Veda Workflow Language

Бушенев В.А., Карпов Р.К. Мельников В.
ООО “Смысловые Машины”
Россия. Республика Коми. Сыктывкар.

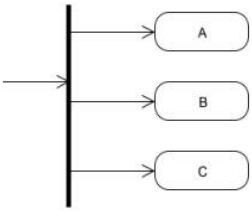
Аннотация. После анализа существующих языков моделирования потоков работ (workflow), мы остановились на таком замечательном языке как YAWL. Язык YAWL создан на основе математическом аппарате сетей Петри, однако для преодоления ограничений последних, был обогащен дополнительными механизмами по управлению потоками и преобразованию данных. Созданная спецификация в YAWL представляется в XML формате. YAWL не представлен как самостоятельный язык, а базируется на комплексе включающем в себя следующие компоненты: редактор сетей, движок исполнения и хранилище. Такой подход хорош своей комплексностью. Однако, в силу технических ограничений, а также применения RDF для описания и хранения онтологий в Veda, мы не имели возможности применить это решение для разработанной нами платформы. Поэтому, мы пошли путем создания своего языка. Язык VWL (**V**eda **W**orkflow **L**anguage) внешне напоминает YAWL, заимствует некоторые его идеи и термины, имеет графический редактор сетей, среду исполнения и хранения. В этом документе приводится описание языка и его элементов, как в графическом виде, используя визуальные примитивы, так и в семантическом виде, с помощью одного из форматов описания онтологии.

Введение.

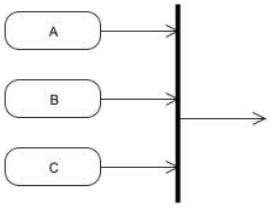
Часто встречающиеся конструкции по управлению потоками работ можно назвать шаблонами/паттернами. В настоящее время считается, что для описания большинства бизнес задач достаточно 20 различных шаблонов. Ниже приведено описание наиболее распространенных шаблонов с их графическими образами в формате UML. (1)

*Шаблон **Последовательность** (последовательная маршрутизация)*
Простейший «элемент» бизнес-процесса: Два узла соединены переходом. После того, как исполнитель выполнил действие первого узла, управление переходит ко второму.

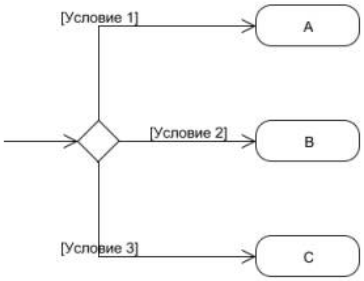
*Шаблон **Параллельное расщепление** (разветвитель, параллельная маршрутизация, И-расщепление)*
Представляет собой узел, в который приходит только один переход, и из которого исходит два или более переходов. Причем, после того, как управление передано узлу, поток управления бизнес-процессом распадается на несколько потоков, которые выполняются параллельно. Для каждого исходящего перехода должен существовать «свой» поток управления.



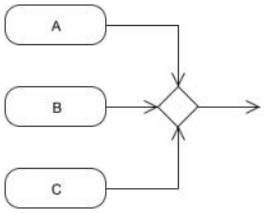
*Шаблон **Синхронизация** (И-объединение, рандеву, синхронизатор)*
Узел, в котором соединяются два или более перехода, а выходит только один. Управление не будет передано дальше, пока все потоки управления бизнес-процессом (по количеству входящих переходов) не достигнут данного узла. После того, как в узел придут все потоки управления, будет инициирован только один поток, соответствующий исходящему переходу.



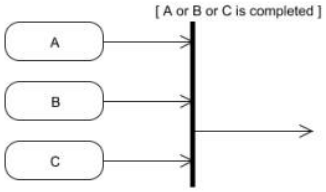
*Шаблон **Эксклюзивный выбор** (XOR-расщепление, условная маршрутизация, выбор, решение)*
Узел в графе WF-процесса, в который приходит только один, и из которого исходит два или более переходов. Причем, после того, как управление перешло к данному узлу, в нем делается выбор, по какому из исходящих переходов управление будет передано далее. Паттерн часто используется в «связке» с паттерном «простое соединение».



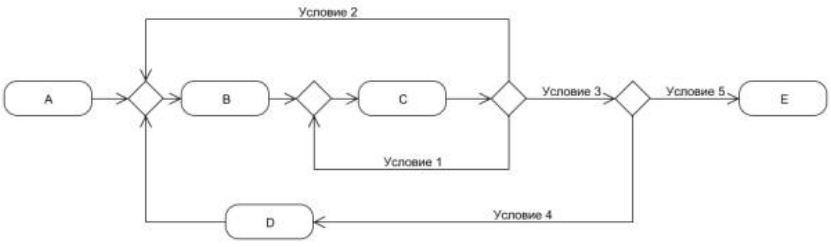
*Шаблон **Простое соединение** (XOR-объединение, асинхронное объединение, соединение)*
Узел в графе WF-процесса, в котором соединяются два или более перехода, а выходит только один. После того, как в узел пришло управление от любого из входящих потоков, поток управления передается на единственный исходящий переход. Предполагается, что управление может прийти в узел только по одному из входящих переходов.



*Шаблон **Дискриминатор***
Узел в графе WF-процесса, в котором соединяются два или более перехода, а выходит только один переход. Как только узлу передается управление (по любому из входящих переходов), сразу активизируется исходящий переход, то есть начинает выполняться следующее действие бизнес-процесса. Выполнение других активных потоков не прерывается, однако передача управления каждого из этих потоков в узел дискриминатора игнорируется, и пришедший поток завершает свое существование. Данный паттерн часто используется в связке с паттерном «Параллельное расщепление».



*Шаблон **Произвольные циклы** (петли, итерация, цикл)*
Набор узлов в графе WF-процесса, в котором один или несколько узлов могут обходиться многократно, то есть - в эти узлы много раз может входить (и, соответственно выходить) управление. Причем, в отличие от регулярных циклов, точки «входов» в набор повторяющихся узлов (или «выходов» из них) могут быть различными.



*Шаблон **Отложенный выбор***
Узел в графе workflow-процесса, в который приходит только один, и из которого исходит два или более переходов. После прохождения узла, поток управления бизнес-процессом распадается на несколько потоков, число которых равно количеству исходящих переходов. Далее все потоки, соответствующие исходящим переходам становятся активными. Однако, после того, как первый узел для исполнения выбран, активным остается только поток этого узла, а все остальные принудительно завершаются. Паттерн часто используется в связке с паттерном «Простое соединение».

Большая часть из этих шаблонов по управлению потоком работ может быть описана с помощью VWL.

Реализация данного языка является надстройкой системы Veda, самостоятельное функционирование вне ее не предусмотрено. Созданная на языке VWL спецификация потока работ называется сетью. Сеть представляет собой набор индивидов хранящихся в системе Veda. С помощью графического редактора доступно визуальное моделирование сети. Созданная сеть может быть исполнена в среде системы Veda, с помощью программной надстройки Veda Workflow Engine. Агентами выполняющими задания, рожденные в рамках интерпретации сети, могут быть как пользователи системы, так и программные скрипты.

* использование данной документации требуется изучения базовых понятий системы Veda.
* примеры индивидов описываются в формате Terse RDF Triple Language
* онтология VWL + VWE описана в http://semantic-machines.com/veda/veda-workflow

Veda Workflow Language

Терминология и графическое обозначение.

v-wf:Net - сеть, состоит из элементов сети, поток выполнения направлен от *v-wf:InputCondition* к *v-wf:OutputCondition.*

Элементы сети:

 *v-wf:Condition* - узел, может иметь входящие и исходящие потоки

 *v-wf:InputCondition* - стартовый узел, с которого начинается выполнение сети

 *v-wf:OutputCondition* - в этом узле заканчивается исполнение сети

 *v-wf:Flow* - поток, имеет направление и может иметь условие для перехода к узлу либо задаче

 *v-wf:Task* - задача, служит для описания задания для агентов, а также описывает условия обработки результатов работы агентов

Возможные модификации задачи по управлению потоками:

 *XOR разделение* потоков

 *AND разделение* потоков

 *OR разделение* потоков

 XOR *слияние* потоков

 AND *слияние* потоков

 OR *слияние* потоков

Структуры описывающие поведение элементов сети:

- *v-wf:VarDefine* - переменная, имеет имя и может иметь указание области видимости, в рамках сети может быть входящей, исходящей, либо локальной.
- *v-wf:Mapping* - конструкция описывающая как следует заполнять переменную
- *v-wf:ExecutorDefinition* - задает исполнителей для указанной задачи
- *v-wf:Transform* - структура из набора правил преобразования массива индивидов в новый массив индивидов.

VWL в примерах.

В следующих примерах будет описано поэтапное создание сети от простой к сложной.

Пример 1. Самая простая сеть, ничего полезного не делает, шаблон Последовательность.

Создание сети.
Главное меню -> Создать, в открывшемся шаблоне, в поле тип выбрать “Сеть”, и выбрать из списка элемент <Сеть>. Далее следует соединить InputCondition и OutputCondition потоком (стрелочка --->).
Зададим имя сети в поле [наименование]: “пример сети 1”.

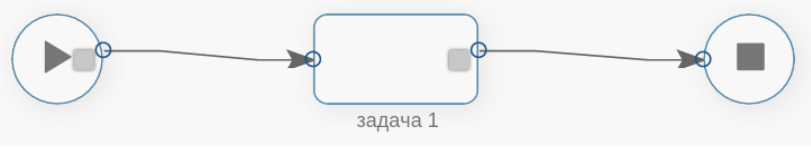
Первая наша сеть готова. Для запуска сети потребуется стартовая форма. Главное меню -> Создать, в открывшемся шаблоне, в поле тип набрать “стартовая” и выбрать из списка <Стартовая форма>,
в поле [Для сети...] найти нашу сеть “пример сети 1”, далее в поле [статус документооборота] внести значение <Ожидает отправки>. После нажатия на кнопку сохранить, наша сеть будет запущена на исполнение.

Как проверить результаты исполнения: Главное меню -> Полнотекстовый поиск. Нажать “Расширенный поиск”, снять ограничение типа “Пользовательская сущность”, в верхнее поле ввести поисковый запрос 'rdf:type' == 'v-wf:Process'. Далее появится список найденных экземпляров запущенных процессов. Найдем наш - “экземпляр маршрута :пример сети 1”, и откроем его:

OutputCondition окрашен в красный цвет, это означает что процесс по этой сети был выполнен.

Пример 2. Сеть с одной пустой задачей, так-же ничего полезного не делает.

Создадим новую сеть аналогично примеру 1, однако в новую сеть мы добавим задачу. В новой сети будет два потока: из InputCondition в [задача 1], и из [задача 1] в OutputCondition.



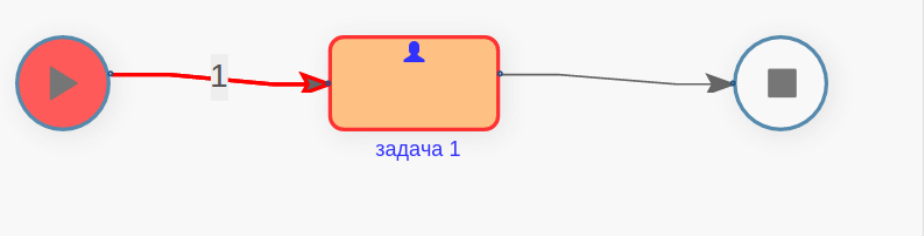
Запустим сеть аналогично примеру 1:

Видно, что исполненная задача отмечена зеленым цветом. Если выделить какой-либо из элементов сети, то в правой панели будет подробная информация о состоянии исполнения элемента сети.

Пример 3. Сеть, в которой исполнителем указан сотрудник.

Создадим новую сеть аналогично Примеру 2. Укажем в качестве [исполнителя] конкретного человека: Андрей Бычин: Аналитик

Запустим сеть:



Видно, что выполнение в отличие от Примера 2, остановилось на задаче 1.
Однако, если мы поищем задачи, которые должны прийти сотруднику Андрею, там ничего не будет. Это связано с тем, что движок workflow довольно абстрактен и ничего не знает о том, как должны выглядеть формы задач (v-wf:DecisionForm), на которые должен отвечать пользователь. А пустую форму движок не умеет создавать. В данной ситуации можно и вручную продвинуть исполнение сети, но это потребует глубоких знаний о внутренностях движка. Это будет описано ниже в разделе *“Veda Workflow Engine, как это работает.”*

Пример 4. Сеть, которая выдает задание сотруднику.

Для того чтобы создать форму ответа на задачу, нам потребуется структура *v-wf:Transform* (Трансформация). С помощью *v-wf:Transform* мы зададим правила трансформации, которые сформируют для нас пользовательскую форму ответа на задачу.

Загрузим в систему следующий фрагмент онтологии:

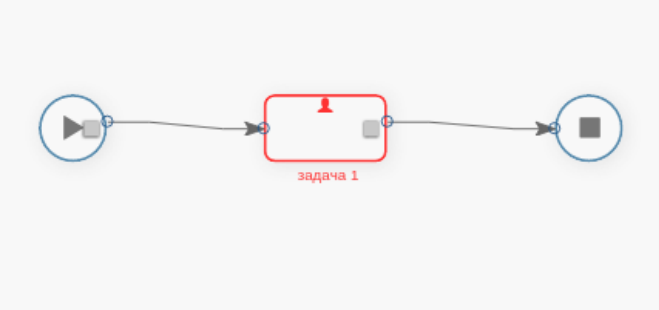
```
:net4-tr1(id генерируется)
   rdf:type v-wf:Transform ; (Тип)
   rdfs:label "создание формы ответа на задачу, net4"^^xsd:string;(Наименование)
   v-wf:transformRule :net4-tr1-r1 ; (Правило преобразования)
.

:net4-tr1-r1
   rdf:type v-wf:Rule ;(Тип)
   v-wf:segregateElement "contentName('@')" ;(Выражение для выделения подмножества элементов)
   v-wf:aggregate    "putUri ('rdf:type', 'v-wf:DecisionForm')";     (Выражение для размещения элемента в объекте)
   v-wf:aggregate    "putUri ('rdf:type', 's-wf:UserTaskForm')";
   v-wf:aggregate    "putString ('rdfs:label', 'задание')";
   v-wf:aggregate    "putBoolean ('v-wf:isCompleted', false)";
   v-wf:aggregate    "putExecutor ('v-wf:to')";
   v-wf:aggregate    "putWorkOrder ('v-wf:onWorkOrder')";
   v-wf:aggregate    "putUri ('v-wf:possibleDecisionClass', 'v-wf:DecisionAchieved')";
   v-wf:aggregate    "putUri ('v-wf:possibleDecisionClass', 'v-wf:DecisionNotPerformed')";
.
```

Индивид типа v-wf:Transform, содержит одно или несколько правил преобразования данных v-wf:Rule. Когда подойдет время выполнения элемента “*задача 1”*. Движок проверит атрибут [*трансформация для создания формы решения*], и при наличии правил преобразования данных, выполнит указанную трансформацию. На вход трансформации будет подан массив переменных исполняемой задачи. Так как мы сами не задавали никаких переменных в сети или задаче, массив будет содержать только одну переменную *curTask*, которую создает сам движок workflow.

Поле [*v-wf:segregateElement*] содержит js выражение, которое отфильтрует только одно поле ‘@’ из всех полей содержащихся в переменной curTask. Далее будут выполнены все выражения из поля [*v-wf:aggregate*]. Результатом их исполнения будет индивид типа v-wf:DecisionForm, так-же движком будут

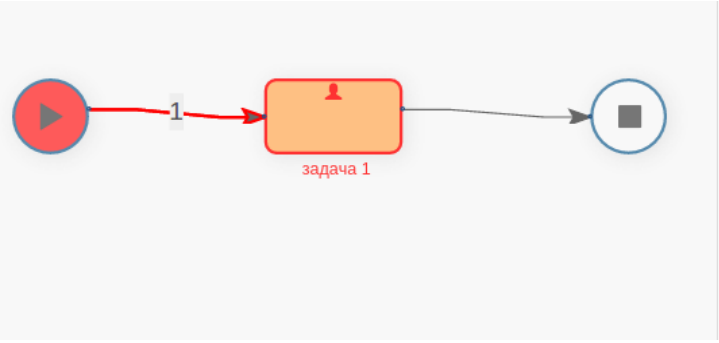
выданы права для исполнителя на изменение индивида задачи. Более подробно о преобразовании данных в разделе Трансформация: как это работает.

Ссылку на :net4-tr1, впишем в элемент сети *задача 1*, в поле [*трансформация для создания формы решения*](В поле “Создание задачи” наименование трансформации)



Элемент сети	**td:net4-t1**
объединение	None
разветвление	None
трансформация для создания формы решения	создание формы ответа на задачу, net4
исполнитель	Андрей Бычин

Запустим сеть:



выполняемая задача

df71b2f27f3ad5cb2c163d55b4a66b623

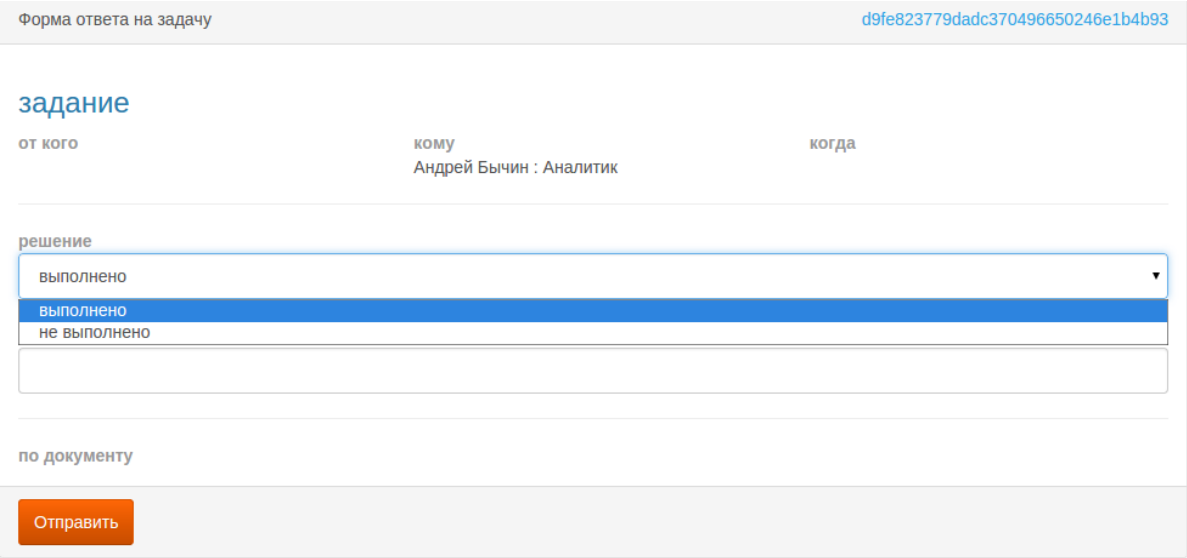
список заданий для исполнителей

1. **dcd0127296eb25a5d271a200c8373bc82**
   исполнитель
   - Сотрудник : Андрей Бычин : Аналитик

   список порожденных пользовательских форм
   - Форма решения Форма ответа на задачу : задание

Видно, что в правой информационной панели появилась ссылка на созданную пользовательскую форму.

Эту форму можно найти во входящих задачах сотрудника Андрея. Выглядит она так:



Пример 5. Создание формы задачи, с указанием кому и от кого пришла задача.

Первое, что нам понадобится, это маппинг переменной initiator:

```
:test-map-dec-initiator
  rdfs:label "Маппинг decision initiator"@ru;
  rdf:type v-wf:Mapping ;
```

```
  v-wf:mapToVariable d:var_dec_initiator ;
  v-wf:mappingExpression  "process.getInputVariable ('initiator')" ;
  rdfs:isDefinedBy s-wf: .
```

Для выделения отправителя из стартовой форм применим следующую трансформацию:

```
:test-tr1
    rdf:type v-wf:Transform ;
    rdfs:label "пробная трансформация"^^xsd:string;
    v-wf:transformRule :test-tr1-rule-initiator, :test-tr2-rule-docid, :test-tr3-rule-tag.
```

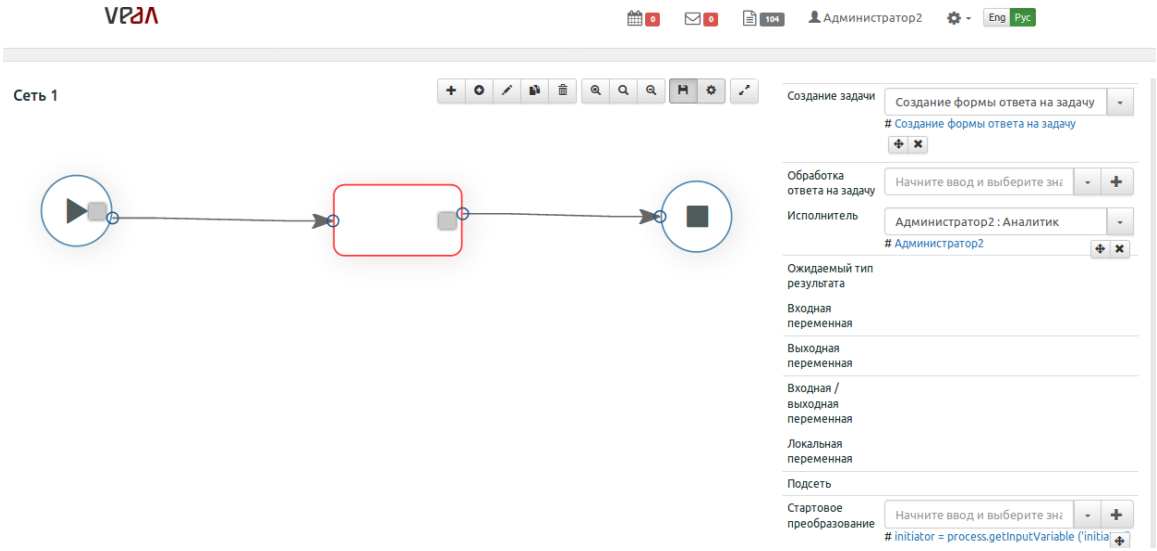
В этом примере трансформация для создания формы будет состоять из двух преобразований, переводящих множество входных индивидов в один выходной индивид. Преобразование выглядит следующим образом:

```
:test-tr-dec-form
  rdf:type v-wf:Transform ;
  rdfs:label "Создание формы ответа на задачу"^^xsd:string;
  v-wf:transformRule :test-tr-dec-form-rule1, :test-tr-dec-form-rule2.
```

Каждое из правил обрабатывает отдельную группу объектов. Первое преобразование выделяет, кто исполнитель задачи и создает элементы формы, второе – кто отправитель. Поле v-wf:grouping обозначает признак, по которому группируются индивиды. Это может быть как просто текст, так и функция.

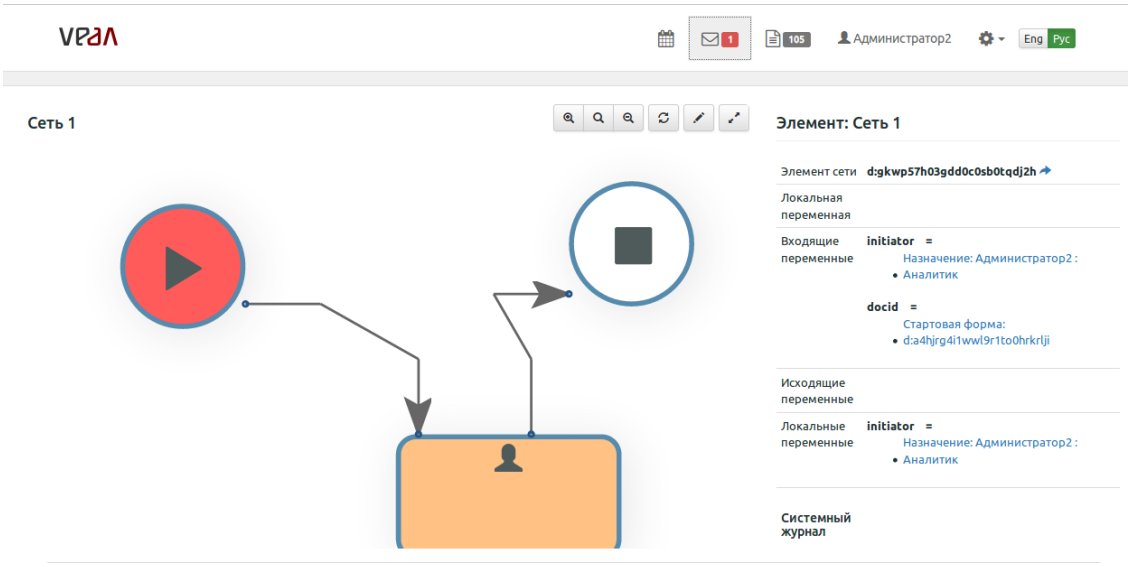
```
:test-tr-dec-form-rule1
  rdf:type v-wf:Rule ;
  v-wf:segregateObject "objectContentStrValue('v-wf:variableName', 'initiator')==false" ;
  v-wf:segregateElement "contentName('@')" ;
  v-wf:aggregate "putUri ('rdf:type', 'v-wf:DecisionForm')";
  v-wf:aggregate "putUri ('rdf:type', 's-wf:UserTaskForm')";
  v-wf:aggregate "putString ('rdfs:label', 'задание')";
  v-wf:aggregate "putBoolean ('v-wf:isCompleted', false)";
  v-wf:aggregate "putExecutor ('v-wf:to')";
  v-wf:aggregate "putWorkOrder ('v-wf:onWorkOrder')";
  v-wf:aggregate "putUri ('v-wf:possibleDecisionClass', 'v-wf:DecisionAchieved')";
  v-wf:aggregate "putUri ('v-wf:possibleDecisionClass', 'v-wf:DecisionNotPerformed')";
  v-wf:grouping "1".

:test-tr-dec-form-rule2
  rdf:type v-wf:Rule ;
  v-wf:segregateObject "objectContentStrValue('v-wf:variableName', 'initiator')" ;
  v-wf:segregateElement "contentName('v-wf:variableValue')" ;
  v-wf:aggregate  "putValue ('v-wf:from')";
  v-wf:grouping "1".
```

Создадим простую сеть с одной задачей, назначим исполнителем пользователя Администратор2: Аналитик. В стартовом преобразовании добавим наш маппинг, и в поле создание задачи добавим наше преобразование для создания формы.

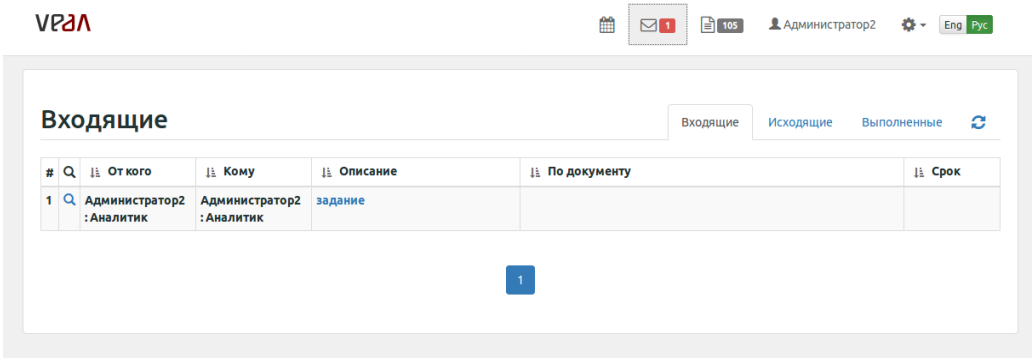


Теперь создадим стартовую форму для этой сети, укажем для неё трансформацию, установим статус документооборота в "Ожидает отправки" и запустим процесс.

Откроем запущенный маршрут, видим теперь, что задача отправлена и ожидает выполнения.

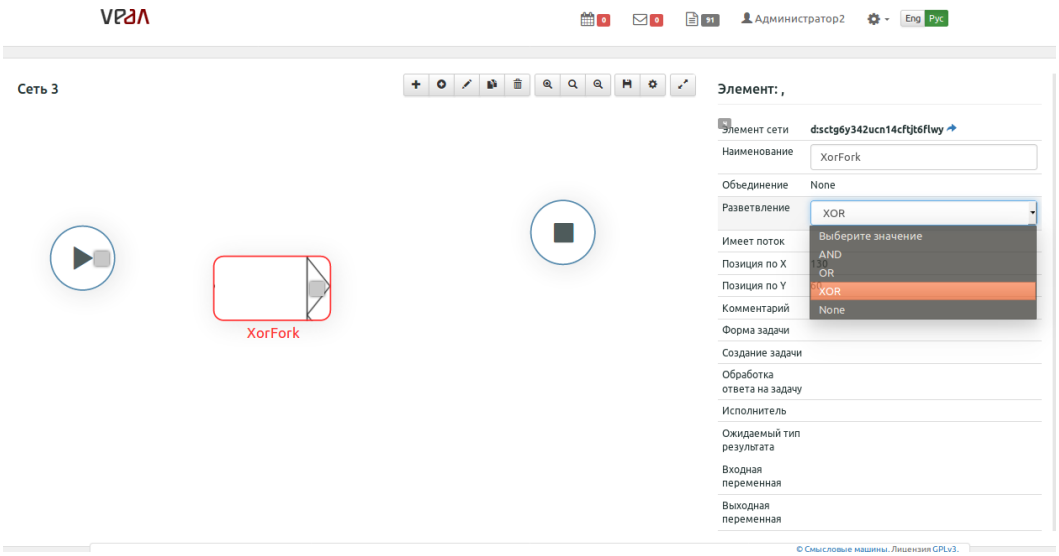


В папке входящих задач появится новая задача, теперь поле отправитель тоже заполнено, так как задачу мы отправили сами себе, то поля "от кого" и "кому" совпадают.

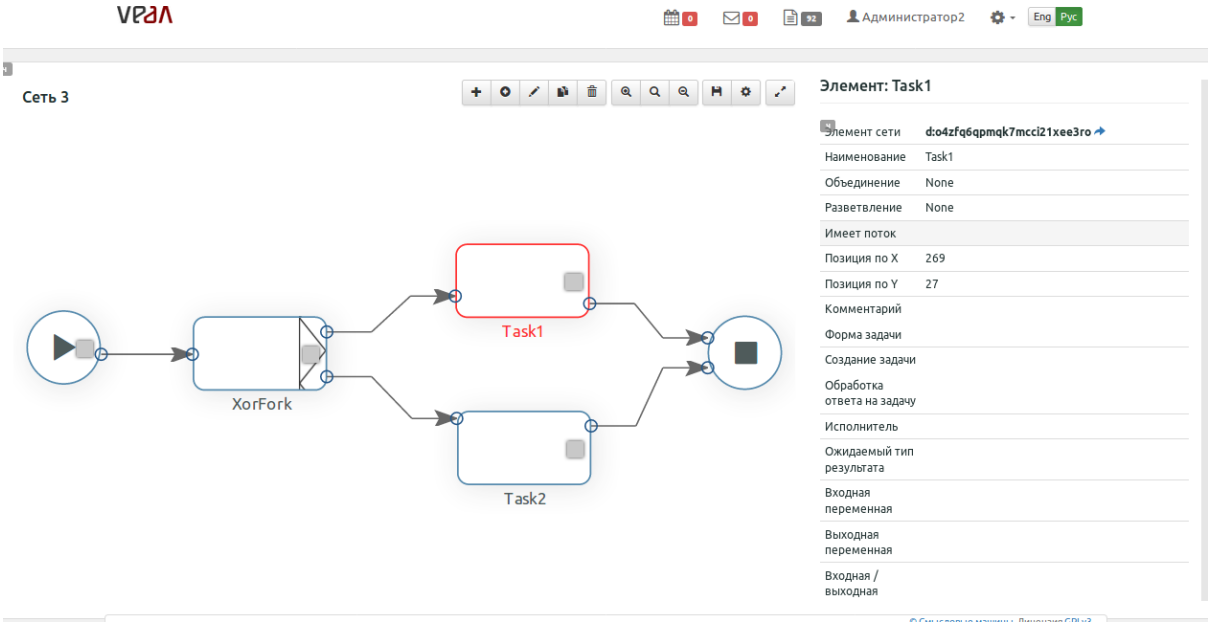


Пример 6. Сеть с разветвлением.

Для рассмотрения примера с разветвлением нужно создать более сложную сеть.
Первым шагом добавим задание, и сделаем его XOR разветвлением:



Добавим два простых задания и соединим сеть потоком:

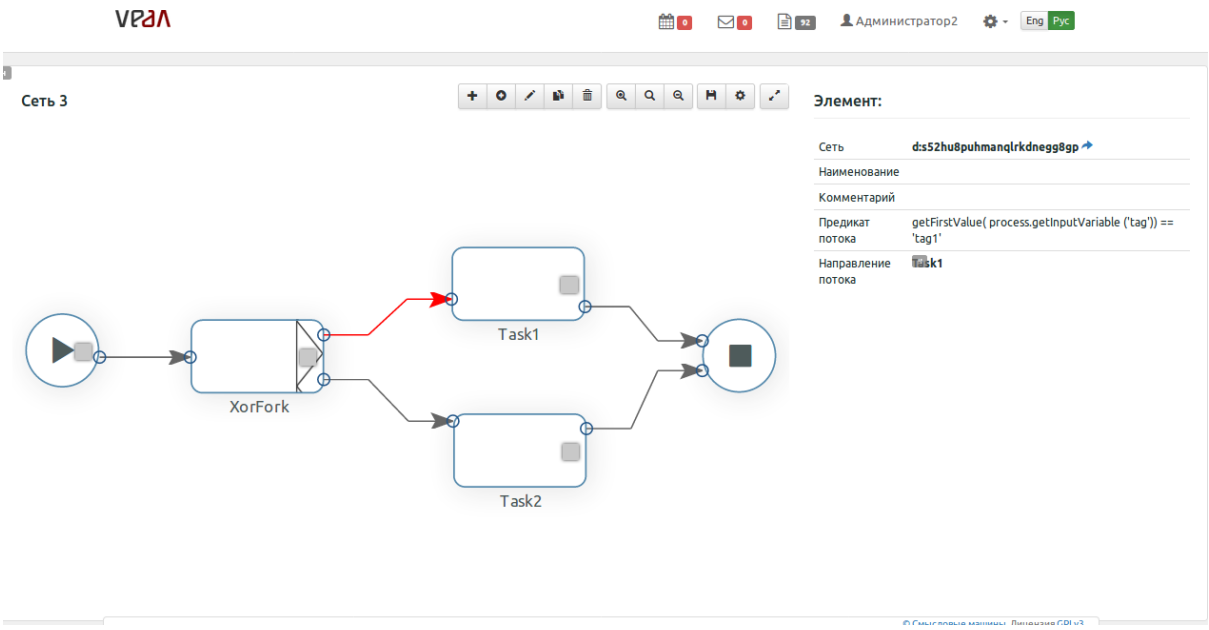


Для работы нам будут необходимы три переменные docid, tag, initiator.
Получим их с помощью следующей трансформации:

```
:test-tr1
   rdf:type v-wf:Transform ;
   rdfs:label "пробная трансформация"^^xsd:string;
   v-wf:transformRule :test-tr1-rule-initiator, :test-tr2-rule-docid, :test-tr3-rule-tag;
.

:test-tr1-rule-initiator
  rdf:type v-wf:Rule ;
  v-wf:aggregate "putUri ('rdf:type', 'v-wf:Variable')"^^xsd:string;
  v-wf:aggregate "putValue ('v-wf:variableValue')"^^xsd:string;
  v-wf:aggregate "putString ('v-wf:variableName', 'initiator')"^^xsd:string;
  v-wf:segregateElement "contentName('v-s:creator')" ;
.

:test-tr2-rule-docid
  rdf:type v-wf:Rule ;
  v-wf:aggregate "putUri ('rdf:type', 'v-wf:Variable')"^^xsd:string;
  v-wf:aggregate "putValue ('v-wf:variableValue')"^^xsd:string;
```

```
  v-wf:aggregate "putString ('v-wf:variableName', 'docid')"^^xsd:string;
  v-wf:segregateElement "contentName('@')" ;
.

:test-tr3-rule-tag
  rdf:type v-wf:Rule ;
  v-wf:aggregate "putUri ('rdf:type', 'v-wf:Variable')"^^xsd:string;
  v-wf:aggregate "putValue ('v-wf:variableValue')"^^xsd:string;
  v-wf:aggregate "putString ('v-wf:variableName', 'tag')"^^xsd:string;
  v-wf:segregateElement "contentName('v-s:tag')" ;
.
```

Теперь необходимо описать условный переход для потока. Для этого необходимо описать предикат потока. Выделив, стрелку заполним это поле выражением getFirstValue(process.getInputVariable ('tag')) == 'tag1' для потока в направлении Task1.



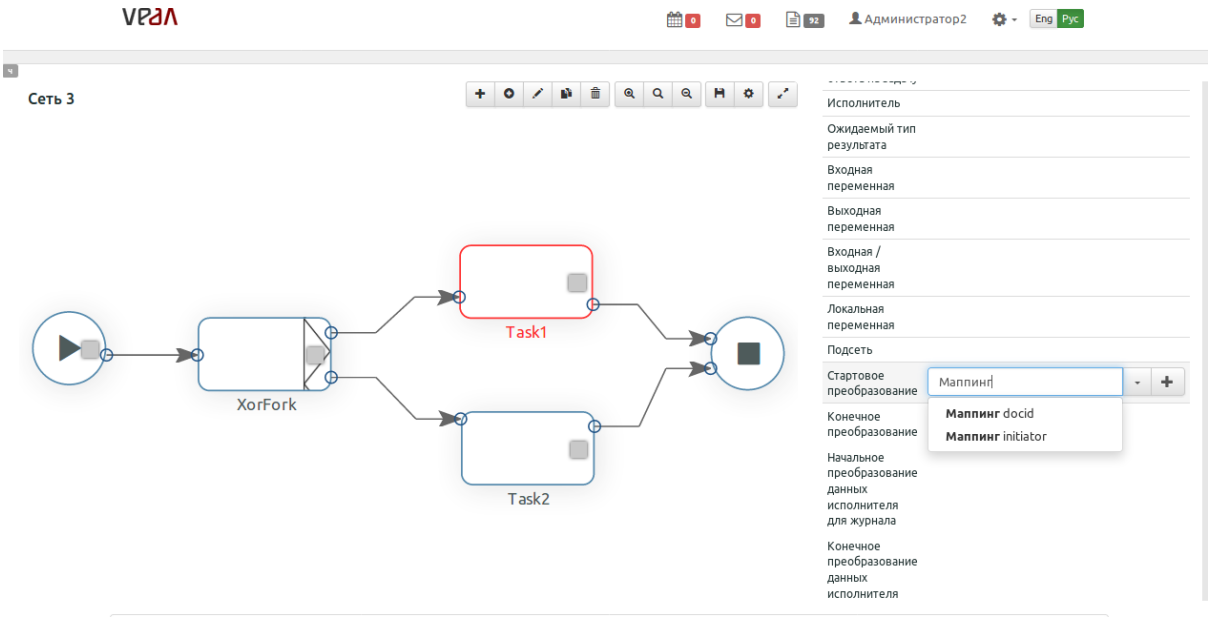
Похожим выражением getFirstValue(process.getInputVariable ('tag')) == 'tag2' заполним предикат потока в направлении Task2.

Для Task1 необходимы два стартовых преобразования для определения локальных переменных:

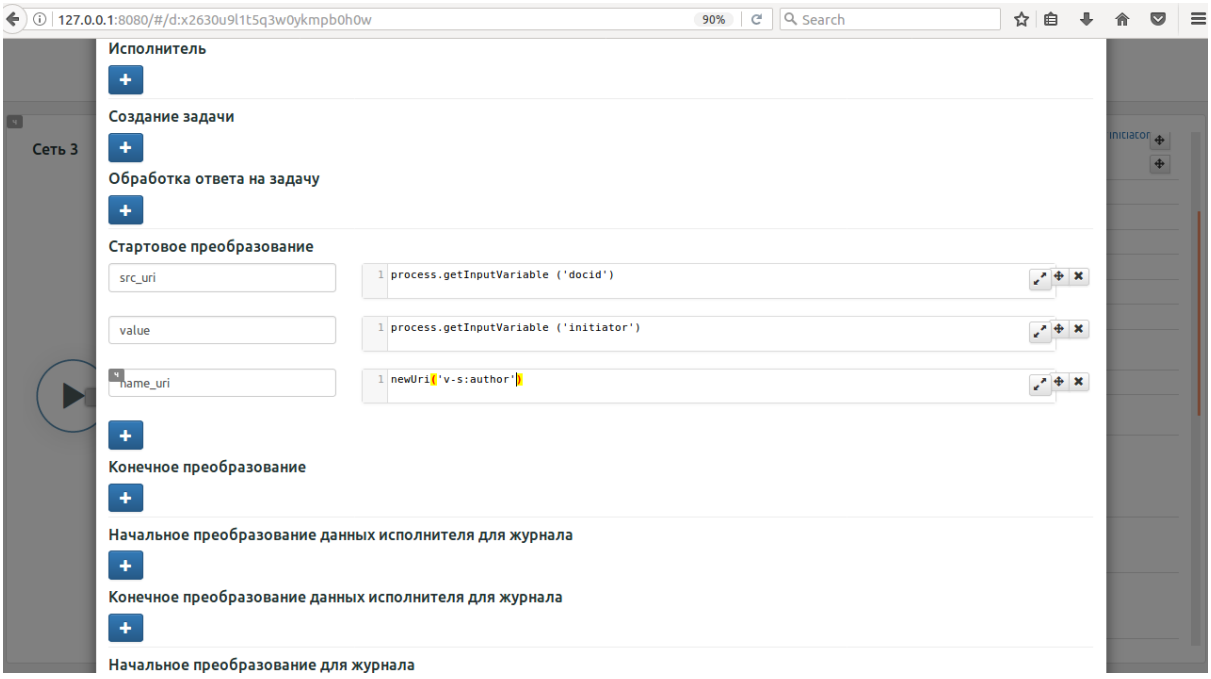
```
:test-map-docid
  rdfs:label "Маппинг docid"@ru;
  rdf:type v-wf:Mapping ;
  v-wf:mapToVariable d:var_docid ;
  v-wf:mappingExpression  "process.getInputVariable ('docid')" ;
  rdfs:isDefinedBy s-wf: ;
.

:test-map-initiator
  rdfs:label "Маппинг initiator"@ru;
  rdf:type v-wf:Mapping ;
  v-wf:mapToVariable d:var_initiator ;
  v-wf:mappingExpression  "process.getInputVariable ('initiator')" ;
  rdfs:isDefinedBy s-wf: ;
.

d:var_docid
  rdf:type v-wf:VarDefine ;
  v-wf:varDefineName "src_uri" ;
  rdfs:label "src_uri" ;
  v-wf:varDefineScope v-wf:Net ;
.

d:var_initiator
```

```
rdf:type v-wf:VarDefine ;
v-wf:varDefineName "value" ;
rdfs:label "value" ;
v-wf:varDefineScope v-wf:Net ;
.
```

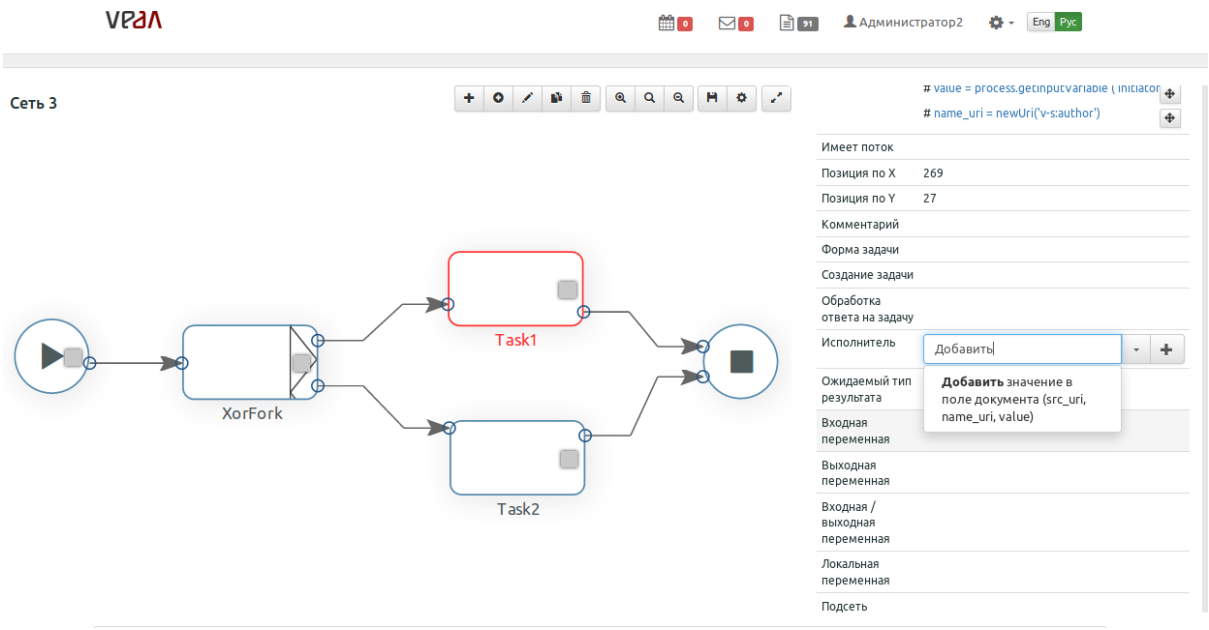
Добавим их в поле “Стартовое преобразование”. Для того чтобы найти интересующее нас преобразование можно ввести часть названия “Маппинг”.



name_uri создадим вручную для этого нужно дважды кликнуть по прямоугольнику задания, после этого появится всплывающее окно, в котором мы можем создать name_uri:

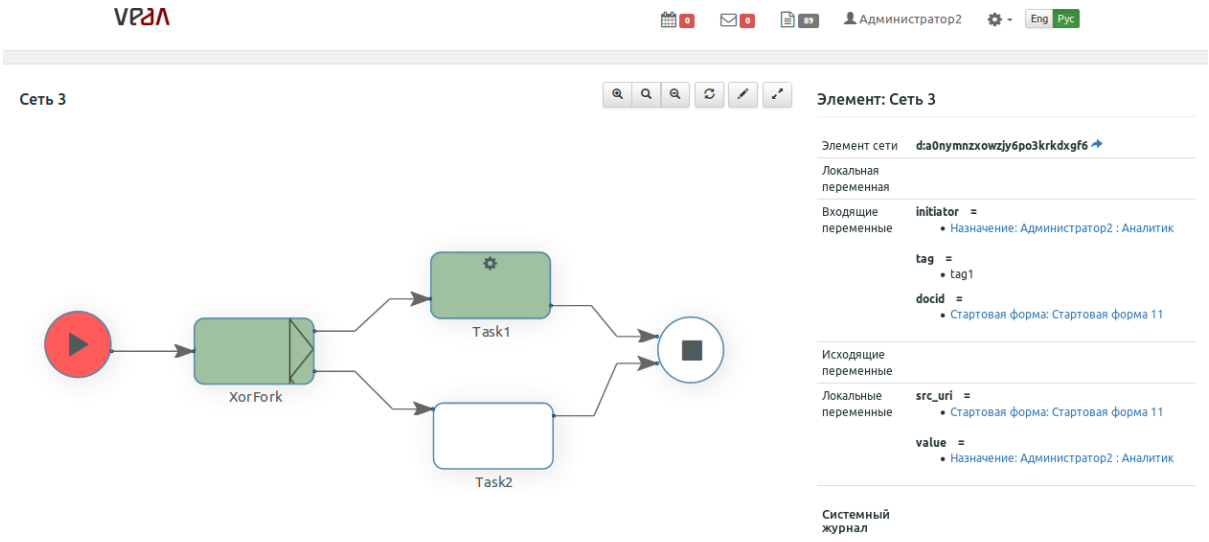


после заполнения поля нажимаем кнопку сохранить и переходим к добавлению исполнителя задачи. Для автоматического выполнения добавим коделет add_value_to_document



Для Task2 подобных действий выполнять не будем.

Перейдем к созданию стартовой формы. Выберем нашу сеть, и в поле "использовать преобразование" укажем наше преобразование. В условии разветвления используется значение из поля "ключевые слова", запишем в нём tag1. Запустим процесс. Видим, что маршрут пошёл по ветви с условием на tag1.



Трансформация: как это работает.

Алгоритм работы трансформации простой: вычленяем из входной информации только нужное нам, преобразуем отобранное, и последним этапом группируем.

Рассмотрим детальнее этот процесс. На вход функции трансформации подается массив индивидов, и ссылку на правила преобразования. На выходе будет новый массив индивидов. При этом, размер выходного массива может быть произвольным. Другими словами, мы можем сделать из одного индивида несколько, либо наоборот из нескольких один.

Фазы преобразования данных:

A. Фильтрация информации на уровне индивидов и их полей
B. Преобразование полей и сохранение новых во временный буфер
C. Группировка полей
D. Создание новых индивидов

Первые две фазы исполняются столько раз, сколько у нас есть полей во всех индивидах. Иначе говоря, алгоритм обходит все поля всех исходных индивидов, используя два вложенных цикла. Текущий обрабатываемый в цикле индивид будем называть ***объект***, а текущее поле во вложенном цикле - ***элемент***.

Класс трансформации v-wf:Transform содержит в себе ссылки на правила преобразования v-wf:Rule, которые в свою очередь описывают выражения для фаз преобразования -
v-wf:segregateObject - фильтрация индивидов
v-wf:segregateElement - фильтрация полей
v-wf:aggregate - преобразование полей
v-wf:grouping - группировка

Доступные js функции по фазам:

A. Фильтрация по индивидам: v-wf:segregateObject [expression == true/false]

- objectContentStrValue (name, value) - объект содержит поле [name] с содержимым [value]

A. Фильтрация по полям: v-wf:segregateElement [expression == true/false]

- contentName (name) - элемент имеет имя [name]
- elementContentStrValue (name, value) - элемент имеет имя [name] и содержит строковое значение [value]

B. Преобразование: v-wf:aggregate [expression == {data: xxx; type: ttt}]

- getElement () - возвращает значение элемента
- putFieldOfIndividFromElement (name, field) - сохраняет в буфер поле с именем [name], взятое из поля [field], индивида найденного в базе, по ссылке содержащейся в значении элемента
- putElement (name) - сохраняет в буфер поле с именем [name] и значением из элемента
- putFieldOfObject (name, field) - сохраняет в буфер поле с именем [name], взятое из поля [field] объекта
- putUri (name, value) - сохраняет в буфер поле с именем [name], значением [value] и типом Uri
- putString (name, value) - сохраняет в буфер поле с именем [name], значением [value] и типом String
- putBoolean (name, value) - сохраняет в буфер поле с именем [name], значением [value] и типом Boolean
- putExecutor (name) - сохраняет в буфер поле с именем [name] ссылки на исполнителей сети
- putWorkOrder (name) - сохраняет в буфер поле с именем [name] ссылки на рабочее задание

Пример 1. Преобразование - много -> один

Допустим, у нас есть массив из нескольких индивидов:

```
tst:individual1
        rdf:type tst:colorA ;
        rdfs:label "red" ;
        v-s:login "BychinA" .

tst:individual2
        rdf:type tst:colorB ;
        rdfs:label "green" ;
        v-s:login "KarpovR" .

tst:individual3
        rdf:type tst:colorA ;
        v-s:login "KarpovR" .

tst:individual4
        rdf:type tst:typeA ;
        rdfs:label "long" .
```

А нам нужен один индивид (xxxn - любой id), содержащий некоторые поля из нескольких индивидов:

```
:xxx1
        rdf:type :typeX ;
        tst:colorA "red” ;
        tst:colorB "green" ;
        tst:typeA "long" .
```

Для начала отфильтруем нужные нам индивиды:

v-wf:segregateObject "objectContentStrValue ('rdf:type', 'colorB’) || objectContentStrValue ('rdf:type', 'colorA’)"

Далее фильтруем нужные поля: *v-wf:segregateElement "contentName('rdf:type')"*

Теперь преобразуем данные: *v-wf:aggregate “putFieldOfObject (getElement(), ‘rdfs:label’)”*

И добавим тип для нового индивида: *v-wf:aggregate "putUri ('rdf:type', ‘:typeX')";*

В итоге получилось правило:

```
tst:transformation1
        rdf:type v-wf:Transform ;
        v-wf:transformRule tst:rule1, tst:rule2.

tst:rule1
        rdf:type v-wf:Rule ;
        v-wf:segregateObject "objectContentStrValue ('rdf:type', 'colorB’) || objectContentStrValue ('rdf:type', 'colorA’)"  ;
        v-wf:aggregate “putFieldOfObject (getElement(), ‘rdfs:label’)” ;
        v-wf:grouping          "1".

tst:rule2
        rdf:type v-wf:Rule ;
        v-wf:segregateObject "objectContentStrValue ('rdf:type', 'colorB’) || objectContentStrValue ('rdf:type', 'colorA')" ;
        v-wf:aggregate    "putUri ('rdf:type', ':typeX')";
        v-wf:grouping          "1".
```

Пример 2. Преобразование - один -> много

дано:

```
tst:individX
        rdf:type :typeX ;
        tst:color "red” ;
        tst:color "green" ;
        tst:color "blue" .
```

требуется получить:

```
:xxxx1
        rdf:type :typeY ;
        tst:color "red" .

:xxxx2
        rdf:type :typeY ;
        tst:color "green" .

:xxx3
        rdf:type :typeY ;
        tst:color "blue" .
```

В этом примере нам не понадобится v-wf:segregateObject, так как фильтровать объекты не нужно, входящий массив содержит один индивид.

Правило:

```
tst:transformation2
        rdf:type v-wf:Transform ;
        v-wf:transformRule tst:rule21 .

tst:rule21
        rdf:type v-wf:Rule ;
        v-wf:segregateElement "contentName('tst:color')" ;
        v-wf:aggregate        "putUri ('rdf:type', ':typeY')";
        v-wf:aggregate        "putElement ('tst:color')".
```

VWE: Veda Workflow Engine

Бушенев В. А. Максименко В. В.
ООО Смысловые Машины
Россия. Республика Коми. Сыктывкар.

Ограничения данной реализации:

- OR и XOR слияния потоков не реализованы.
- Агентами являются v-s:Appointment и v-s:Codelet

Veda Workflow Engine (VWE), как это работает.

Объекты VWE:

- *v-wf:StartForm* - стартовая форма с исходными данными процесса
- *v-wf:Process*- процесс, описывает запущенный экземпляр сети *net*
- *v-wf:WorkItem* - рабочий элемент, описывает запущенный экземпляр задачи *task*
- *v-wf:WorkOrder* - рабочее задание для конкретного исполнителя
- *v-wf:DecisionForm* - форма ответа(решения) пользователя
- *v-wf:Variable* - экземпляр переменной

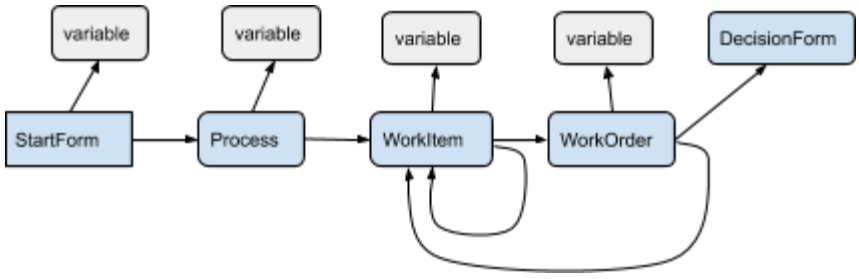
Принцип работы VWE:

В основе работы VWE лежит - такое свойство платформы Veda, как возможность исполнения заданного JavaScript по событию создания или изменения индивида. Это аналог триггеров в SQL базах данных.

Таким образом, запуск сети начинается с обработки индивида типа v-wf:StartForm, который порождает v-wf:Process, который в свою очередь находит в сети v-wf:InputCondition и порождает для него v-wf:WorkItem. Последний порождает, в зависимости от ситуации, либо снова v-wf:WorkItem, либо v-wf:WorkOrder, которые обрабатывают агенты и помещают в них результаты своей деятельности. Обработка v-wf:WorkOrder порождает v-wf:WorkItem. Заканчивается выполнение сети обработкой элемента сети типа v-wf:OutputCondition. В процессе обработки этого элемента, никаких новых v-wf:WorkItem не порождается, и движок завершает исполнение процесса.

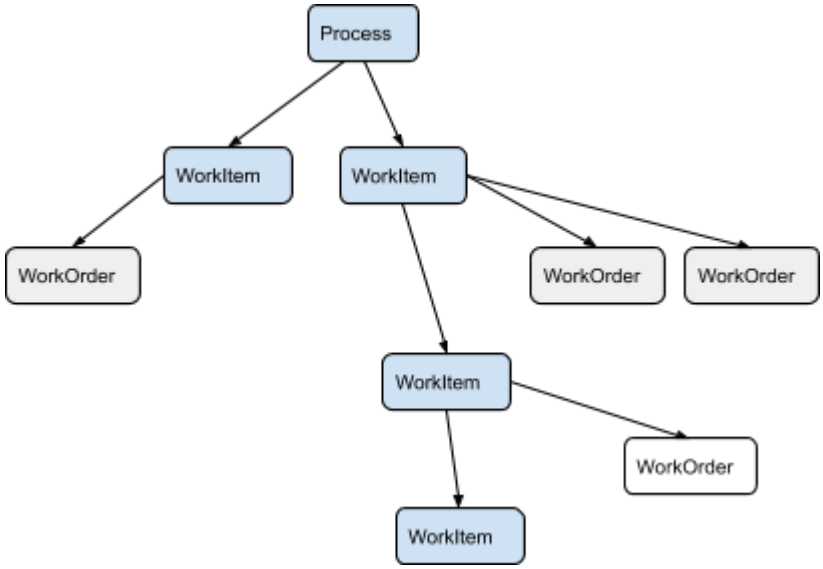
Схематично процесс порождения объектов можно представить в виде диаграммы.

Рисунок 1. Диаграмма порождения различных типов объектов в ходе исполнения процесса.



Все созданные индивиды в процессе обработки связываются друг с другом, таким образом создается дерево выполнения сети, которое можно обойти для дальнейшего анализа.

Рисунок 2. Дерево порожденных индивидов.



Рассмотрим детальнее каждый этап исполнения сети.

1. Старт сети

Исполнение сети начинается с обработки события создания индивида типа v-wf:StartForm. Это стартовая форма запуска процесса, она обязана содержать следующие атрибуты:
v-s:hasStatusWorkflow = v-s:ToBeSent
v-wf:forNet - ссылка на сеть, которая должна быть запущена.
v-wf:useTransformation - указание на правила преобразования данных, подготавливающие переменные сети. Обычно они формируются из других полей данной стартовой формы.
В процессе обработки стартовой формы будет создан процесс из v-wf:forNet, а также необходимые переменные, согласно правилам преобразования (v-wf:useTransformation). Переменные будут привязаны к процессу в поле v-wf:inVars. Обработка стартовой формы закончена.

2. Обработка экземпляра v-wf:*Process*

Первым шагом обработки является создание локальных переменных для текущего процесса. Затем, в сети, которую представляет процесс, происходит поиск элемента типа v-wf:InputCondition. Для найденного элемента подготавливается экземпляр v-wf:WorkItem, который связан с обрабатываемым процессом, его сетью и найденным элементом типа v-wf:InputCondition.
Вновь созданный экземпляр v-wf:WorkItem связывается с текущим процессом полем v-wf:workItemList. Экземпляр v-wf:WorkItem сохраняется в базе данных. Обработка процесса закончена.

3. Обработка экземпляра v-wf:*WorkItem*

Если поле v-wf:isCompleted == true, то обработка прекращается.

Рассматривается поле v-wf:join, если == v-wf:AND. Так как каждая из входящих задач порождает v-wf:WorkItem для текущей задачи, то данная проверка на v-wf:join == v-wf:AND, будет происходить каждый раз. Далее найдем все v-wf:WorkItem порожденные от задач имеющих выходы к текущей задаче и проверим, все ли они были успешно завершены. Если да, то продолжим обработку, иначе закончим обработку.
! Важно обратить внимание, что из всех входов от других задач, пройдет дальше только один из них (первый из списка).

Далее, если тип элемента == v-wf:Task, выполняется формирование входящих переменных задачи из поля v-wf:startingMapping.

> Здесь происходит вычисление исполнителей (агентов) для задачи. Данные для вычисления берутся из поля v-wf:executor. Здесь могут быть индивиды трех типов: v-s:Appointment, v-wf:Codelet,

v-wf:executorExpression. Последний представляет собой JavaScript выражение, результатом работы которого будет список из индивидов типа v-s:Appointment или v-wf:Codelet.
Для каждого элемента списка исполнителей будут порождены задания - индивиды типа v-wf:WorkOrder. В каждом из них будет указание на текущий рабочий элемент, исполнителя, а также подсеть, если задача представляет собой запуск подсети.
Если не было найдено ни одного исполнителя, будет сформировано пустое рабочее задание.
Список сформированных рабочих заданий будет включен в текущий рабочий элемент в поле v-wf:workOrderList, в дальнейшем этот список будет использоваться для определения, все ли рабочие задания были исполнены.
На этом обработка завершается.

Если тип элемента == v-wf:InputCondition
Происходит выбор следующих элементов из полей v-wf:hasFlow->v-wf:flowsInto, и порождение для каждого из них соответствующих рабочих элементов.

Если тип элемента == v-wf:OutputCondition
Если был указан v-wf:parentWorkOrder то выполняется формирование переменных по условиям из поля v-wf:completedMapping, которые сохраняются в [v-wf:parentWorkOrder]->v-wf:outVars. В случае отсутствия условий для формирования переменных, в v-wf:outVars помещается v-wf:complete. Далее в рабочий элемент помещается поле v-wf:isCompleted = true.

4. *Обработка экземпляра* v-wf:*WorkOrder*

[Обработка новых рабочих заданий]
Здесь берутся только необработанные рабочие задания.
Если не указаны исполнители, то в исходящие переменные (v-wf:outVars) заносится v-wf:complete.

Если тип исполнителя v-s:Codelet, то проводим прямое исполнение данного скрипта, и обрабатываем результаты его работы, путем преобразования результатов в переменные с помощью указаний в v-wf:completedMapping и помещения их в v-wf:outVars. Далее переходим к [Обработка результатов рабочих заданий].

Если тип исполнителя v-s:Appointment, то производим формирование v-wf:DecisionForm для пользователя, с помощью правил трансформации указанных v-wf:startDecisionTransform. Так же выдаются права исполнителям на редактирование вновь созданных v-wf:DecisionForm.

Если исполнитель v-wf:Net, или указано, что используется подсеть (v-wf:useSubNet == true), то генерируем новый подпроцесс.

Обработка завершена.

[Обработка результатов рабочих заданий]

5. Обработка экземпляра DecisionForm

Если v-wf:isCompleted == true или поле v-wf:takenDecision не заполнено, то обработка завершается.

При составлении данного документа была использована информация из следующих источников:

1 - ECM-Journal.ru, Перспективы WorkFlow-систем. Паттерны – попытка навести порядок в "мире" workflow
16 мая 2007 г. 12:25, Андрей Михеев, Михаил Орлов,
http://ecm-journal.ru/post/Perspektivy-WorkFlow-sistem-Patterny--popytka-navesti-porjadok-v-mire-workflow.aspx