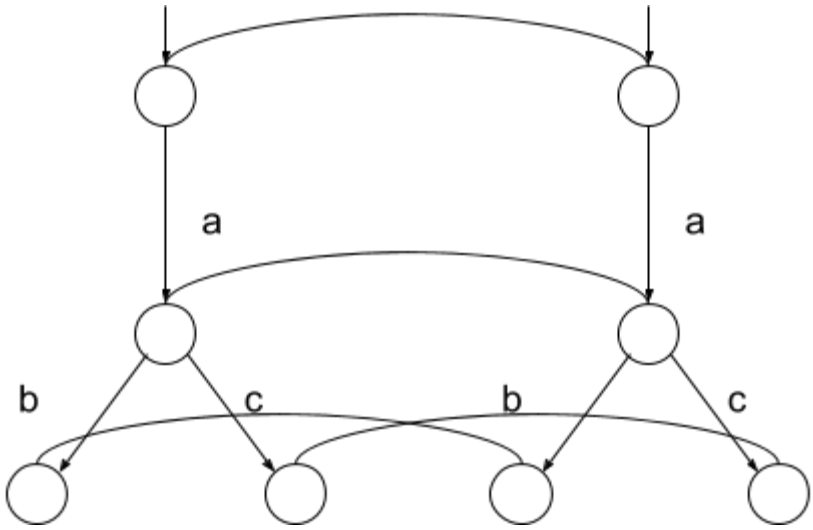
According to Robin Milner, LTS are bisimilar equivalent when non-distinguishable states are observed as having two states that are equivalent if for all possible transitions labelled by the same action, there exist equivalent resulting states.



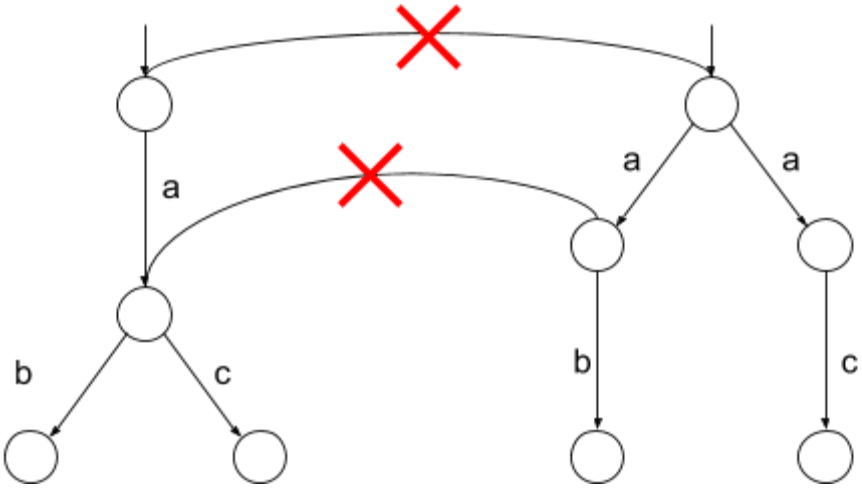
Bisimulation Equivalence states that if sRt (s relates t) and s --^{a}> s', then t --^{a}> t' and s'Rt' such that s, t are elements of set S.

The illustration on the left shows state s is being mimicked by state t therefore resulting in Bisimulation Equivalence. State s has action a that leads to state s'. Since s is related to t, t mimics the same action a that leads to state t'. In this situation, we can infer that s' and t' are related. The scenario fits the description of Bisimulation Equivalence hence we can conclude that they are bisimilar equivalent.

This is suggested by most experts because it preserves the behavioral properties and makes it easier to measure in terms of O (Big O) notation unlike Trace equivalence which has no clear way of calculating O notation.

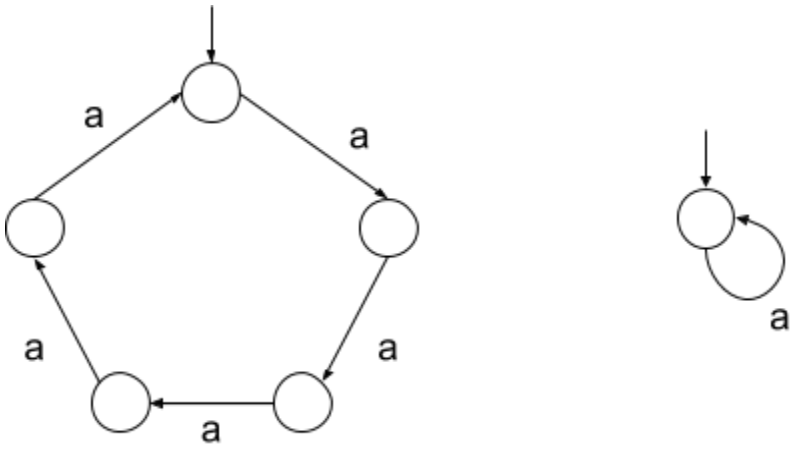


The illustration above shows strong bisimilar equivalent LTS since every possible transitions are labelled by the same action for all corresponding states.



The illustration above shows bisimulation inequivalence since the action a on the left LTS led to a state wherein it has two possible transitions by taking actions b or c which is not possible on the right LTS. If the two states infer bisimulation inequivalence, then their source states are also not bismilar equivalent.

There exists a theorem stating that every transition system has a unique minimal transition system that is bisimilar equivalent to it as shown below.

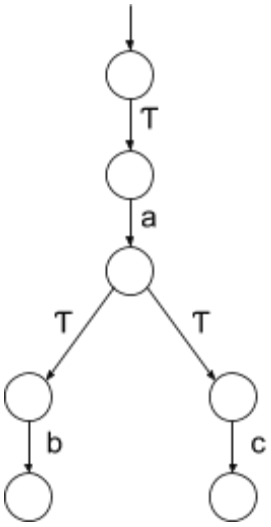


This implies that we may be faced with situations wherein we can just live it to fate to decide for us and hope for the best. Regardless, we should not be afraid to fail and rather accept it since it is the best way to learn.

Others may take a long time to be successful and achieve their goals and others might get it the first time. We just need to be patient and take our own pace.

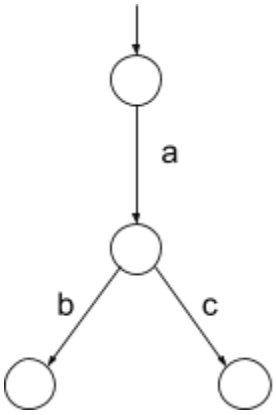
Internal Actions

LTS may contain several hidden actions and they are defined as T actions. These are non-observable actions that may be done before achieving a state.

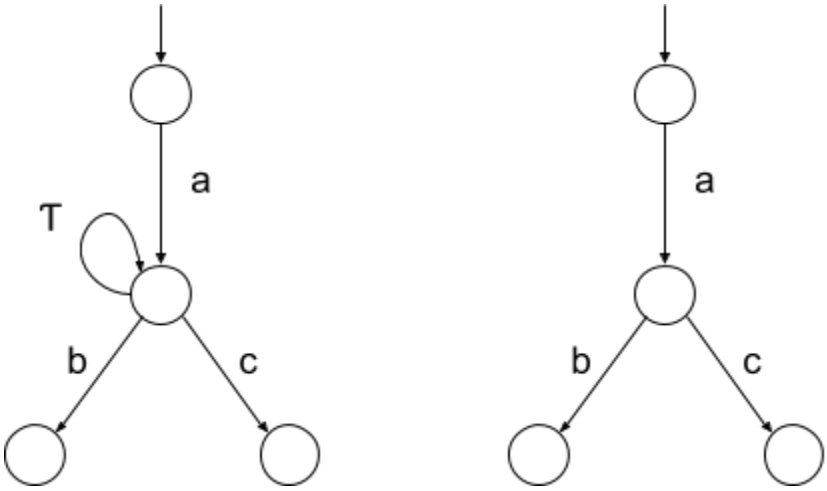


The illustration on the left shows a LTS with visible T actions. On some LTS, we can safely remove T actions without compromising the behavioral properties of the original one.

The illustration on the right shows a LTS that is somewhat similar to the left one but not really. If we tried to analyze the left LTS, we can see that we can go to a state wherein our only choice is b action. But the right LTS does not have that state.



The unclear clear explanation regarding when the T actions could be removed will be that it depends whether it affects the behavioral properties of the LTS or not. The illustration below shows a clearer explanation of the matter.

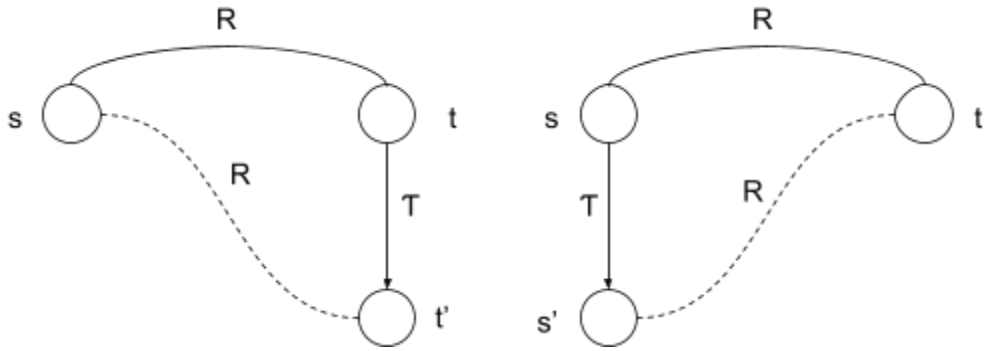


If we remove the internal action T on the left LTS, the resulting LTS will be the one on the right. The behavioral properties remain unchanged because states still have the same possible transitions on the corresponding actions.

This implies that some may be making extra effort in secret to guarantee the achievement of their goals and some do not. It is up to us if we are willing to do those extra efforts that lead to our success.

Branching Bisimulation Equivalence

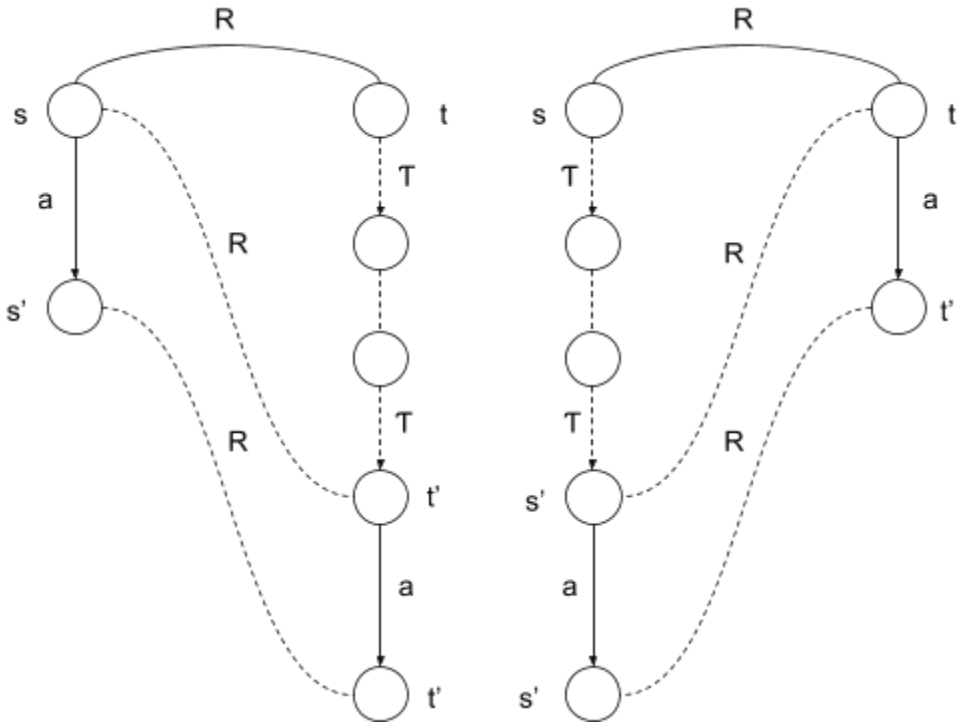
Introducing internal actions in Bisimulation Equivalence, we get Branching Bisimulation Equivalence. If states s and t are related, then internal actions leading to s' and t' states will lead to relations sRt' and s'Rt.



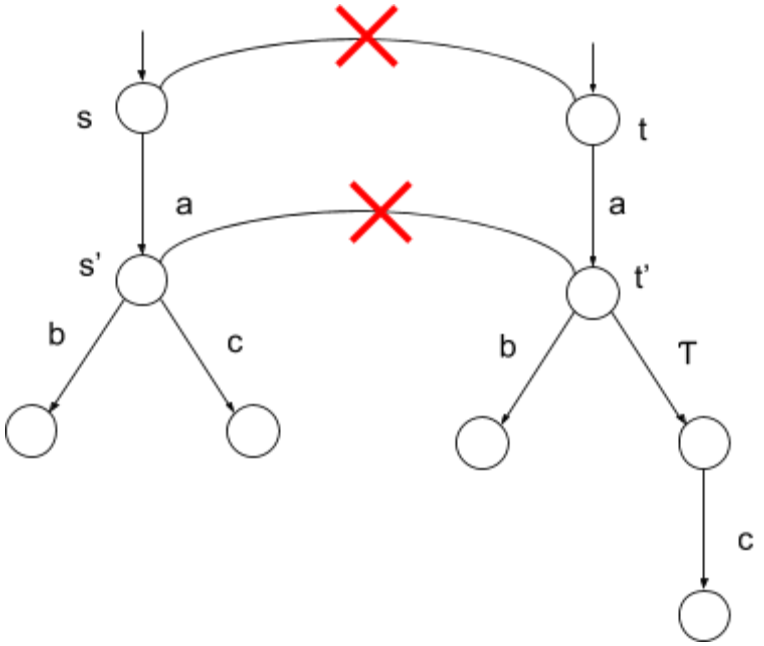
Branching Bisimulation Equivalence states that if s --^{a}> s' for s' is an element of a set S,

- then a = T and s'Rt,
- or t',t'' are elements of set S such that t --^{T}> ... --^{T}> t'--^{a}> t'', sRt' and s'Rt''.

The second item is shown as when a state takes action a and mimicked by the other state as shown below.



However, on some LTS with internal actions but closely similar structure, branching bisimulation equivalence is not possible.

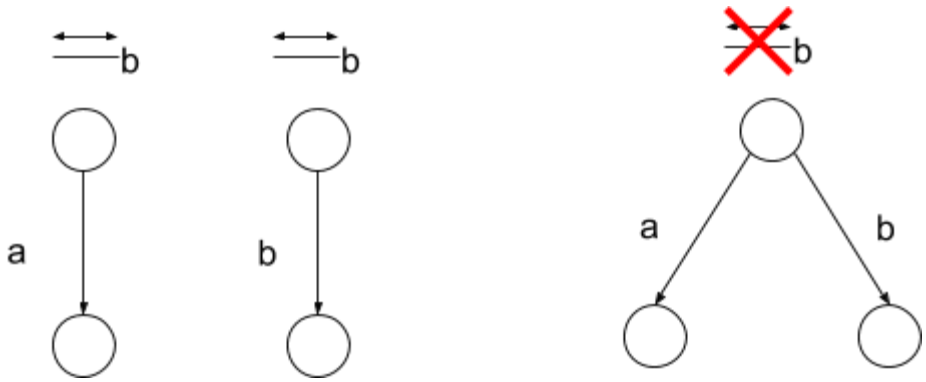


The illustration above shows that internal actions can cause Branching Bisimulation Inequivalence by changing the behavioral properties of the LTS. In the left LTS, the state s' can have two options namely the b and c options. But on the right LTS, state t' has options action b and internal action T.

This implies that we shouldn't compare ourselves to other people based on our status. As explained earlier, they may be making extra efforts in making sure that they achieve their goals or they don't. Regardless, it just goes to show that our individual efforts may or may not result in different choices in life.

Rooted Branching Bisimulation Equivalence

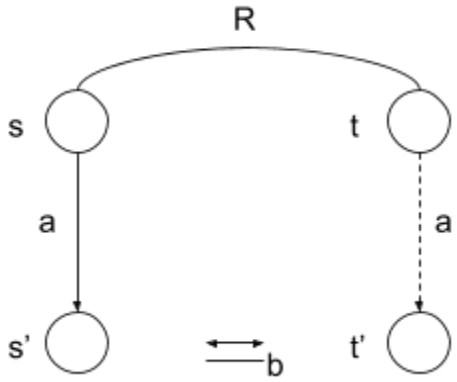
As discussed earlier, when combining behaviors, internal actions may or may not change the behavioral properties. In times that internal actions change behavioral properties, Branching bisimulation may not apply in these circumstances. Hence, Rooted Branching Bisimulation Equivalence was introduced to address this issue.



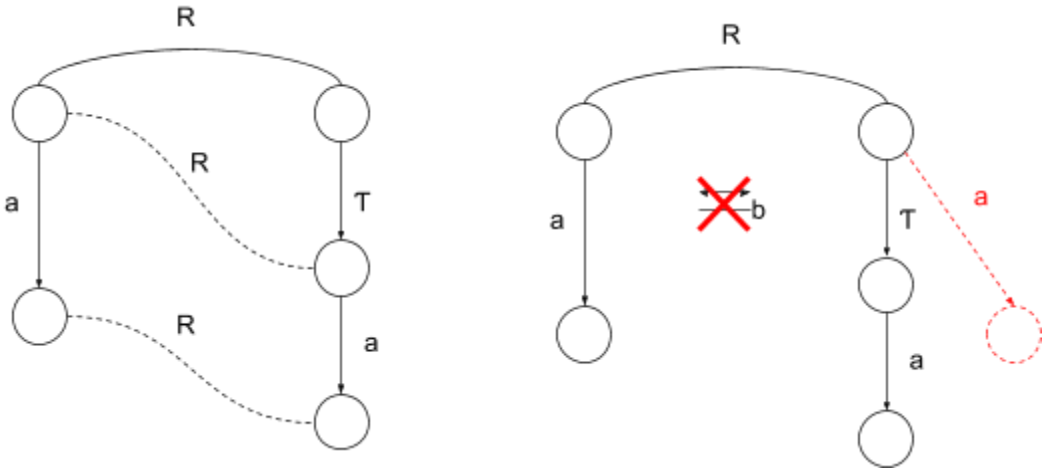
The illustration above shows how behaviors are equivalent when separated but not when combined. Relating the states together is possible under conditions that follow:

- If s --^{a}> s' for s' is an element of a set S, then t --^{a}> t' and s' <--$_{b}$> t' for t' is an element of a set S.

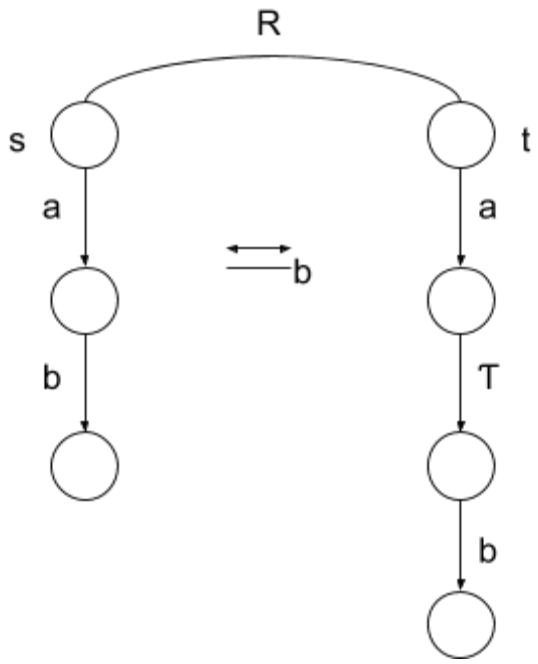
Like the Bisimulation Equivalence principle, a Rooted Branching Bisimilar Equivalent states' initial states should also be Rooted Branching Bisimilar Equivalent.



The illustration above shows that the state s is related to state t. State s has option for action a that leads to state s' and this was mimicked by state t. State t has action a that leads to state t'. Since it satisfies the principles of Rooted Branching Bisimulation Equivalence, we can conclude that the two states s' and t' are Rooted Branching Bisimilar Equivalent (s' <--$_{b}$> t').



The illustration above shows a situation where an internal action T changes the behavioral properties of Branching Bisimilar Equivalent states. This situation resulted in Rooted Branching Bisimilar Inequivalent states as the state on the right is missing an action a and instead has an internal action T.



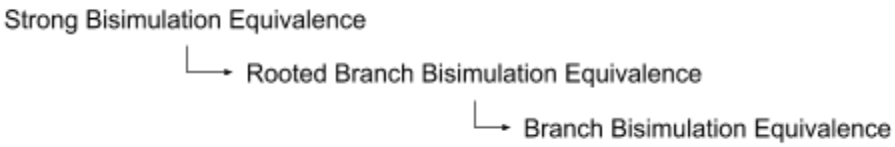
The illustration on the left shows a Rooted Branching Bisimulation Equivalence because the conditions are met.

State s --^{a}> s' and t --^{a}> t' therefore s' <--$_{b}$> t' for t' is an element of a set S.

The initial states of the Rooted Branching Bisimilar Equivalent states are also Rooted Branching Bisimilar Equivalent states.

Since the next states are not connected via actions of the Rooted Branching Bisimilar Equivalent states, determining if they are Rooted Branching Bisimilar Equivalent states is not necessary. Regardless they are still Branching Bisimilar Equivalent states since it means that they are automatically if they are Rooted Branching Bisimilar Equivalent states.

The precedence of Bisimilar Equivalence is described as Strong Bisimulation Equivalence results to Rooted Branching Bisimulation Equivalence which then results to Branching Bisimulation Equivalence as shown in the illustration below.

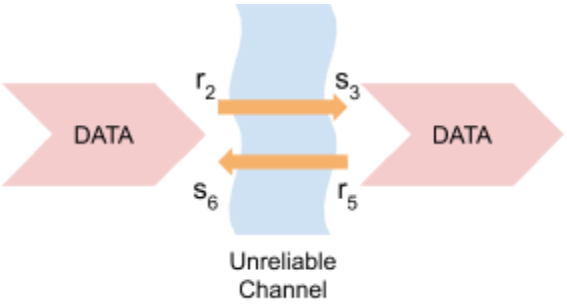


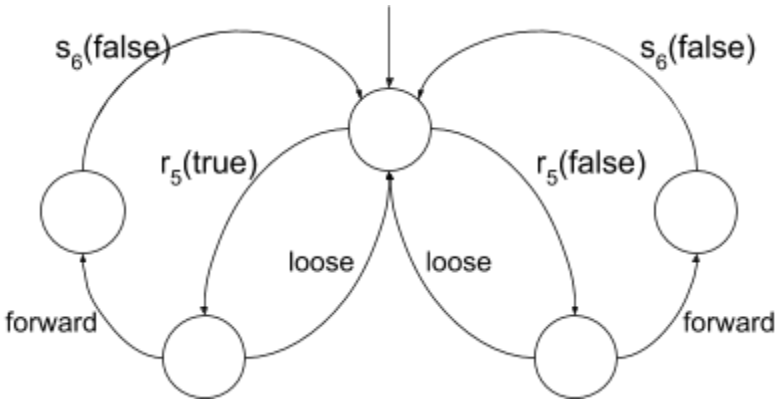
This implies that our destinies start from the same places. The events in our lives may still be the same with others and some might not. But in order to secure these achievements, we must decide and take on the actions that lead to what we want our destiny to be.

Week 3 - Behavioral Equivalences pt. 2

Alternating Bit Protocol

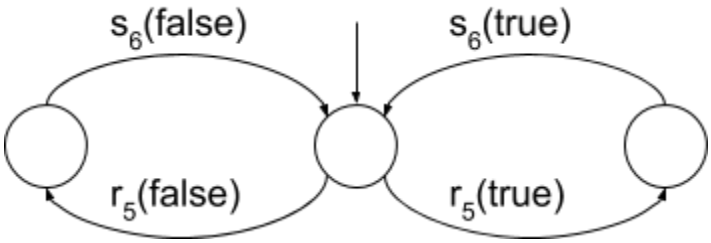
When dealing with data transfer electronically, it is always a challenge to keep that data as complete as possible since, ironically, we still rely on unreliable transfer channels.





However, to ensure that data is transferred completely and properly, we use iterations of a two-way communication system that sends the data and verifies if it has been received.
The Alternating Bit Protocol is used in these types of data transfer. It can be represented as an automaton as seen on the illustration on the left.

As discussed in the previous lesson, every labelled transition systems (LTS) have a unique minimal LTS that is bisimilar equivalent to it. Hence, simplifying the automaton above, gives us the said unique minimal LTS illustration on the right.

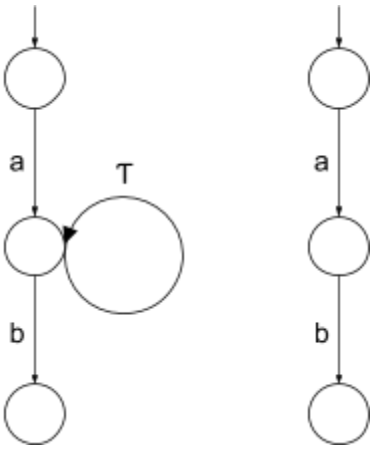


Note: Somehow, TCP (Transmission Control Protocol) in networking uses this type of acknowledgement system that results in a more reliable data transfer but often compromising transfer speed. On the other hand, UDP (User Data Protocol) does not require acknowledgements when transferring data but rather it sends data continuously. This results in better transfer speed but unreliable data transfer.

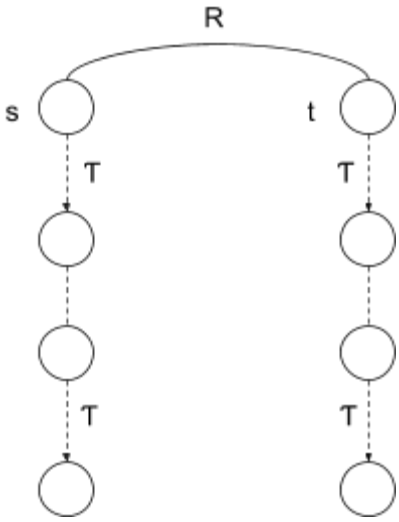
Divergence Preserving Branching Bisimulation

Divergence is defined as the infinite sequence of τ actions in a given LTS. Branching Bisimulation Equivalence makes it so that internal actions τ are insignificant when identifying behavioral properties. Hence, Branching Bisimulation Equivalence removes Divergencies.

The problem with this is that the resulting LTS are not equivalent when a divergence is present on one and not in the other.



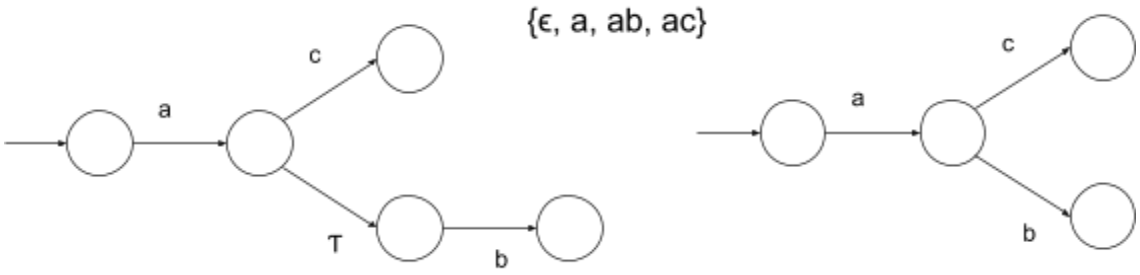
The illustration above shows Branching Bisimilar Equivalent states but one has its T actions removed. The difference between the two is that the left one can have an infinite sequence of T actions or a divergence but the right one does not have this infinite sequence but rather just a finite one.



Divergence Preserving Branching Bisimulation Equivalence resolves this issue by adding one condition in the Branching Bisimulation Equivalence which is that if state s can have an infinite number of T actions, then state t should also be able to mimic this and have an infinite number of T actions as shown below. This ensures that if Divergence is necessary in a LTS, it will be retained and not removed.

Weak Trace Equivalence

A Trace Equivalence is considered Weak if and only if internal actions T are omitted in the set of sequence S.



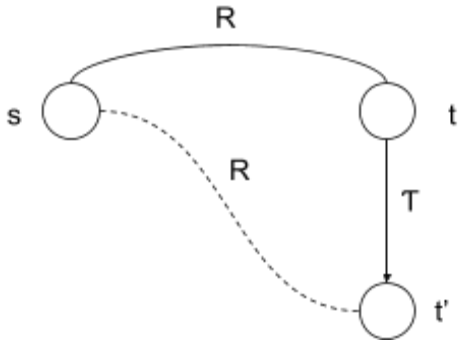
This allows the removal of T actions in the LTS since the set of sequence S will be the same. It can be useful in small-scale LTS but difficult to calculate in larger ones.

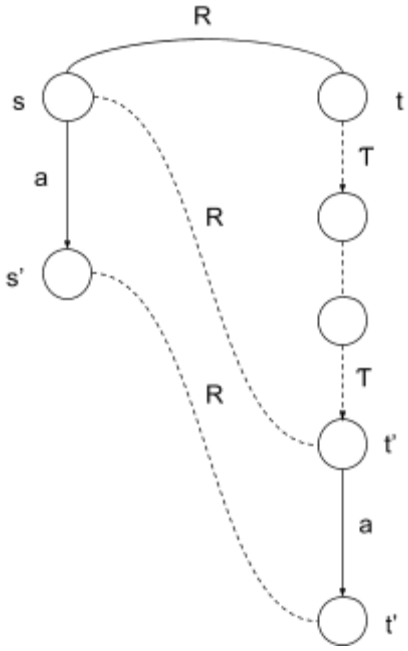
Weak Bisimulation Equivalence

Another method in introducing internal actions in Bisimilar Equivalent states is by considering them as Weak Bisimilar Equivalent states which was introduced by Robin Milner in 1980. Although the concept is similar to Branching Bisimulation Equivalence, there is one difference in terms of the precedence of actions and how they are seen.

In Branching Bisimulation Equivalence, if s --^{a}> s' for s' is an element of a set S,

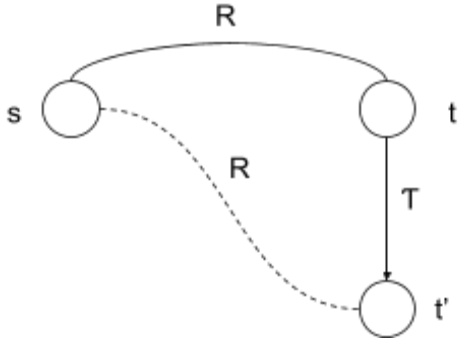
- then a = τ and s'Rt,
- or t',t'' are elements of set S such that t --$^{\tau}$> ... --$^{\tau}$> t''--^{a}> t'', sRt' and s'Rt''.

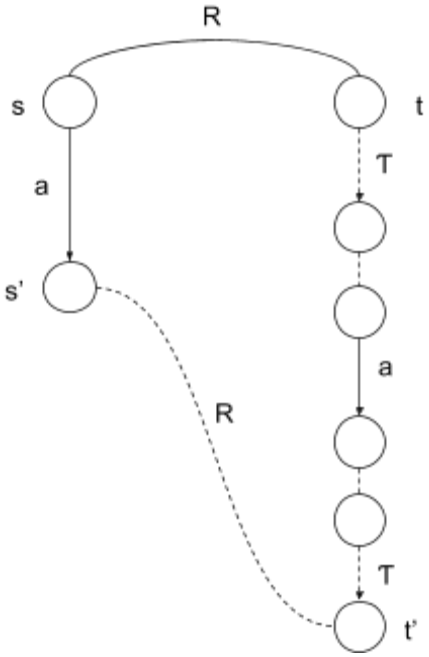




In Weak Bisimulation Equivalence, if s --^{a}> s' for s' is an element of a set S,

- then a = τ and s'Rt,
- or t' is an element of set S such that t --$^{\tau}$> ... --^{a}> ... --$^{\tau}$> t' and s'Rt'.

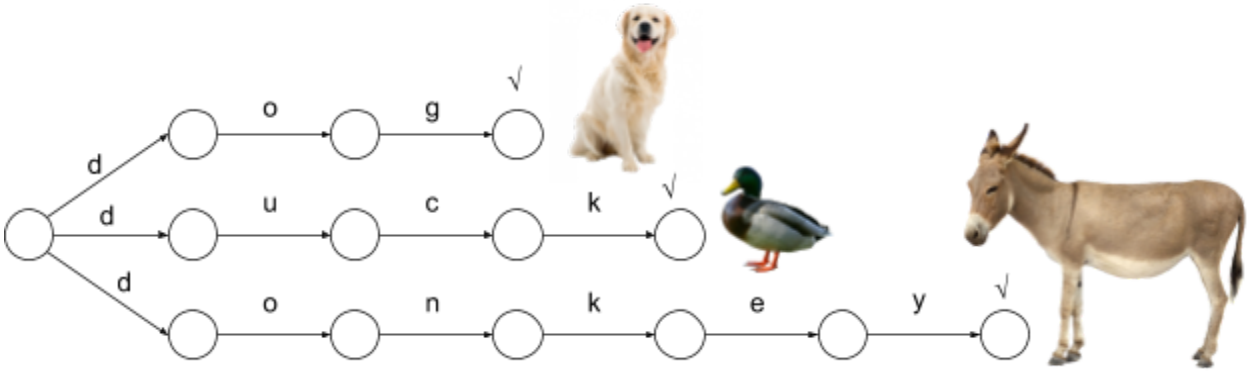




Language, Completed Trace, and Failure Equivalences

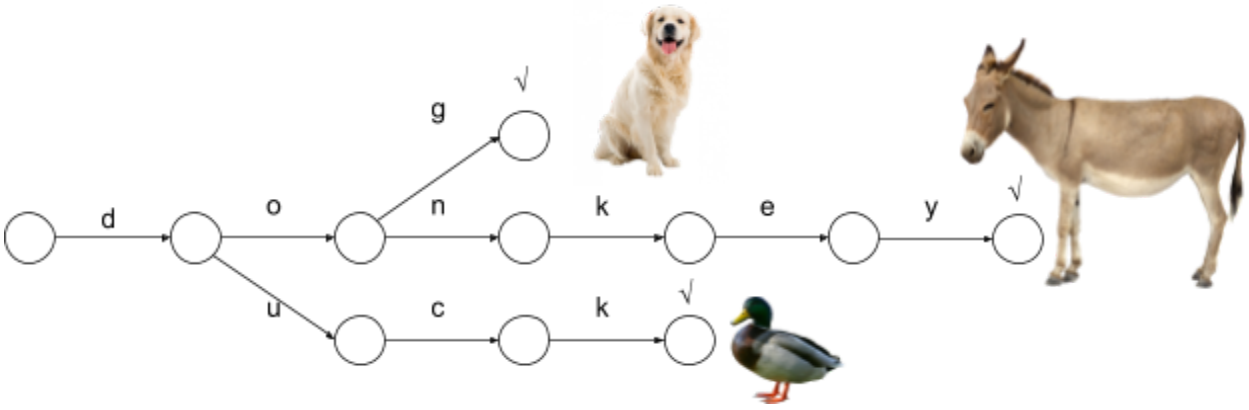
Natural language consists of words. Words are traces that end in a termination. The problem with the natural language is the inefficient way of reading the words.

Naturally, we see words a whole to identify its meaning. However, a computer recognizes words as a trace and it reads letter by letter to identify its meaning. If we wanted to find the meaning of a word and we searched the computer but it is reading it as a completely separate trace, it would be very inefficient since the computer will have to go back letter by letter if the word it's trying to find is not the one its reading as shown in the illustration below.



If the computer is trying to find the word donkey in the LTS above, it will take so much time to find it because after reading "d-o" in the dog trace, realizing the next letter should be "n" instead of "g", it will go back letter by letter and read "d" on the duck and again go back because the next letter is not correct until it reaches the "y" in the donkey trace and the termination symbol.

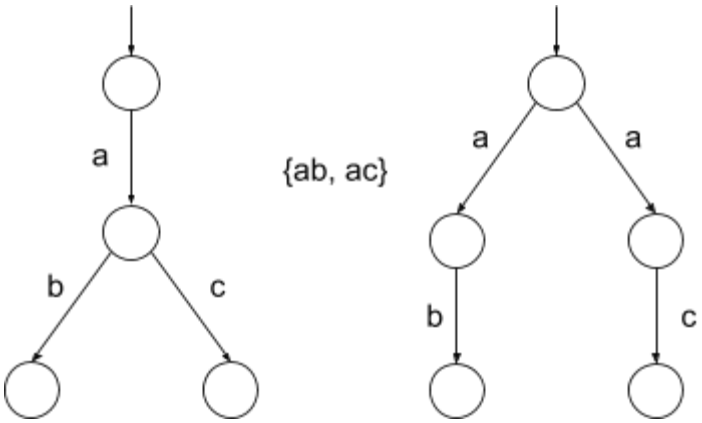
As you can see, it's very inefficient because it's slow to read and wastes a lot of memory space. A more efficient way to read it is through combining the states of the LTS and redirecting the actions if the next letter is different.



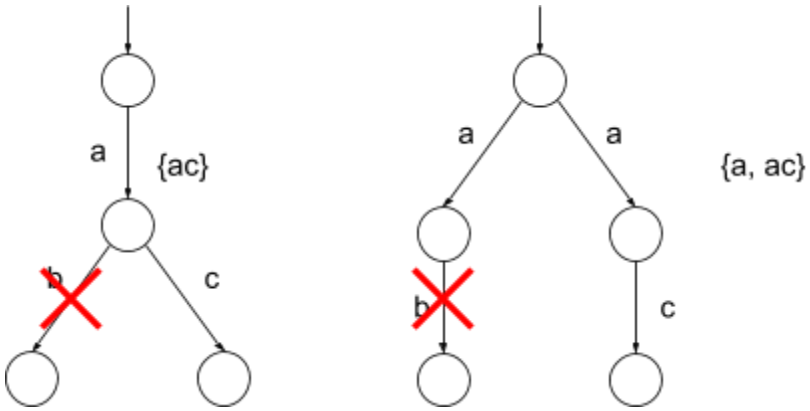
Now the computer can read the words faster and the LTS will take less memory space hence, making it more efficient to read.

Completed Trace Equivalence

Trace Equivalences are considered Completed when the sets of sequence are equal when we identify the traces that end in a deadlock or a termination occurs.

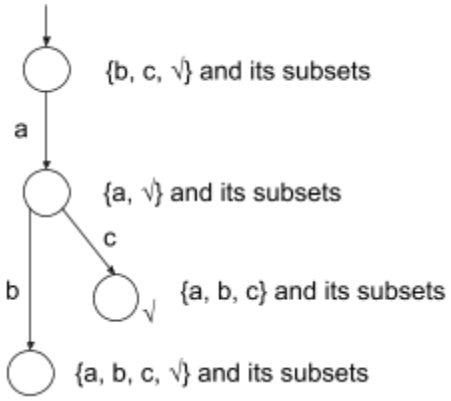


The illustration above shows two LTS that are Completed Trace Equivalent. However, by doing a simple operation called Blocking. We can block a certain action in both LTS and may result in a Completed Trace Inequivalence as you can see in the illustration below.

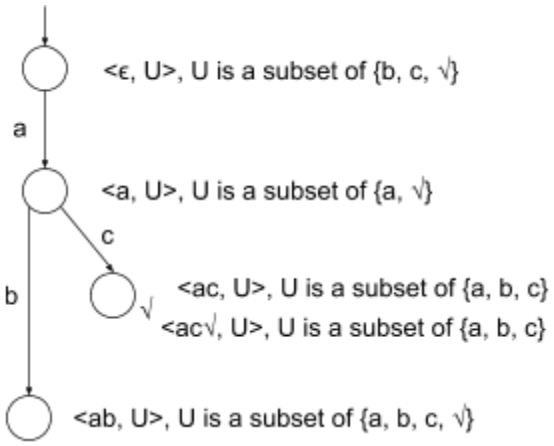


Failure Equivalence

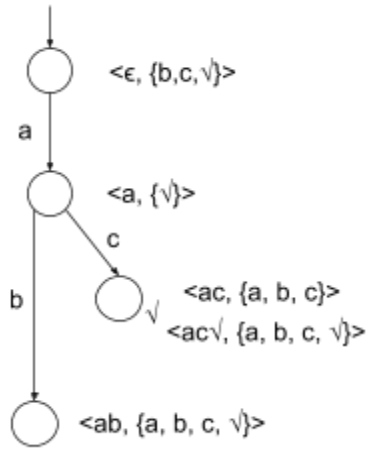
To solve the Completed Trace Inequivalence issue above, we can identify whether the LTS are Failure Equivalent under the basic operations involved. We can do that by finding the Refusal Sets and Failure Pairs of the states. Refusal Sets are the sets and their subsets wherein the impossible actions in that state are listed as you can see in the illustration below.

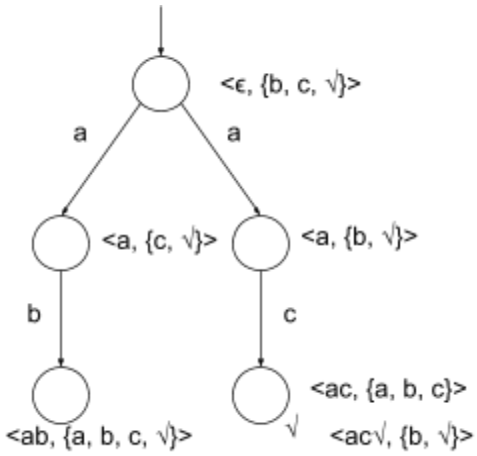


Failure Pairs are ordered pairs of Refusal Sets that are identified in such a manner that it shows the impossible actions in the corresponding state as illustrated below.



States are Failure Equivalent if they have the same Failure Sets. This will identify if the LTS are Failure Equivalent under basic operations.





When and which Behavioral Equivalence to use?

Strong Bisimulation Equivalence is always recommended because it is the safest one to use.

When reducing, *Trace Equivalence* is used when the sequence of actions are important, while *Branch Bisimulation Equivalence* is used when eventualities are not important since it removes the T actions.

Trace Equivalence has no clear way of calculating it but *Branch Bisimulation Equivalence* can be calculated easier than *Bisimulation Equivalence.*

When behaviors are not *Strongly Bisimulation Equivalent*, use *Failure Equivalence* especially when basic operations are involved in the process.

Divergence Preserving Branch Basimulations Equivalence is safe to use when divergences occur because it removes the T actions without changing the behavioral properties.

Weak Bisimulation Equivalence is used to compare with *Branch Bisimulation Equivalence.* But *Branch Bisimulation Equivalence* is still easier to calculate.

*Weak Trace Equivalence i*s the least recommended because even though it removes the T actions, it does not preserve the *Branch Bisimulation Equivalence.*