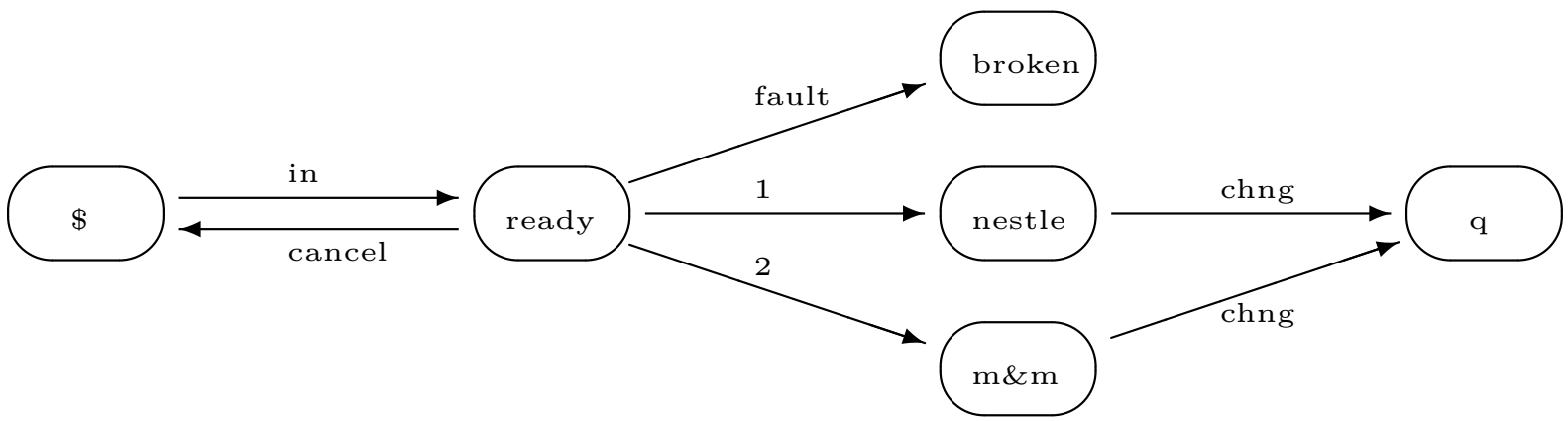
Concurrency vs. Nondeterminism: Automata

We can motivate concurrency by its absence. The point is that we can have systems that are nondeterministic, but are not concurrent. Consider the following faulty automaton to buy candy:



Concurrency vs. Nondeterminism: Automata (II)

Although in the above automaton each labeled transition from each state leads to a single next state, the automaton is nondeterministic in the sense that the automaton's computations are not confluent, and therefore completely different outcomes are possible.

For example, from the `ready` state the transitions `fault` and `1` lead to completely different states that can never be reconciled in a common subsequent state.

Concurrency vs. Nondeterminism: Automata (III)

So, the automaton is in this sense nondeterminisitc, yet it is strictly sequential, in the sense that, although at each state the automaton may be able to take several transitions, it can only take one transition at a time.

Since the intuitive notion of concurrency is that several transitions can happen simultaneously, we can conclude by saying the our automaton, although it exhibits a form of nondeterminism, has no concurrency whatsoever.

Automata as Rewrite Theories

We can specify such an automaton as a system module,

```
mod CANDY-AUTOMATON is
  sort State .
  ops $ ready broken nestle m&m q : -> State .
  rl [in] : $ => ready .
  rl [cancel] : ready => $ .
  rl [1] : ready => nestle .
  rl [2] : ready => m&m .
  rl [fault] : ready => broken .
  rl [chng] : nestle => q .
  rl [chng] : m&m => q .
endm
```

Rewite Rules as Transitions

Note that rewrite rules do not have an equational interpretation. They are not understood as equations, but as transitions, that in general cannot be reversed.

This is why, in a rewite theory (Σ, E, R) the equations in E are totally different from the rules R, since equations and rules have a totally different semantics.

However, operationally Maude will assume that the equations in E are confluent, terminating, and sort decreasing modulo axioms B, and will compute with such equations and also with the rules in R by rewriting, yet distinguishing equation simplification (the `reduce` command) from rewriting with rules (the `rewrite` command).

The **rewrite** Command

Maude can execute rewrite theories with the `rewrite` command (can be abbreviated to `rew`). For example,

```
Maude> rew $ .
rewrite in CANDY-AUTOMATON : $ .
rewrites: 5 in 0ms cpu (0ms real) (~ rewrites/second)
result State: q
```

The `rewrite` command applies the rules in a fair way (all rules are given a chance) hopefully until termination, and, if it terminates, gives one result.

The `rewrite` Command (II)

In this example, fairness saves us from nontermination, but in general we can esily have nonterminating computations.

For this reason the `rewrite` command can be given a numeric argument stating the maximum number of rewrite steps. Furthermore, using Maude's the `trace` command we can observe such steps. For example,

The **rewrite** Command (III)

```
Maude> set trace on .
Maude> rew [3] $ .
rewrite [3] in CANDY-AUTOMATON : $ .
*********** rule
rl [in]: $ => ready .
empty substitution
$ ---> ready
*********** rule
rl [cancel]: ready => $ .
empty substitution
ready ---> $
*********** rule
rl [in]: $ => ready .
empty substitution
$ ---> ready
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result State: ready
```

The `search` Command

Of course, since we are in a nondeterministic situation, the `rewrite` command gives us one possible behavior among many.

To systematically explore all behaviors from an initial state we can use the `search` command, which takes two terms: a ground term which is our initial state, and a term, possibly with variables, which describes our desired target state.

Maude then does a breadth first search to try to reach the desired target state. For example, to find the terminating states from the `$` state we can give the command (where the "`!`" in `=>!` specifies that the target state must be a terminating state),

The **search** Command (II)

```
Maude> search $ =>! X:State .
search in CANDY-AUTOMATON : $ =>! X:State .

Solution 1 (state 4)
states: 6 in 0ms cpu (0ms real)
X:State --> broken

Solution 2 (state 5)
states: 6 in 0ms cpu (0ms real)
X:State --> q
```

We can then inspect the search graph by giving the command,

The **search** Command (III)

```
Maude> show search graph .
state 0, State: $
arc 0 ===> state 1 (rl [in]: $ => ready .)

state 1, State: ready
arc 0 ===> state 0 (rl [cancel]: ready => $ .)
arc 1 ===> state 2 (rl [1]: ready => nestle .)
arc 2 ===> state 3 (rl [2]: ready => m&m .)
arc 3 ===> state 4 (rl [fault]: ready => broken .)

state 2, State: nestle
arc 0 ===> state 5 (rl [chng]: nestle => q .)

state 3, State: m&m
arc 0 ===> state 5 (rl [chng]: m&m => q .)

state 4, State: broken
state 5, State: q
```

The **search** Command (IV)

We can then ask for the shortest path to any state in the state graph (for example, state 5) by giving the command,

```
Maude> show path 5 .
state 0, State: $
===[ rl [in]: $ => ready . ]===>
state 1, State: ready
===[ rl [1]: ready => nestle . ]===>
state 2, State: nestle
===[ rl [chng]: nestle => q . ]===>
state 5, State: q
```

The **`search`** Command (V)

Similarly, we can search for target terms reachable by one or more rewrite steps, or zero or more steps by typing (respectively):

- `search` t `=>+` t' .
- `search` t `=>*` t' .

The **search** Command (VI)

Furthermore, we can restrict any of those searches by giving an equational condition on the target term. For example, all terminating states reachable from `$` other than `broken` can be found by the command,

```
Maude> search $ =>! X:State such that X:State =/= broken .
search in CANDY-AUTOMATON : $ =>! X:State
such that X:State =/= broken = true .

Solution 1 (state 5)
states: 6 in 0ms cpu (0ms real)
X:State --> q
```

The **search** Command (VII)

Of course, in general there can be an infinite number of solutions to a given search. Therefore, a search can be further restricted by giving as an extra parameter in brackets the number of solutions (i.e., target terms that are instances of the pattern and satisfy the condition) we want:

```
search [1] in CANDY-AUTOMATON : $ =>! X:State .

Solution 1 (state 4)
states: 6 in 0ms cpu (0ms real)
X:State --> broken
```

The **search** Command (VIII)

In our `CANDY-AUTOMATON` example the number of states is finite, but for a general rewrite theory the number of states reachable from an initial state can be infinite. So, even if we search for a single solution, the search process may not terminate, because no such solution exists. To make search terminating, at least for unconditional rewrite rules, we can add a second parameter, namely, a bound on the length of the paths searched from the initial state.

```
search [1, 1] in CANDY-AUTOMATON : $ =>! X:State .

No solution.
states: 2  rewrites: 2 in 0ms cpu (36ms real) (~ rewrites/second)
```

Labelled Transition Systems

Our `CANDY-AUTOMATON` example is just a special instance of a general concept, namely, that of automaton, also called a labeled transition system (LTS) by which we mean a triple: $A = (A, L, T)$ with:

- A is a set, called the set of states,
- L is a set called the set of labels, and
- $T \subseteq A \times L \times A$ is called the set of labeled transitions.

LTS's as Rewrite Theories

Note that we have associated to our candy automaton a rewrite theory (system module) `CANDY-AUTOMATON`.

This is of course just an instance of a general transformation, that assign to a LTS A a rewrite theory $R(A)$ with a single sort A, constants $x \in A$, and for each $(x, l, y) \in T$ a rewrite rule $l : x \longrightarrow y$.