+Decision Management Community Challenge Mar 2019
Offering Donated Organs for Transplant

Solution using Stateful Sessions in Drools

(Bob Moore, JETset Business Consulting, 20 Mar 2019)

1 Problem Statement (from the web site)

The Business Context:
The development of techniques for organ transplant means that now one person's death may lead to a lifesaving operation for another. This challenge considers one of the many crucial decision-making processes involved in this miracle of modern medicine. It is based on a (very) simplified description of one step in one of the processes used by the NHS Blood and Transplant (NHSBT) service in the UK.
Once an organ becomes available for donation, how is it decided who will be the recipient? The number of people wanting transplants considerably exceeds the number of organs available, so in general the recipient must be selected from a set of potential candidates for the organ. Strict guidelines are published by NHSBT to ensure this process is fair and equitable. A register is held of all patients awaiting transplants and using these guidelines a prioritized list of compatible candidates can be drawn up. In this context, a 'candidate' may be a named individual or may be one of the transplant centres around the country. In the latter case, the centre is responsible for selecting the individual patient most suited to receive the donated organ.
Once the candidate list is created, an organ can be 'offered' to the candidate at the top of the list. If the offer is accepted, the organ is delivered to the appropriate transplant centre where a patient can have surgery to implant the organ into their body. However, frequently the initial offer of the organ is declined. There are many reasons this may happen, for example:

- The surgeon responsible for the patient may determine that match between organ and potential recipient is inadequate
- The potential recipient may be too ill to undergo surgery at the time the organ becomes available
- There may be operational challenges at the transplant centre meaning surgery cannot take place in a timely manner

If the offer is declined, then the organ will be offered to the next candidate on the list, again this may be declined. Typically, an organ is offered two or three times before it will be accepted. A fall-back policy exists in the event that none of the candidates on the list can accept the offer.
Now we have a very high-level outline of how the recipient of a donated organ is selected from the possible candidates, let us move to a more concrete but still very simplified description of the offer process which will form the basis of the challenge.

The Example:
We will consider only heart and lung transplants, and the problem is probably best understood in terms of an example scenario. Let us suppose a young man dies in a motor accident and his heart and lungs become available for donation. The register of patients is consulted to find potential recipients. This will result in three distinct lists.

- An ordered list of named high priority patients requiring a heart transplant
- An ordered list of named high priority patients requiring a lung transplant
- An ordered list of transplant centres (TCs) responsible for patients requiring either heart transplants or lung transplants (or both)

To be concrete, let us suppose that the lists are as follows:

High Priority Heart Candidates	High Priority Lung Candidates	Transplant Centres Candidates
Adam at TC North	Diane at TC West	TC South
Barbara at TC South	Edgar at TC South	TC West
Charles at TC West	Fiona at TC North	TC North

An important element in the offering process is time. Offers can never be accepted immediately and are rarely declined immediately. Surgeons have to be contacted and they need time to consider any offer. Since donated organs are only viable for a limited period of time once an offer is made, a response is expected within about an hour.
In our scenario, the offering process starts out by TC North being contacted offering the Heart specifically for Adam, and TC West being contacted offering the Lungs specifically for Diane. Ideally, both offers are accepted, and the organs are retrieved and sent to the TCs.
Things get more complicated if offers are declined. When a donor's heart and lungs are both available and if no one on either of the high priority lists is able to accept the offers for them, NHSBT policy is that the two organs are now offered as a ' heart-lung block' to the TCs in turn. The two independent offer processes for a heart and a pair of lungs are replaced by one combined process for the block. Since the organs are initially offered separately and the responses are not instantaneous, this means there must be a synchronisation phase where the first organ to be declined by all the candidates on its high priority list ‘waits’ to see if the other organ is accepted by a candidate on the other high priority list, or if it also is declined by all.
When offered a heart-lung block a TC may accept the whole block, just the heart or just the lungs. Any declined organs are then offered to the next TC on the list.
To give a few of the scenarios which might play out:

- The first two offers of the heart are declined, but it is accepted for Charles, while the offer of the lungs is accepted for Diane.
- The first offer of the heart is declined, but it is accepted for Barbara, no one on the high priority lung list accepts the offer of the lungs. The lungs are offered to the TCs in turn and finally accepted by TC North.
- No offers are accepted on the high priority lists, so the organs are offered as a heart-lung block to TC South. TC South accepts the heart but declines the lungs which are then offered to TC West, which accepts the lungs.

The last scenario is the critical one, as two independent asynchronous processes merge into a single process. This changes the decision-making process from simply picking the next option on a list to something more complex.

The Objective:
The objective of the challenge is to devise a decision system which determines what to do next, following a response to an offer of an organ or organs. The response can be a full accept ('I accept all the offered organs'), a partial accept (e.g. 'I accept the heart but decline the lungs') or a decline ('I decline all the offered organs').
The possible next steps are:

- Report the process is complete if the response is an accept and the organ in the response is the only one left which had not already been accepted
- Invoke the fall-back policy for an organ if the response is a decline and now all candidates have been offered and have declined the organ
- Take no action at this point (this occurs when one organ has been declined by all candidates on its high priority list, but the other organ has not)
- Offer the organ(s) in the response to the next candidate on the (appropriate) list
- Offer the other organ to the one in the response to the next candidate on the appropriate list for this other organ (this occurs for example when the lungs are 'waiting' to see if the heart is accepted by a candidate on the high priority list, and the current offer response is an accept for the heart, at which point the lungs can be offered to the first TC on the TC list)
- Offer a heart-lung block to the first TC on the TC list because all candidates on both high priority lists have declined the individual organ offers
- Offer the remaining unallocated organ following a partial accept of a heart-lung block to the next TC on the TC list

The next step obviously depends on the candidate lists and the organs being offered as well as the offer response, but the sting in the tail is that it also depends on the current state of the system. Specifically:

- Which offers have previously been declined (so – among other things – the next candidate on a list can be determined)
- Which offers have previously been accepted (so – among other things – you don't try to offer a heart-lung block after someone else has already accepted the heart)
- Which offers have been made, for which a response has not yet been received (so – among other things – you know whether to move from single organ offers to a heart-lung block offering)

Hence, we are dealing with a stateful system. With this in mind, to be deemed 'satisfactory' a solution should make it clear *where* the current state is held in the solution and *how* it is both accessed *and* updated. One hopes that solutions to the challenge based around both stateful and stateless decision engines will be submitted, with state held internally or externally as appropriate.
Typically, the system will be called multiple times before all the organs from a single donor are accepted. It would be nice therefore if solutions could also outline how this recurrent activation of the system might take place, including the initial offering of the organ(s), perhaps with BPMN or CMMN diagrams as deemed appropriate.

2 Getting a handle on things with a Domain Model

The objective of the challenge is to decide what action to take in response to an offer, and as identified in the challenge, to do this we need to understand the current state of the state of the offering process. So, I thought the first thing I should do was probably try and build a model of what we were talking about.The challenge is written in descriptive style, so needs to be picked through to work out what business concepts are needed. My reading caused me to come up with the following concepts which form the basis of the domain object model:

- The organ donor
- The organ(s) which are being donated
- The candidates
- The candidate lists
- Offers which have been made
- Responses to offers which have been received
- The next action to take

While central to processing, at first sight the donor does not seem to be explicitly used in the decision process. However, although not directly stated in the challenge, it should be obvious that offers for organs from many different donors may be being to be made at the same time[1], so the decision system needs to handle concurrent offerings. An object representing the donor makes this simple by letting us link the other objects to it.

Before starting, I'd made a decision that I would use the Drools Rule Engine, on the grounds that I wanted to explore using a stateful approach within the engine (something I've never needed to do before), It's free to use, and while I find the arguments put forward a bit spurious, the Drools documentation definitely champions stateful over stateless reasoning. Since Drools is a Java toolkit, the object model comprises Java classes. After many false starts, I emerged with the following six classes:

Donor The 'root' object – it essentially captures the full state of how the offering process is progressing. It links to the organs being offered for donation (**Organ** class), the various candidate lists (**CandidateList** class) and a 'next action' (**NextAction** class). It keeps track of which candidates have been offered an organ and for which an offer is awaited in a set of **currentOffers**. To provide a 'history' of the processing it also records the offer responses (**OfferResponse** class) (although this is not actually used in the decisioning itself). The donor object has a **donorId** attribute, which identifies the donor, and all the objects involved in the a given offering share this attribute. It has a **status** attribute, indicating where in the overall offering process we are. All the other object classes (apart from **CandidateList**), also have a **status** attribute used in determining which rules are applicable.

[1] Visit https://www.organdonation.nhs.uk/supporting-my-decision/statistics-about-organ-donation/ for statistics about organ donations in the UK. Note that heart/lung donations average only about one a day, so multiple concurrent donations for these organs is much less common than for say kidney donations.

Organ This simply identifies an organ (by its **organId**) linked to a donor and indicates its status. If both a heart and pair of lungs are available a heart-lung block is created, but if only one organ is available it is not. The possible status values of an organ are:

Unavailable	this organ cannot be offered for some reason
Offered	this organ is currently being offered to candidates
Accepted	this organ has been accepted by a candidate
Fallback	none of the candidates have accepted the organ so it needs to be passed to the fallback process

CandidateList This simply attaches a name (its **organId**) to a list of candidates (**Candidate** class). As will be seen the lists are used more like stacks than lists, so are built in ascending priority, with the highest priority candidates at the end of the list, not the beginning. Again, to simplify matters the name of a high priority list is the name of the organ it relates to, so the lists are called “HEART”, “LUNGS” and “CENTRE”

Candidate This identifies a candidate recipient for an organ offered by a donor. It identifies the **candidateId** (which may be an individual or a TC), the TC they are linked to (**centreId**), and the organ for which they are a candidate. **Candidate** objects appear in **CandidateList** instances, but are used both in the **NextAction** class where they identify who an offer is to be made to, and in the **OfferResponse** class where they give the identify which to the offer the response relates to.

NextAction This object is the ‘output’ of the decision process and specifies what to do next. There are four ‘status’ values for the next action:

Complete	Signals all organs have been accepted and processing is over
Fallback	Signals processing is over but that some organs need to be passed to the fallback process
No Action	Signals no action is needed at this time
Make Offers	Identifies which candidate(s) to offer organs to

For the last action, additionally the **NextAction** object has a list of which organs to make offers for and the appropriate **Candidate** for each. In practise the list only ever contains more than one entry if at the start of the offering process the donor has entries in both the heart and the lungs high priority lists.

OfferResponse: This object is ‘usually’ the input object to the decision process and specifies the response to an offer. Within the execution framework its used to represent offer and response. It has four possible statuses:

Full Accept	An acceptance of all organs offered
Partial Accept	An acceptance of either the heart or the lungs from a heart-lung block
Decline	A decline of the offer

New	A special value used when for the initial 'dummy' response created when a new donor is submitted

The **OfferResponse** additionally contains a copy[2] of the **Candidate** object from the previous **NextAction**, to show which offer it is a response to**,** and in the event of the response being a partial accept, an **acceptedOrganId** attribute identifies which organ (heart or lungs) was.

The Java classes used to implement these are basically POJOs[3], with a few simple utility routines to reduce the amount of code needed to add and remove items from lists and other collections.

To slightly simplify the rules, if there are no high priority candidates for an organ, an empty candidate list is created (rather than having a **null**). Also, if a donor is providing both a heart and a pair of lungs, an organ of type heart-lung block is also added to the donor (so a donor has either one organ for offer, only the heart or only the lungs, or three heart, lung and block). The 'id' attributes (**donorId**, **candidateId**, **centreId** & **organId**) are as strings to simplify printing but are assumed to be unique.

3 Making the Decision

Reading the challenge, it is evident that the decision will be based on a number of business rules, but as with the model, the rules need to be abstracted from the descriptive text. Ron Ross' submission[4] for the challenge, does an excellent job of picking out the core 'next action' rules which tells us how to work out the next action, but it is incomplete in terms of building an executable solution. The rules Ron identifies tell you what to do when the system is in a particular state. But we start the decision making when we receive an offer response, which we need to use to update the state. So, we need some rules to update the state with the consequences of the response, before the 'next action' rules can be run. In addition, as glossed over a bit in the challenge, we will also need some rules to initialise the state correctly when the offering process for a donor commences. My solution involves four separate groups of rules, 'Initialise', 'Response', 'Next Action' and 'Tidy Up'.

3.1 Drools Preliminaries

Before we go into the details a little about Drools for those unfamiliar with it.

Drools is an open source rules engine written in Java. To use it, you create a rules engine instance (a **KieSession** object) which loads the rules you want to run, then you build some Java objects, inject them into the rule engine and kick it off. I find the Drools rule language very 'technical' in style. This is not a problem for me in terms of understand and writing the rules, but it does mean that reading the rules can be challenging if you are not used to programming.

---

[2] We use a clone rather than a reference to the original Candidate to avoid aliasing issues, where altering the object in the Java execution framework unintentionally updates the object in working memory

[3] Plain Old Java Objects – basically just a collection of attributes which might be primitive values or other objects

[4] See https://dmcommunity.files.wordpress.com/2019/03/challenge2019mar.behavioral-business-rules-for-heart-lung-donations.pdf

The default mode of operation used by Drools is essentially forward chaining using an algorithm broadly based on RETE, where the rules are applied to a 'working memory'. This requires the objects which the rules are applied to, to be explicitly placed into the working memory with an **insert** directive. When objects are modified, or removed, **modify** and **delete** directives generally need to be used to ensure the working memory recognises the changes. When started, the engine fires all the rules it can, and when it is finished, you return to the calling Java code. Typically, the outcome is then read off one or more of the objects one originally inserted, which is the approach used here.

Drools can either be used in a stateful or stateless manner. In the stateful mode, after all the rules have fired, the engine retains its working memory (the 'state'), so if one wants to insert some new objects, or modify the existing ones, one can do this while retaining whatever you pushed into the working memory before. This seems ideal for this kind of problem, because broadly the processing we need to do is:

- Insert the initial information about the donor and run the rules. This will give us an initial set of offers to send out to candidates for the donated organs (at this point we drop out of Drools back into Java)
- Send out the offers

At this point conceptually we 'wait' for a response to the offer(s) (we will come back to exploring how we 'wait' when discussing activation frameworks for the decision service). Note we might send out one or two offers initially (one for a heart, one for a pair of lungs).

When we get a response to an offer what we do is:

- Insert the offer response into Drools, so the rules can combine the response with the existing state of the donor offering process to decide the next appropriate action (at this point we drop out of Drools back into Java)
- If the next action is Make Offers we send out a new offer and 'wait' for a response – note at this stage we only ever make at most one new offer for each response we respond to[5].
- If the next action is No Action, then we just 'wait' for another offer response (there will be one!)
- Finally, if the next action is Complete or Fallback we are finished (at least with this donor)[6]

The key thing that Drools is doing for us here is preserving the state in its working memory. If Drools doesn't do it for us, we'll need to do it ourselves.

One of the general problems with getting declarative rules to 'work' correctly, is that in most cases there are generally some 'process' aspects to the way in which they are used. Most commonly we find some rules need to be looked at before others. Drools has several ways to control execution. My preferred method is a rule flow, but using

[5] For other organs which can be donated, this may not be the case, for example a liver can be divided into two lobes each of which can be transplanted separately. So, what starts as one offer can become two. A pair of lungs can also be split, and each lung provided to a different recipient. As noted in the challenge description we are looking at a simplified version of a small part of the overall problem.

[6] Well finished with the offer process. The next steps involve doctors, nurses and a cast of supporting characters to all of whom I take my hat off in admiration.

these involves extra work, so for simplicity I have used rule priorities – or salience as it is termed in Drools. The 'Initialise Rules' have highest priority, so are considered first, the 'Response Rules' which update the state are considered next. The 'Next Action Rules' are considered as the next step and finally the 'Tidy Up Rules' come into play.

3.2 Simplifying Assumptions

To make life easier, I have made assumptions with regard to the way the offer process works. While I probably have made some other assumptions which I've not recognised as such, the principal ones I am aware of are:

- There will always be at least one entry on the Transplant Centre (TC) list.
- Partial Accept responses are only received for offers of a heart-lung block.
- There will be exactly one response to every offer.

The first assumption is how things work in the 'real world', so is quite reasonable. The second assumption is simply a data consistency matter. The final assumption say we are not considering some problems which may arise in the real-world environment. It basically says we can assume we never have situations where:

- We get a response which does not relate to an offer which has been made
- We never get a response to an offer
- We get multiple responses for the same offer.

In the 'real world' the first of these scenarios should never happen, but a real solution should be able recognise it and handle it appropriately. The second scenario can arise in the real world. If a TC fails to respond to an offer within about an hour, the Duty Office which makes the offers will eventually be obliged to unilaterally withdraw the offer and make a fresh offer to the next candidate on the list[7]. Obviously after an offer is unilaterally withdrawn, a delayed response from the TC might later turn up, leading to the final scenario. It is certainly feasible to add the ability handle failures to respond to an offer into to the solution and indeed Drools has sophisticated event processing mechanisms, easily capable of handling this, but adding this in was deemed to overcomplicate the solution.

3.3 State Management

One of the primary objectives of the challenge is to explore some ideas about stateful systems, rather than the stateless ones which are more commonly addressed. In reality the problems posed in terms of managing the state here are about the bare minimum needed to make things a little interesting, but they are enough. The two issues to we need to track are:

- What is the status of each organ? Is it 'offered', 'accepted' or 'unavailable'?
- What is the status of each candidate in each list? Have they been made an offer? Has a response been received? What kind of response was it?

The first is very straightforward, since we preserve the state between successive calls in the rule engine, all we need to do is have rules update it appropriately. The second requires some implementation decisions. One approach would have been to mark each

[7] In practise the Duty Office makes concerted efforts to contact TCs over delayed responses, so a unilateral withdrawal of an offer is a rare occurrence.

candidate as an offer is made and again when the response is received, but this requires searching down the list to find the first unoffered candidate. A simpler approach is to treat a candidate list like a stack and 'pop' the next candidate from the list – after all once we have made an offer to a candidate, regardless of the response, we are not going to send then another. This not only makes finding the next candidate easy, but also determining if the candidate list is empty. Once a candidate is popped, it is pushed into a set of ongoing offers. When a response to the offer is received, it can then be matched with and removed from this set. The presence of outstanding offers is critical in terms of detecting whether the offering of one organ needs to be paused while the fate of another is determined.

3.4 The Rules

3.4.1 The Initialise Rules

These rules handle the creation of the appropriate initial state for the donor offering. These rules come into play once a **Donor** object is inserted into working memory and the rule engine is activated. The first rule simply inserts the subsidiary objects into working memory for a 'new' donor:

```
rule "Initialise - Add Objects to Working Memory"
  salience 100
  when
    $donor: Donor(status == Constants.NEW)
  then
    insert($donor.getNextAction());
    for(Organ organ: $donor.getOrgans().values()) {
      insert(organ);
    }
    for(CandidateList candidateList: $donor.getCandidateLists().values()) {
      insert(candidateList);
      for(Candidate candidate: candidateList.getCandidates()) {
        insert(candidate);
      }
    }
    modify($donor) { setStatus(Constants.INITIATING) }
end
```

As may be seen Drools rules have a very programmatic look to them, not least because the actions of the rule in the 'then' part are basically fragments of Java code. The clause in the 'when' part of the rule simply seeks out any **Donor** objects in memory with the status 'new'. If any are found, then we 'bind' the donor object to the variable **$donor** so it can be used in the logic of the 'then' part of the rule. The donor **Organ**(s), the **nextaction** instance, the **CandidateList**s and their associated **Candidate** objects are then all inserted into the working memory, and we update the donor status to 'initiating'

The next four initialise rules set up the initial state of the organs to be offered. The first two concern the handling of a heart-lung block. Basically, they ensure that you don't try to offer a heart-lung block, if there are candidates on any of the high priority list, but you do if there are not (and both a heart and lungs are available). In the latter case additionally, you do not offer the heart or lungs separately.

```
rule "Initialise - Set Heart-Lung block unavailable"
  salience 100
  when
    $donor: Donor($donorId: donorId, status == Constants.INITIATING)
    exists CandidateList($donorId == donorId,
      organId == Constants.HEART || == Constants.LUNGS, candidates.size() > 0)
    $organ: Organ($donorId == donorId, organId == Constants.HEART_LUNG_BLOCK)
  then
    modify($organ) { setStatus(Constants.UNAVAILABLE) }
end
```

The first condition in this rule – which is shared by the other remaining rules binds the **$donorId** to ensure we are dealing only with the objects related to the correct donor, and that we are still in the process of initialising the state. The second rule checks if either of the high-priority lists are non-empty and the last checks pick out the **Organ** objects representing the heart, the lungs and the heart-lung block.

```
rule "Initialise - Offer Heart-Lung block"
  salience 100
  when
    $donor: Donor($donorId: donorId, status == Constants.INITIATING)
    CandidateList($donorId == donorId, organId == Constants.HEART, candidates.size() == 0)
    CandidateList($donorId == donorId, organId == Constants.LUNGS, candidates.size() == 0)
    CandidateList($donorId == donorId, organId == Constants.CENTRE, candidates.size() > 0)
    $heart: Organ($donorId == donorId, organId == Constants.HEART)
    $lungs: Organ($donorId == donorId, organId == Constants.LUNGS)
    $block: Organ($donorId == donorId, organId == Constants.HEART_LUNG_BLOCK)
  then
    modify($heart) { setStatus(Constants.UNAVAILABLE) }
    modify($lungs) { setStatus(Constants.UNAVAILABLE) }
    modify($block) { setStatus(Constants.OFFERED) }
end
```

This rule will fire if both the high priority lists are empty in which case, we also make the heart and lungs ‘unavailable’, and ready things to offer the heart-lung block.

The next pair of rules determine whether to offer the heart or the lungs if it we have only one organ, or the organ’s high priority list has entries (so the rule "Initialise - Set Heart-Lung block unavailable" will fire while "Initialise - Offer Heart-Lung block" will not). The first rule may fire twice, once for heart and once for lungs if both high priority lists are populated. The second can only fire once (because there is only organ)

```
rule "Initialise - Offer Organ with HP List"
  salience 100
  when
    $donor: Donor($donorId: donorId, status == Constants.INITIATING)
    $organ: Organ($organId: organId, $donorId == donorId, status == Constants.NEW,
       organId != Constants.HEART_LUNG_BLOCK)
    CandidateList($donorId == donorId, $organId == organId, candidates.size() > 0)
  then
    modify($organ) { setStatus(Constants.OFFERED) }
end

rule "Initialise - Offer Organ if it is the only one"
  salience 100
  when
    $donor: Donor($donorId: donorId, status == Constants.INITIATING)
    $organ: Organ($donorId == donorId, status == Constants.NEW)
    not (exists( Organ($donorId == donorId, organId == Constants.HEART_LUNG_BLOCK)))
  then
    modify($organ) { setStatus(Constants.OFFERED) }
end
```

Once the initial offer status of the organs has been established, we need determine what the initial offers should be. One could use some special rules or mechanism to do this, but it turns out to be much simpler (and more elegant?) to make use of the same set of rules (the 'Next Action Rules') as we do when responding to an offer we have already made. However, the 'Next Action Rules' need an **OfferResponse** object to work with, so to do this we need a rule to insert a 'dummy' response into working memory to trigger the firing of the 'Next Action Rules'.

```
rule "Initialise - Generate dummy offer response"
  salience 100
  when
    $donor: Donor($donorId: donorId, $nextAction: nextAction, status == Constants.INITIATING)
    $organ: Organ($donorId == donorId, status == Constants.OFFERED)
  then
    Candidate candidate = new Candidate($donorId, $organ.getOrganId());
    OfferResponse offerResponse = new OfferResponse(candidate);
    insert(offerResponse);
end
```

One important point to notice about this rule, is that it fires for each organ which is to be offered, so in the case we have entries on both heart and lung priority lists, we generate two **OfferResponse** objects. The status of these **OfferResponse** objects is defaulted to 'new', so will not trigger any of the 'Response' rules (see below)

The final initialise rule updates the status of the donor to indicate initialisation is complete and the offering is now 'in progress'.

```
rule "Initialise - Mark donor offering in progress"
  salience 99
  when
    $donor: Donor($donorId: donorId, $nextAction: nextAction, status == Constants.INITIATING)
    forall( Organ($donorId == donorId,
      status == Constants.OFFERED || == Constants.UNAVAILABLE) )
  then
    modify($donor) { setStatus(Constants.IN_PROGRESS) }
end
```

Note here the 'forall' condition checking that the initialisation of all the organs is complete prior to moving to the next set of rules, and the lower salience which ensures that the other rules fire before this one. It is also worth observing that once the donor state has changed from 'new' and 'initialising' these initial rules cannot not fire again for this donor.

3.4.2 The Response Rules

These rules fire following the insertion of an **offerResponse** into working memory following receipt of a response to an offer generated from a previous invocation of the rule engine.. From the challenge, the status of an **offerResponse** is a 'decline', a 'full accept' or a 'partial accept' and we have three rules one for each possible response. As noted above the dummy **offerResponse** created when a donor is initialised has a status 'new' so none of these rules are applicable.

The decline rule is the simplest:

```
rule "Process DECLINE Response"
  salience 50
  when
    $donor : Donor($donorId: donorId, $nextAction: nextAction,
      status == Constants.IN_PROGRESS)
    $offerResponse : OfferResponse($offeredTo: candidate, donorId == $donorId,
      status == Constants.DECLINE)
    exists (Candidate($offeredTo.candidateId == candidateId)
      from $donor.currentOffers.values())
  then
    $donor.addOfferResponse($offerResponse);
    modify($nextAction) { reset() }
    modify($donor) { removeOffer($offeredTo.getCandidateId()) }
end
```

The conditions in the ‘when’ part of the decline rule and the actions in the ‘then’ part of the rule are shared with the rules for full and partial accepts, but the latter have a bit more work to do, so have additional bindings in the conditions and additional actions in the then part. Essentially the conditions identify a donor which is still in progress, and an offer response for this donor to respond to (which in this case is a ‘decline’). It then checks this response matches an offer which has been made (the ‘exists’ condition). The actions are to push the **offerResponse** onto the **offerResponses** list of the donor (this is simply to give a traceback of the offer process and has no role in actually making a decision), to ‘reset’ the donor’s **nextAction** instance (since this will currently hold the previously determined ‘next action’ from the last invocation of the rules), and removes the offer corresponding to the current response from the list of current offers as we now have received a response to it.

The ‘full accept’ rule is as follows:

```
rule "Process FULL_ACCEPT Response"
  salience 50
  when
    $donor : Donor($donorId: donorId, $nextAction: nextAction,
      status == Constants.IN_PROGRESS)
    $offerResponse : OfferResponse($offeredTo: candidate, donorId == $donorId,
      status == Constants.FULL_ACCEPT)
    exists (Candidate($offeredTo.candidateId == candidateId)
      from $donor.currentOffers.values())
    $acceptedOrgan: Organ(donorId == $donorId, organId == $acceptedOrganId)
  then
    $donor.addOfferResponse($offerResponse);
    modify($nextAction) { reset() }
    modify($donor) { removeOffer($offeredTo.getCandidateId()) }

    CandidateList candidateList = $donor.getCandidateLists().get($acceptedOrgan.getOrganId());
    if(candidateList != null) {
      modify(candidateList) { getCandidates().clear() }
    }
    modify($acceptedOrgan) { setStatus(Constants.ACCEPTED) }
end
```

As can be seen the conditions are very similar to a decline, but in this case, we demand the status is a ‘full accept’ and we bind a variable to the ‘accepted’ organ. In the then part the same actions are taken as for a decline, but then we need a couple of extra actions. The first is to clear any remaining entries from the high priority list for this organ. In the ‘Next Action’ rules we test if the high priority lists are empty, so failing to do this block offers from the TC list[8]. Last but not least, we mark the organ as accepted.

[8] I’m sure there is a more elegant way to do this, but I’ve not figured one out yet

Again the ‘partial accept’ rule shares its first three conditions and first three actions with the other rules in this group. It additionally requires binding identifiers to the organ which was offered (which is always a Heart-Lung Block), the organ which was accepted, (Heart or Lungs) and the remaining organ once the accepted one is taken away. The addition actions then simply set the status of the three organs appropriately.

```
rule "Process PARTIAL_ACCEPT Response"
  salience 50
  when
    $donor : Donor( $donorId: donorId, $nextAction: nextAction,
      status == Constants.IN_PROGRESS )
    $offerResponse : OfferResponse($acceptedOrganId: acceptedOrganId,
      $offeredTo: candidate, donorId == $donorId, status == Constants.PARTIAL_ACCEPT)
    exists (Candidate($offeredTo.candidateId == candidateId)
      from $donor.currentOffers.values())
    $offeredOrgan: Organ(donorId == $donorId, organId == $offeredTo.organId)
    $acceptedOrgan: Organ(donorId == $donorId, organId == $acceptedOrganId)
    $remainingOrgan: Organ(donorId == $donorId, this != $offeredOrgan &&  != $acceptedOrgan)
  then
    $donor.addOfferResponse($offerResponse);
    modify($nextAction) { reset() }
    modify($donor) { removeOffer($offeredTo.getCandidateId()) }

    modify($offeredOrgan) { setStatus(Constants.UNAVAILABLE) }
    modify($acceptedOrgan) { setStatus(Constants.ACCEPTED) }
    modify($remainingOrgan) { setStatus(Constants.OFFERED) }
end
```

3.4.3 The Next Action Rules

The ‘initialise’ and ‘response’ rules are really all about setting up and updating the state of the offering process, it is the ‘next action rules’ which generate the outcome decision. There are five of these. The first two rules detect that the offering process has finished, the other three generate new offers of the organs which have not yet been accepted. In a sense there is also an implicit ‘sixth’ rule, which is simply to do nothing. This corresponds to the situation of there being an outstanding offer which we need to wait for before deciding what happens next.

The first rule addresses ‘normal’ completion – where all the offered organs have been accepted.

```
rule "Next Action - Complete"
  when
    $donor: Donor( $donorId: donorId, $nextAction: nextAction,
      status == Constants.IN_PROGRESS)
    forall(Organ($donorId == donorId,
      status == Constants.ACCEPTED || status == Constants.UNAVAILABLE))
  then
    modify($donor) { setStatus(Constants.COMPLETE) }
    modify($nextAction) {setStatus(Constants.COMPLETE) }
end
```

Normal completion has occurred if all the organs are accepted or unavailable. We set the next action and the status of the donor as ‘complete’

If we have made offers and got responses back from all the available candidates and at least one organ is still being ‘offered’ we revert to ‘fallback’ which is the alternative form of completion:

```
rule "Next Action - Fall Back"
  when
    $donor: Donor($donorId: donorId, status == Constants.IN_PROGRESS)
    $nextAction: NextAction($donorId == donorId, status == Constants.NO_ACTION)
    CandidateList($donorId == donorId, organId == Constants.HEART, candidates.size() == 0)
    CandidateList($donorId == donorId, organId == Constants.LUNGS, candidates.size() == 0)
    CandidateList($donorId == donorId, organId == Constants.CENTRE, candidates.size() == 0)
    $organ: Organ($organId: organId, $donorId == donorId, status == Constants.OFFERED)
  then
    modify($donor) { setStatus(Constants.COMPLETE) }
    modify($organ) { setStatus(Constants.FALLBACK) }
    modify($nextAction) {setStatus(Constants.FALLBACK),
      addOffer(new Candidate($donorId, $organId)) }
end
```

The conditions check that we are not about to make an offer to the very last candidate (the next action is 'no action' rather than 'make offers'), that the all the lists are empty, and that an organ is still under offer (a little consideration shows that actually only one organ can be in this state, as if we started offering with both heart and lungs, and neither has be accepted, we must be offering the block). The action sets the donor status to complete and the next action and the status of the remaining organ to fallback. It also creates a 'dummy' candidate to hold the unallocated organ for easy reference.

The next rule deals with making offers from the high priority lists:

```
rule "Next Action - Make Offer from HP List"
  when
    $donor: Donor($donorId: donorId,
      $nextAction: nextAction, status == Constants.IN_PROGRESS)
    $organ: Organ($organId: organId, $donorId == donorId,
      status == Constants.OFFERED, organId != Constants.HEART_LUNG_BLOCK)
    $offerResponse : OfferResponse(donorId == $donorId,
      candidate.organId == $organId)
    $candidateList: CandidateList($donorId == donorId,
      organId == $organId, candidates.size() > 0)
    forall (Candidate(organId != $organ.organId) from $nextAction.offers)
  then
    Candidate candidate = $candidateList.last();
    modify($candidateList) { removeLast() }
    modify($nextAction) { setStatus(Constants.MAKE_OFFERS), addOffer(candidate) }
    modify(candidate) { setStatus(Constants.OFFERED) }
    modify($donor) { addOffer(candidate) }
    delete(candidate)
end
```

The first condition finds a donor we are still processing. The second conditions pick out either the heart or the lungs for the donor, while the third condition checks we can match both the donor and the organ against the offer response. We then check if the high priority list associated with the organ still has entries in it. The final condition simply checks we are not already about to offer this organ to a candidate (without this, we would end up offering the organ to all the high priority candidates at the same time). There are six actions executed in the then part of the rule, these are shared with the next two rules[9].

[9] Drools does provide the facility for creating 'functions' which ought to make it easy to provide a procedural abstraction of this and only write the actions out once. However, for reasons I can't really fathom, the complier for functions lacks the ability to recognise directives like 'modify' which makes the process more tedious. Frustrated I simply did a cut and paste to duplicate the code in three places

One thing to note is that this rule can fire two times in the initial call to the rules with a new donor, once for lungs and once for the heart. When responding to an offer it will only fire at most once.

The fourth rule in this group provides the switch between separately offering heart and lungs to offering a heart-lung block. It requires many conditions to be satisfied!

```
rule "Next Action - Make first offer of block from TC List"
  when
    $donor: Donor($donorId: donorId, $nextAction: nextAction,
      status == Constants.IN_PROGRESS, nextAction.status == Constants.NO_ACTION,
      !outstandingOffers())
    CandidateList($donorId == donorId, organId == Constants.HEART, candidates.size() == 0)
    CandidateList($donorId == donorId, organId == Constants.LUNGS, candidates.size() == 0)
    $candidateList: CandidateList($donorId == donorId, organId == Constants.CENTRE,
      candidates.size() > 0)
    $heart: Organ($donorId == donorId, organId == Constants.HEART,
       status != Constants.ACCEPTED)
    $lungs: Organ($donorId == donorId, organId == Constants.LUNGS,
       status != Constants.ACCEPTED)
    $block: Organ($donorId == donorId, status == Constants.UNAVAILABLE,
      organId == Constants.HEART_LUNG_BLOCK)
  then
    modify($heart) { setStatus(Constants.UNAVAILABLE) }
    modify($lungs) { setStatus(Constants.UNAVAILABLE) }
    modify($block) { setStatus(Constants.OFFERED) }
    Candidate candidate = $candidateList.last();
    modify($candidateList) { removeLast() }
    modify(candidate) { setOrganId(Constants.HEART_LUNG_BLOCK), setStatus(Constants.OFFERED) }
    modify($nextAction) {setStatus(Constants.MAKE_OFFERS), addOffer(candidate) }
    modify($donor) { addOffer(candidate) }
    delete(candidate)
end
```

We check firstly that we are not already about to make an offer (next action status is ‘no action’) and that there are no outstanding offers (**outstandingOffers** is a convenience method on the Donor class to check if there are any entries in the **currentOffers** set). If there are current offers, we cannot move on to offering from the TC list. The next conditions check that there are no candidates on either of the high priority list, but that there are TC list (as per assumptions above). Finally, we bind names to the three organs and check they are in the appropriate state for switching from single organ to offering a block. If all the conditions are satisfied, the ‘then’ part of the rule updates the status of the organs appropriately, and then performs the six actions described above.

The final ‘next action’ rule simply handles the next offer from the TC list. Most of the conditions are the same as in the preceding rule, we should have no outstanding offers, we should not have already decided to make an offer, the high priority lists should be empty, the TC list should not. The difference here is that instead of looking for both heart and lungs to be offered (and the block to be unavailable), is that this rule only applies when exactly one organ is under offer – this the meaning of the condition:

```
ArrayList( size == 1 )
             from collect( Organ( $donorId == donorId, status == Constants.OFFERED ) )
```

If all the conditions are satisfied, then we perform exactly the same actions as described for offers from the high priority list.

```
rule "Next Action - Make offer from TC List"
  when
    $donor: Donor($donorId: donorId, $nextAction: nextAction,
      status == Constants.IN_PROGRESS, nextAction.status == Constants.NO_ACTION,
      !outstandingOffers())
    CandidateList($donorId == donorId, organId == Constants.HEART, candidates.size() == 0)
    CandidateList($donorId == donorId, organId == Constants.LUNGS, candidates.size() == 0)
    $candidateList: CandidateList($donorId == donorId,
      organId == Constants.CENTRE, candidates.size() > 0)
    ArrayList( size == 1 )
              from collect( Organ( $donorId == donorId, status == Constants.OFFERED ) )
    $organ: Organ($donorId == donorId, $organId: organId, status == Constants.OFFERED)
  then
    Candidate candidate = $candidateList.last();
    modify($candidateList) { removeLast() }
    modify(candidate) { setOrganId($organId), setStatus(Constants.OFFERED) }
    modify($nextAction) {setStatus(Constants.MAKE_OFFERS), addOffer(candidate) }
    modify($donor) { addOffer(candidate) }
    delete(candidate)
end
```

The Tidy Up Rules

Because the state of the offering process is held in the working memory, after an **offerResponse** instance has been processed, there may be objects in memory which are no longer wanted, and which indeed may inappropriately trigger additional rules when we re-invoke the service. In particular, the **offerResponse** itself needs be removed. If the offer process is complete for the donor, the **donor** object and any remaining sub objects should also be removed. While the rules prevent further processing of a donor once its status is ‘complete’ or ‘fallback’ failure to delete them from the working memory effectively causes a memory leak. The rules operate at low salience, so we only tidy up when everything else is done.

The first tidy up rule will fire each time the engine is called and simply deletes any **OfferResponse** objects from the working memory (even if we are offering to multiple concurrent donors, the rule engine only processes one at a time, so any such objects are implicitly linked to the donor we are currently working with although many donor objects may exist).

```
rule "Tidy Up Offer Response"
  salience -100
  when
    $offerResponse : OfferResponse()
  then
    delete($offerResponse)
end
```

The two other tidy up rules kick in when the offering process for a donor completes. The first remove remaining **Candidate** instances from working memory which were at one time linked to the completed donor, the second walks the donor object tree removing the next action, the candidate lists, the organs and finally the donor itself.

```
rule "Tidy Up Candidates"
  salience -100
  when
    $donor: Donor($donorId: donorId,
      status == Constants.COMPLETE || == Constants.FALLBACK)
    $candidate : Candidate(donorId == $donorId)
  then
    delete($candidate)
```

```
end
```

Note that the final rule has lower salience that the one before. We need to ensure this is the last rule to fire, as when it does the donor is no longer in working memory and so we cannot no longer match other objects against its **donorId** attribute.

```
rule "Tidy Up Donor"
  salience -200
  when
    $donor: Donor($donorId: donorId, $nextAction: nextAction,
      status == Constants.COMPLETE || == Constants.FALLBACK)
  then
    delete($nextAction);
    for(Organ organ: $donor.getOrgans().values()) {
      delete(organ);
    }
    for(CandidateList candidateList: $donor.getCandidateLists().values()) {
      delete(candidateList);
    }
    delete($donor)
end
```

4 From Rules to a Service

4.1 A simple command line framework

How do we provide a framework to execute the rules, so we end up with a callable decision service? I had (well still have) hopes of wrapping things up into a web service, but I've not had time to build a proper interface. Still it's fairly simple to set up a command line interface to run things. Stripped of Drools boiler plate the main code looks something like this:

```
Donor donor = Utils.loadDonor(sc);
while (donor != null) {
  processDonor(kSession, donor, sc);
  donor = Utils.loadDonor(sc);
}
```

We 'load' a sample donor from a test library (using **loadDonor**) and go into a loop where we call a method making the offering for a donor (using **processDonor**), and then load another test case (or exit) as we wish. The test library stores sample donors as JSON documents on the disk and I have 8 sample scenarios which cover all the different combinations of which organs are available to offer and which high priority lists are populated. **loadDonor** simply uses the scenario number to select the correct JSON file and a Java/JSON library to read it in and build the donor object.

The full logic of **processDonor** looks a bit intimidating, but most of the code is about printing things to the console to let you know what's happening. Filleted of most of this code it looks as below. To start we set up a Boolean flag to indicate if we still have work to do and create a list to store which offers need to be responded too. We then make an initial call into Drools passing in the donor. This will trigger the initialise rules and will populate the donor's **NextAction** instance with the first offer(s) to be made.

We then move into a while loop. This starts by examining what the next action is. If it is 'complete' or 'fall back' (which it won't be on the first time through the loop, but will be eventually), we set the loop flag so we exit. If the next action is 'no action' we just

output a message to this effect and pause execution by forcing the user to make an input at the console. If none of these three situations apply the next action will be ‘make offers’, so we append the new offers to our list of offers to make.

At this point we should have at least one, and at most two offers to make and needing responses. There is a safety check to ensure this is true, but it will only be false if there is a bug in the code! We extract the first offer (actually, we could extract either, as if there are two they will be for different organs) and pass it off to another method (**processOneResponse**). This simply asks the user what the offer response should be (decline, full accept, partial accept), and if the last of these, which organ is being accepted, it then populates an **OfferResponse** instance appropriately with the response, a clone of the candidate information in the offer, and the accepted organ. It then inserts the **OfferResponse** into the Drools working memory and fires off the rule engine again. When this completes we move back up to the top of the while loop in **processDonor** and look at the next action again (since the next action is the outcome decision of the rules). Eventually all the organs will be accepted giving a ‘normal’ completion, or we will run out of candidates to offer things to and hit the fallback completion.

```
private static void processDonor(KieSession kSession, Donor donor, Scanner sc) {
  boolean notComplete = true;
  ArrayList<OfferResponse> offerResponses = new ArrayList<OfferResponse>();
  insertObjAndFireRules(kSession, donor);
  while (notComplete) {
    NextAction nextAction = donor.getNextAction();
    String type = nextAction.getStatus();
    if (type.equals(COMPLETE)) {
      notComplete = false;
    } else if (type.equals(FALLBACK)) {
      notComplete = false;
    } else if (type.equals(NO_ACTION)) {
      System.out.println("No action at this time please wait for next offer response!");
      sc.nextLine();
    }

    // add in any new offers from last call to rules
    offerResponses.addAll(Utils.getOffersToMake(nextAction, donor));

    // if we get here there should be some offers awaiting a response and processing
    if (notComplete && offerResponses.size() > 0) {
      OfferResponse offerResponse = offerResponses.remove(0);
      processOneResponse(kSession, donor, offerResponse, sc);
    } else if (notComplete) {
      System.out.println("************* WHOOPS! ******** ");
      notComplete = false;
    }
  }
}
```

4.2 Sample Output

Let’s look at some simple runs of the decision service. The output has several lines deleted to remove extra monitoring information and focus on the more interesting parts.

First let’s look at scenario 4 where we have a heart, two high priority candidates and two TCs. In the first run we simply decline all the offers

```
Input Scenario 1-8, type 0 to exit
4
Number of Rules executed = 6
```

```
*** There are 1 offers awaiting a response ***
***************** Give response for the offer to candidate:
DonorId: Petra CandidateId: Alvin, CentreId: T-C N, Status: NEW OrganId: HEART
Give type of response status (F)ULL_ACCEPT, (P)ARTIAL_ACCEPT, (D)ECLINED
d <------ offer declined
Proccessing: DonorId: Petra Status: DECLINE
Candidate: DonorId: Petra CandidateId: Alvin, CentreId: T-C N, Status: NEW OrganId: HEART
Number of Rules executed = 3

*** There are 1 offers awaiting a response ***
***************** Give response for the offer to candidate:
DonorId: Petra CandidateId: Bella, CentreId: T-C S, Status: NEW OrganId: HEART
Give type of response status (F)ULL_ACCEPT, (P)ARTIAL_ACCEPT, (D)ECLINED
d <------ offer declined
Proccessing: DonorId: Petra Status: DECLINE
Candidate: DonorId: Petra CandidateId: Bella, CentreId: T-C S, Status: NEW OrganId: HEART
Number of Rules executed = 3

*** There are 1 offers awaiting a response ***
***************** Give response for the offer to candidate:
DonorId: Petra CandidateId: T-C S, CentreId: T-C S, Status: NEW OrganId: HEART
Give type of response status (F)ULL_ACCEPT, (P)ARTIAL_ACCEPT, (D)ECLINED
d <------ offer declined
Proccessing: DonorId: Petra Status: DECLINE
Candidate: DonorId: Petra CandidateId: T-C S, CentreId: T-C S, Status: NEW OrganId: HEART
Number of Rules executed = 3

*** There are 1 offers awaiting a response ***
***************** Give response for the offer to candidate:
DonorId: Petra CandidateId: T-C W, CentreId: T-C W, Status: NEW OrganId: HEART
Give type of response status (F)ULL_ACCEPT, (P)ARTIAL_ACCEPT, (D)ECLINED
d <------ offer declined
Proccessing: DonorId: Petra Status: DECLINE
Candidate: DonorId: Petra CandidateId: T-C W, CentreId: T-C W, Status: NEW OrganId: HEART
Number of Rules executed = 4 <------ we fire different rules this time as we reach FALLBACK

Number of unresponded offers is 0
Offer Responses are as follows
       DonorId: Petra Status: DECLINE
Candidate: DonorId: Petra CandidateId: Alvin, CentreId: T-C N, Status: NEW OrganId: HEART
       DonorId: Petra Status: DECLINE
Candidate: DonorId: Petra CandidateId: Bella, CentreId: T-C S, Status: NEW OrganId: HEART
       DonorId: Petra Status: DECLINE
Candidate: DonorId: Petra CandidateId: T-C S, CentreId: T-C S, Status: NEW OrganId: HEART
       DonorId: Petra Status: DECLINE
Candidate: DonorId: Petra CandidateId: T-C W, CentreId: T-C W, Status: NEW OrganId: HEART
***** ENTER FALL BACK for HEART *****
Input Scenario 1-8, type 0 to exit
```

As should be expected, if none of the candidates accept the offer, we go into the fallback process. If we try again, but accept the organ for the second candidate we get the following:

```
Input Scenario 1-8, type 0 to exit
4
Number of Rules executed = 6

*** There are 1 offers awaiting a response ***
***************** Give response for the offer to candidate:
DonorId: Petra CandidateId: Alvin, CentreId: T-C N, Status: NEW OrganId: HEART
Give type of response status (F)ULL_ACCEPT, (P)ARTIAL_ACCEPT, (D)ECLINED
d <------ offer declined
Proccessing: DonorId: Petra Status: DECLINE
Candidate: DonorId: Petra CandidateId: Alvin, CentreId: T-C N, Status: NEW OrganId: HEART
Number of Rules executed = 3
```

```
*** There are 1 offers awaiting a response ***
***************** Give response for the offer to candidate:
DonorId: Petra CandidateId: Bella, CentreId: T-C S, Status: NEW OrganId: HEART
Give type of response status (F)ULL_ACCEPT, (P)ARTIAL_ACCEPT, (D)ECLINED
a <------ offer accepted (we can type 'a' or 'f' for a full accept)
Proccessing: DonorId: Petra Status: FACCEPT
Candidate: DonorId: Petra CandidateId: Bella, CentreId: T-C S, Status: NEW OrganId: HEART
Accepted Organ: HEART
Number of Rules executed = 6<------ we fire different rules this time as we COMPLETE

Number of unresponded offers is 0
Offer Responses are as follows
       DonorId: Petra Status: DECLINE
Candidate: DonorId: Petra CandidateId: Alvin, CentreId: T-C N, Status: NEW OrganId: HEART
       DonorId: Petra Status: FACCEPT
Candidate: DonorId: Petra CandidateId: Bella, CentreId: T-C S, Status: NEW OrganId: HEART
Accepted Organ: HEART
***** ORGAN PROCESSING COMPLETE ******
Input Scenario 1-8, type 0 to exit
```

And as a final example, scenario 8, the 'big one' with both organs, each with two high priority candidates. We explore what happens if none of the high priority candidate accept an offer, the first TC offered a heart-lung block and accepts the heart, and then the second TC is offered the lungs which it accepts. The key points in the scenario are:

- After the second high-priority candidate on the heart list ('Bella') declines the offer, the heart must 'wait' to see what happens with the outstanding offer to 'Edgar' for the lungs, so the next action is 'no action'
- After the second high-priority candidate on the lungs list ("Edgar") is declined, the heart and lungs stop being offered, and instead a heart-lung block is offered
- After the heart is accepted by T-C South, only the lungs are offered to T-C West

```
Input Scenario 1-8, type 0 to exit
8
Number of Rules executed = 11

*** There are 2 offers awaiting a response *** <------ one for the heart one for the lungs
***************** Give response for the offer to candidate:
DonorId: Tilly CandidateId: Alvin, CentreId: T-C N, Status: NEW OrganId: HEART
Give type of response status (F)ULL_ACCEPT, (P)ARTIAL_ACCEPT, (D)ECLINED
d<------ offer declined
Proccessing: DonorId: Tilly Status: DECLINE
Candidate: DonorId: Tilly CandidateId: Alvin, CentreId: T-C N, Status: NEW OrganId: HEART
Number of Rules executed = 3

*** There are 2 offers awaiting a response *** <------ one for the heart one for the lungs
***************** Give response for the offer to candidate:
DonorId: Tilly CandidateId: Diane, CentreId: T-C W, Status: NEW OrganId: LUNGS
Give type of response status (F)ULL_ACCEPT, (P)ARTIAL_ACCEPT, (D)ECLINED
d<------ offer declined
Proccessing: DonorId: Tilly Status: DECLINE
Candidate: DonorId: Tilly CandidateId: Diane, CentreId: T-C W, Status: NEW OrganId: LUNGS
Number of Rules executed = 3

*** There are 2 offers awaiting a response *** <------ one for the heart one for the lungs
***************** Give response for the offer to candidate:
DonorId: Tilly CandidateId: Bella, CentreId: T-C S, Status: NEW OrganId: HEART
Give type of response status (F)ULL_ACCEPT, (P)ARTIAL_ACCEPT, (D)ECLINED
d<------ offer declined
Proccessing: DonorId: Tilly Status: DECLINE
Candidate: DonorId: Tilly CandidateId: Bella, CentreId: T-C S, Status: NEW OrganId: HEART
Number of Rules executed = 2
```

```
Number of unresponded offers is 1 <------ ie an offer for the lungs is outstanding
No action at this time please wait for next offer response!
hit return to continue<------ the heart must now 'wait' to see what happens to the lungs

*** There are 1 offers awaiting a response *** <------ for the lungs
***************** Give response for the offer to candidate:
DonorId: Tilly CandidateId: Edgar, CentreId: T-C S, Status: NEW OrganId: LUNGS
Give type of response status (F)ULL_ACCEPT, (P)ARTIAL_ACCEPT, (D)ECLINED
d<------ offer declined – at this point we move to offering the heart-lung block
Proccessing: DonorId: Tilly Status: DECLINE
Candidate: DonorId: Tilly CandidateId: Edgar, CentreId: T-C S, Status: NEW OrganId: LUNGS
Number of Rules executed = 3

*** There are 1 offers awaiting a response *** <------ this is for the heart-lung block
***************** Give response for the offer to candidate:
DonorId: Tilly CandidateId: T-C S, CentreId: T-C S, Status: NEW OrganId: HLBLK
Give type of response status (F)ULL_ACCEPT, (P)ARTIAL_ACCEPT, (D)ECLINED
p<------ partial accept so we need to say which organ has been accepted
Give organ partially accepted of response status ({HEART, LUNGS)
h<------ the heart is accepted so next offer will be of the lungs
Proccessing: DonorId: Tilly Status: PACCEPT
Candidate: DonorId: Tilly CandidateId: T-C S, CentreId: T-C S, Status: NEW OrganId: HLBLK
Accepted Organ: HEART
Number of Rules executed = 3

*** There are 1 offers awaiting a response ***  <------ for the remaining organ (the lungs)
***************** Give response for the offer to candidate:
DonorId: Tilly CandidateId: T-C W, CentreId: T-C W, Status: NEW OrganId: LUNGS
Give type of response status (F)ULL_ACCEPT, (P)ARTIAL_ACCEPT, (D)ECLINED
a<------ offer accepted, so processing is complete
Proccessing: DonorId: Tilly Status: FACCEPT
Candidate: DonorId: Tilly CandidateId: T-C W, CentreId: T-C W, Status: NEW OrganId: LUNGS
Accepted Organ: LUNGS
Number of Rules executed = 4

Number of unresponded offers is 0
Offer Responses are as follows
       DonorId: Tilly Status: DECLINE
Candidate: DonorId: Tilly CandidateId: Alvin, CentreId: T-C N, Status: NEW OrganId: HEART
       DonorId: Tilly Status: DECLINE
Candidate: DonorId: Tilly CandidateId: Diane, CentreId: T-C W, Status: NEW OrganId: LUNGS
       DonorId: Tilly Status: DECLINE
Candidate: DonorId: Tilly CandidateId: Bella, CentreId: T-C S, Status: NEW OrganId: HEART
       DonorId: Tilly Status: DECLINE
Candidate: DonorId: Tilly CandidateId: Edgar, CentreId: T-C S, Status: NEW OrganId: LUNGS
       DonorId: Tilly Status: PACCEPT
Candidate: DonorId: Tilly CandidateId: T-C S, CentreId: T-C S, Status: NEW OrganId: HLBLK
Accepted Organ: HEART
       DonorId: Tilly Status: FACCEPT
Candidate: DonorId: Tilly CandidateId: T-C W, CentreId: T-C W, Status: NEW OrganId: LUNGS
Accepted Organ: LUNGS
***** ORGAN PROCESSING COMPLETE ******
Input Scenario 1-8, type 0 to exit
```

5 Discussion

5.1 Stateful vs Stateless

What do we actually mean by saying a decision stateless as compared to a stateful? The general interpretation seems to be that if you ask a 'stateless' system the same question more than once you always get the same answer. It's fair to say that looking at the whole start-to-end aspects of making a decision, almost all (automated) decision systems are stateful to at least some extent, because most depend on immediate case

data and persistent but slowly changing data. Consider a loan application. If I fill in my application today, I may get offered a £5000 at 5% APR. If I do it tomorrow updates at the credit bureau overnight, it may be that I now have a lower credit rating and I might instead be offered only £4500 at 5.5% APR. This despite the overall system having received exactly the same data from me as it did the day before.

In most cases we gloss over this by passing the persistent data together with the immediate case data as input, so while the full end-to-end system is stateful, the core decisioning component itself need not be.

To make progress in the discussion I think it's better to turn the question around a bit and ask, under what circumstances we should view a decision as 'stateful' and it seems to me we do this when a couple of things are happening.

Firstly, we almost always need to invoke the decision system multiple times in order get a decision. Usually these invocations take place in fairly quick succession ('quick' only signifyingly that we generally ignore the possibility of the slowly changing persistent data changing during these repeated calls). Furthermore, these multiple invocations may come from different sources (though this is not the case in the challenge).

Secondly, the effect of making an invocation will enriches the state in some way so each invocation potentially changes the behaviour of the system when we invoke it subsequently. For example, consider the final sample output in the previous section. If instead of declining the offer of the lungs to Edgar the TC accepts it, then the next offer will be of the heart not the heart-lung block. The challenge is just sufficiently complex to call for a solution with both these characteristics. It is important to notice that not all stateful systems give a 'final' decision no matter how many times they are invoked. In the challenge and in dialog systems like on-line medical diagnosis we will get to a final conclusion, but there are 'always on' systems which only give a description of the current state of affairs, while continuously evaluating new information and updating this description as time goes by.

For a genuinely stateful problems which do have a definitive end point, like the one posed by the challenge, we can often get away with using a stateless engine. All we need to do is to treat the state as an output as well as an input. Then when the updated state emerges as the result of the running the decision system, we can store it away until it is needed and then bring it back as an input again to the next cycle of the decision process[10]. This approach means we end up with a layered decision system having an inner stateless component, wrapped by an outer stateful one.

Some systems are so intrinsically stateful this approach really makes no sense. The obvious examples those which can be reasonably categorised as Complex Event Processing (CEP) systems and can have states comprising of thousands of objects. It's an expensive operation to load and save these, every time the system activates, and since CEPs generally need to be at least 'soft' real time in performance it is not going to be possible to meet the kind of timing constraints required. Also, most such systems need to be 'always on' – they are monitoring systems. Rather than asking them for

[10] I'm actually in the process of building a second solution using Drools in 'stateless' mode rather than 'stateful' mode based on this approach of treating the 'state' as output as well as input and saving and restoring it outside the rule engine.

decisions, they proactively generate decisions for us, telling us everything is okay, or something is wrong.

The organ offering challenge is a much-simplified version of the real NHSBT problem, but even so for the 'full' problem the 'state' itself is not likely to get much more complicated that the one used in this solution The idea of offering organs to candidates and processing responses to such offers is pervasive, and the domain model supports these ideas fairly fully. The additional complications of a more complete solution would need a lot more rules, because what you offer, when you offer it and to whom you offer it differs for each of the many different organs which can be donated. But for any given donor the state would likely remain quite modestly sized. From past experience, I'd judge systems handing mortgage applications involve more data. So, for this problem managing the storage of state between successive offers outside the rule engine is never likely to be a very heavy burden.

A stateful system where the state is managed externally to a core stateless component also means you can accrue the other advantages the stateless approach gives. The 'full' stateful approach keeps everything in memory, which makes things like load balancing trickier and makes recovery of ongoing activities virtually impossible if the server running the system goes down. In a system where the 'state' is stored externally in a resilient persistent store (i.e. a database), you can take advantage of enterprise level services for load-balancing and automatic fail-over on server crashes, Following, a switch over an ongoing offer process just loads the state from the database and proceeds pretty much as if nothing happened.

5.2 DMN and Stateful Problems?

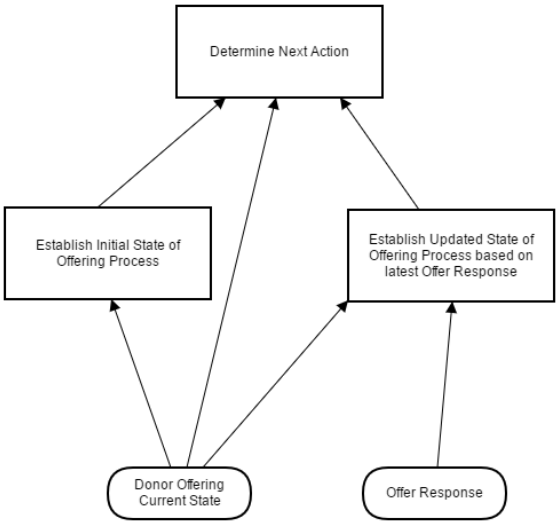
The DMN specification states clearly that it only considers 'stateless' decision (services)[11]. However, I'm not really sure how much of the DMN specification actually forms a significant barrier to using it for stateful applications. Certainly, once you are 'inside' a decision service, with most systems a decision table or a boxed expression isn't really going to care much if you are updating data from the 'state' component, or from the specific inputs provided for this invocation of the service or even distinguish between the two[12]

In my opinion, the first step in enabling us to extend DMN to cope with stateful systems is to be find a way to model them – which for me means looking at what we can express in a DRD. Worrying about other aspects like decision tables, boxed expressions, and how we build composite groups of individual decisions is a bit pointless unless we can cross this hurdle. There is a specific limitation in DRDs that data sources are read only, you only get arrows leaving a data source, never ones entering it (i.e. which would indicate a decision was updating the source as opposed to the source providing data to a decision). So, to address stateful system we would need to add a bit to DRDs – but possibly not much. As 'modest proposal' for DRDs for the first steps in modelling 'stateful' decisions consider the following DRD for the challenge solution above:

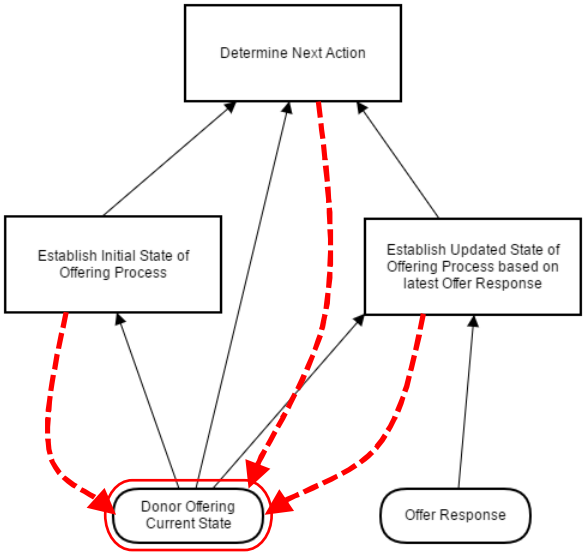
---

[11] See section 5.3.3 in https://www.omg.org/spec/DMN/1.2/PDF

[12] This is not universally true. Consider for example Sapiens Decision, https://www.sapiensdecision.com/. This is based on 'The Decision Model' (TDM) and in many respects well aligned with DMN but it strictly limits reassignments of values as this breaks mandated integrity and consistence constraints. In essence inputs cannot also be outputs within the constraints of TDM.



Now simply add some more arrows (preferable distinctive in nature) and perhaps some highlighting to indicate we are dealing with an input which represents persistent but mutable state, and we might get the following diagram:



There are probably better ways of doing this, but we need to start the conversation to come up with them. It at least gives us a start. Next would be to see where it would take us, which would mean looking at some real stateful problems, and probably ones rather meatier than the one considered here.

6 Source Code

In the past I've generally included the source of the Java code and Rules files with my challenge submissions, but it adds considerable to the length, and I'm not sure who's really that interested, so this time I've omitted them. If you want to see all the gory details drop me a line and I'll be happy to e-mail over a copy.