directly from the triggering combination. In reaction systems, which are introduced in this section, the *if* parts specify the *conditions* that have to be satisfied and the *then* part specifies an *action* to be undertaken. Sometimes, the action is to *add* a new assertion; sometimes it is to *delete* an existing assertion; sometimes, it is to execute some procedure that does not involve assertions at all.

A Toy Reaction System Bags Groceries

Suppose that Robbie has just been hired to bag groceries in a grocery store. Because he knows little about bagging groceries, he approaches his new job by creating BAGGER, a rule-based reaction system that decides where each item should go.

After a little study, Robbie decides that BAGGER should be designed to take four steps:

1 The check-order step: BAGGER analyzes what the customer has selected, looking over the groceries to see whether any items are missing, with a view toward suggesting additions to the customer.
2 The bag-large-items step: BAGGER bags the large items, taking care to put the big bottles in first.
3 The bag-medium-items step: BAGGER bags the medium items, taking care to put frozen ones in freezer bags.
4 The bag-small-items step: BAGGER bags the small items.

Now let us see how this knowledge can be captured in a rule-based reaction system. First, BAGGER needs a working memory. The working memory must contain assertions that capture information about the items to be bagged. Suppose that those items are the items listed in the following table:

Item	Container type	Size	Frozen?
Bread	plastic bag	medium	no
Glop	jar	small	no
Granola	cardboard box	large	no
Ice cream	cardboard carton	medium	yes
Potato chips	plastic bag	medium	no
Pepsi	bottle	large	no

Next, BAGGER needs to know which step is the current step, which bag is the current bag, and which items already have been placed in bags. In the following example, the first assertion identifies the current step as the check-order step, the second identifies the bag as Bag1, and the remainder indicate what items are yet to be bagged:

Step is check-order
Bag1 is a bag
Bread is to be bagged
Glop is to be bagged
Granola is to be bagged
Ice cream is to be bagged
Potato chips are to be bagged

Note that working memory contains an assertion that identifies the step. Each of the rules in BAGGER's rule base tests the step name. Rule B1, for example, is triggered only when the step is the check-order step:

B1 If step is check-order
potato chips are to be bagged
there is no Pepsi to be bagged
then ask the customer whether he would like a bottle of Pepsi

The purpose of rule B1 is to be sure the customer has something to drink to go along with potato chips, because potato chips are dry and salty. Note that rule B1's final condition checks that a particular pattern *does not* match any assertion in working memory.

Now let us move on to a rule that moves BAGGER from the check-order step to the bag-large-items step:

B2 If step is check-order
then step is no longer check-order
step is bag-large-items

Note that the first of rule B2's actions deletes an assertion from working memory. Deduction systems are assumed to deal with static worlds in which nothing that is shown to be true can ever become false. Reaction systems, however, are allowed more freedom. Sometimes, that extra freedom is reflected in the rule syntax through the breakup of the action part of the rule, marked by *then*, into two constituent parts, marked by *delete* and *add*. When you use this alternate syntax, rule B2 looks like this:

B2 (add–delete form)
If step is check-order
delete step is check-order
add step is bag-large-items

The remainder of BAGGER's rules are expressed in this more transparent **add–delete syntax**.

At first, rule B2 may seem dangerous, for it looks as though it could prevent rule B1 from doing its legitimate and necessary work. There is no problem, however. Whenever you are working with a reaction system, you adopt a suitable *conflict-resolution procedure* to determine which rule

to fire among many that may be triggered. BAGGER uses the simplest conflict-resolution strategy, *rule ordering*, which means that the rules are arranged in a list, and the first rule triggered is the one that is allowed to fire. By placing rule B2 after rule B1, you ensure that rule B1 does its job before rule B2 changes the step to bag-large-items. Thus, rule B2 changes the step only when nothing else can be done.

Use of the rule-ordering conflict resolution helps you out in other ways as well. Consider, for example, the first two rules for bagging large items:

B3	If	step is bag-large-items
		a large item is to be bagged
		the large item is a bottle
		the current bag contains < 6 large items
	delete	the large item is to be bagged
	add	the large item is in the current bag

B4	If	step is bag-large-items
		a large item is to be bagged
		the current bag contains < 6 large items
	delete	the large item is to be bagged
	add	the large item is in the current bag

Big items go into bags that do not have too many items already, but the bottles—being heavy—go in first. The placement of rule B3 before rule B4 ensures this ordering.

Note that rules B3 and B4 contain a condition that requires counting, so BAGGER must do more than assertion matching when looking for triggered rules. Most rule-based systems focus on assertion matching, but provide an escape hatch to a general-purpose programming language when you need to do more than just match an antecedent pattern to assertions in working memory.

Evidently, BAGGER is to add large items only when the current bag contains fewer than six items.[†] When the current bag contains six or more items, BAGGER uses rule B5 to change bags:

B5	If	step is bag-large-items
		a large item is to be bagged
		an empty bag is available
	delete	the current bag is the current bag
	add	the empty bag is the current bag

Finally, another step-changing rule moves BAGGER to the next step:

[†]Perhaps a better BAGGER system would use volume to determine when bags are full; to deal with volume, however, would require general-purpose computation that would make the example unnecessarily complicated, albeit more realistic.

B6	If	step is bag-large-items
	delete	step is bag-large-items
	add	step is bag-medium-items

Let us simulate the result of using these rules on the given database. As we start, the step is check-order. The order to be checked contains potato chips, but no Pepsi. Accordingly, rule B1 fires, suggesting to the customer that perhaps a bottle of Pepsi would be nice. Let us assume that the customer goes along with the suggestion and fetches a bottle of Pepsi.

Inasmuch as there are no more check-order rules that can fire, other than rule B2, the one that changes the step to bag-large-items, the step becomes bag-large-items.

Now, because the Pepsi is a large item in a bottle, the conditions for rule B3 are satisfied, so rule B3 puts the Pepsi in the current bag. Once the Pepsi is in the current bag, the only other large item is the box of granola, which satisfies the conditions of rule B4, so it is bagged as well, leaving the working memory in the following condition:

Step is bag-medium-items
Bag1 contains Pepsi
Bag1 contains granola
Bread is to be bagged
Glop is to be bagged
Ice cream is to be bagged
Potato chips are to be bagged

Now it is time to look at rules for bagging medium items.

B7	If	step is bag-medium-items
		a medium item is frozen, but not in a freezer bag
	delete	the medium item is not in a freezer bag
	add	the medium item is in a freezer bag
B8	If	step is bag-medium-items
		a medium item is to be bagged
		the current bag is empty or contains only medium items
		the current bag contains no large items
		the current bag contains < 12 medium items
	delete	the medium item is to be bagged
	add	the medium item is in the current bag
B9	If	step is bag-medium-items
		a medium item is to be bagged
		an empty bag is available
	delete	the current bag is the current bag
	add	the empty bag is the current bag

Note that the fourth condition that appears in rule B8 prevents BAGGER from putting medium items in a bag that already contains a large item. If there is a bag that contains a large item, rule B9 starts a new bag.

Also note that rule B7 and rule B8 make use of the rule-ordering conflict-resolution procedure. If both rule B7 and rule B8 are triggered, rule B7 is the one that fires, ensuring that frozen things are placed in freezer bags before bagging.

Finally, when there are no more medium items to be bagged, neither rule B7 nor rule B8 is triggered; instead, rule B10 is triggered and fires, changing the step to bag-small-items:

B10	If	step is bag-medium-items
	delete	step is bag-medium-items
	add	step is bag-small-items

At this point, after execution of all appropriate bag-medium-item rules, the situation is as follows:

Step is bag-small-items
Bag1 contains Pepsi
Bag1 contains granola
Bag2 contains bread
Bag2 contains ice cream (in freezer bag)
Bag2 contains potato chips
Glop is to be bagged

Note that, according to simple rules used by BAGGER, medium items do not go into bags with large items. Similarly, conditions in rule B11 ensure that small items go in their own bag:

B11	If	step is bag-small-items
		a small item is to be bagged
		the current bag contains no large items
		the current bag contains no medium items
		the bag contains < 18 small items
	delete	the small item is to be bagged
	add	the small item is in the current bag

BAGGER needs a rule that starts a new bag:

B12	If	step is bag-small-items
		a small item is to be bagged
		an empty bag is available
	delete	the current bag is the current bag
	add	the empty bag is the current bag

Finally, BAGGER needs a rule that detects when bagging is complete:

B13	If	step is bag-small-items
	delete	step is bag-small-items
	add	step is done

After all rules have been used, everything is bagged:

Step is done
Bag1 contains Pepsi
Bag1 contains granola
Bag2 contains bread
Bag2 contains ice cream (in freezer bag)
Bag2 contains potato chips
Bag3 contains glop

Reaction Systems Require Conflict Resolution Strategies

Forward-chaining deduction systems do not need strategies for conflict resolution because every rule presumably produces reasonable assertions, so there is no harm in firing all triggered rules. But in reaction systems, when more than one rule is triggered, you generally want to perform only one of the possible actions, thus requiring a **conflict-resolution strategy** to decide which rule actually fires. So far, you have learned about rule ordering:

- *Rule ordering.* Arrange all rules in one long prioritized list. Use the triggered rule that has the highest priority. Ignore the others.

Here are other possibilities:

- *Context limiting.* Reduce the likelihood of conflict by separating the rules into groups, only some of which are active at any time.
- *Specificity ordering.* Whenever the conditions of one triggered rule are a superset of the conditions of another triggered rule, use the superset rule on the ground that it deals with more specific situations.
- *Data ordering.* Arrange all possible assertions in one long prioritized list. Use the triggered rule that has the condition pattern that matches the highest priority assertion in the list.
- *Size ordering.* Use the triggered rule with the toughest requirements, where *toughest* means the longest list of conditions.
- *Recency ordering.* Use the least recently used rule.

Of course, the proper choice of a conflict resolution strategy for a reaction system depends on the situation, making it difficult or impossible to rely on a fixed conflict resolution strategy or combination of strategies. An alternative is to think about which rule to fire as another problem to be solved. An elegant example of such problem solving is described in Chapter 8 in the introduction of the SOAR problem solving architecture.