# DPMine/P: modeling and process mining language and ProM plug-ins*

Sergey A. Shershakov
National Research University
Higher School of Economics
PAIS Lab
20 Myasnitskaya st.
Moscow, 101000, Russia
sshershakov@hse.ru

## ABSTRACT

Process-aware information systems (PAIS) enable developing models for interaction of processes, monitoring accuracy of their execution and checking if they interact with each other properly. PAIS can generate large data logs that contain information about the interaction of processes in time. Studying PAIS logs with the purpose of data mining and modeling lies within the scope of Process Mining. There is a number of tools developed for Process Mining, including the most ubiquitous ProM, whose functionality is extended by plugins. To perform an object-aware experiment one has to sequentially run multiple plugins. This process becomes extremely time-consuming in the case of large-scale experiments involving a large number of plugins. The paper proposes a concept of *DPMine/P* language of process modeling and analysis to be implemented in ProM. The language under development aims at joining separate stages of the experiment into a single sequence, that is an *experiment model*. The implementation of the basic semantics of the language is done through the concept of blocks, ports, connectors and schemes. These items are discussed in detail in the paper, and examples of their use for specific tasks are presented ibid.

## Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: System Adaptation; D.3.2 [**Java**]: Application

## General Terms

Algorithms, Measurement, Experimentation, Languages

## Keywords

Process Mining, Modelling Language, Model, Tool, Process, PAIS

*This work is supported by the Basic Research Program of the National Research University Higher School of Economics.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
*CEE-SECR* '13 October 23 - 25 2013, Moscow, Russian Federation
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2641-4/13/10…$15.00.
http://dx.doi.org/10.1145/2556610.2556622

## 1. ВВЕДЕНИЕ

С развитием информационных систем (ИС) стремительно возросли объемы данных, которыми они оперируют. Это касается как данных, вводимых в систему различными путями (автоматическими, полуавтоматическими и ручными), так и данных, получаемых в результате некоторой обработки, то есть выводимых этой системой на различные виды носителей информации.

Из последнего типа данных можно выделить целый специальный подкласс, к которому относятся так называемые логи данных. *Лог данных* — это своеобразный след, который оставляет после себя некоторая ИС, хранящий информацию о наборе частных состояний такой системы в разное время ее работы. Это так называемые *временны́е логи*, к которым, вообще говоря, относится большинство логов.

Логи данных известны давно, так как являются удобным инструментом для выполнения целого класса задач, к которым можно отнести диагностику ошибок, документирование последовательности обращения к различным узлам системы, сохранение важной системной информации и т.д.

Количество информации, записываемой в лог в течение фиксированного интервала времени, может быть весьма существенным, делая практически невозможным ручной анализ такого лога пользователем, что сталкивает нас с так называемой проблемой «больших данных» (Big Data). Для исследования больших массивов данных, представленных некоторым упорядоченным образом, например с помощью баз данных, привлекаются различные информационно-ориентированные методы, такие как *машинное самообучение* (Machine Learning), *извлечение данных* (Data Mining) и др.

Отдельный интерес представляют *процессно-ориентированные информационные системы* (ПОИС; process-aware information systems, PAIS), основным понятием которых является *процесс* (например, бизнес-процесс или технологический процесс). Такие системы позволяют разрабатывать модели взаимодействия процессов, отслеживать правильность их исполнения и корректность взаимодействия друг с другом. Как и в случае со многими другими ИС, ПОИС могут порождать большие логи, содержащие в себе информацию о взаимодействии процессов во времени.

Исследованием логов ПОИС с целью извлечения знаний о процессах и построения их моделей, исследованием таких моделей занимается дисциплина *Process Mining*, имеющая тесные

связи с извлечением данным, машинным обучением, моделированием и анализом моделей процессов. Основные задачи и цели Process Mining изложены в Манифесте (Process Mining Manifesto) [1] и укрупненно могут быть сведены к трем ключевым проблемам: 1) извлечение модели из лога данных (*process discovery*), 2) проверка соответствия некоторой модели реальным данным (*conformance checking*) и 3) улучшение и исправление модели в соответствии и с учетом изменяющихся данных (*enhancement*).

*Лог* данных является отправной точкой для исследований Process Mining. Как правило, такой лог представляет собой последовательность *событий*, объединенных общим *прецедентом*. Каждое событие является экземпляром некоторой *активности*, представляющей хорошо выделенный этап некоторого процесса. Последовательность событий, относящихся к одному прецеденту, образуют так называемую *трассу*, представляющую отпечаток взаимодействия процессов в отдельно взятом случае.

Одной из наиболее часто используемых в Process Mining форм представления моделей являются *сети Петри* [2]. Сети Петри используются как для внутреннего представления модели различными алгоритмами [3, 4], так и в целях визуализации. Другие модели (например, эвристические и C-сети), могут быть сведены к сетям Петри.

### 1.1 Инструменты Process Mining

К настоящему моменту разработан ряд инструментов для Process Mining, распространяемых как на коммерческой, так и на свободной основе. Одним из наиболее распространенных инструментов в области является *ProM* — кроссплатформенное приложение, функциональность которого расширяется за счет плагинов, число которых составляет несколько сотен.

Плагины, разрабатываемые для ProM, выполняют описанные выше задачи с использованием различных алгоритмов, некоторые из которых в настоящий момент находятся в процессе исследования и улучшения, а часть — представляют собой в основном историческую ценность.

Большое число плагинов выполняет утилитарные и вспомогательные функции: это извлечение данных из различных источников, подготовка (преобразование) данных к формату, подходящему для использования с тем или иным алгоритмом, конвертация различных форматов между собой, визуализация и анимация полученных результатов и др.

Часто для выполнения предметно-ориентированного эксперимента приходится осуществлять последовательный запуск нескольких (иногда десятков) плагинов, каждый из которых выполняет узкую часть большой задачи. Ситуация усложняется, когда подобную последовательность запусков приходится осуществлять снова и снова, изменяя отдельные параметры отдельных плагинов, например с целью поиска оптимальных результатов. Процесс становится исключительно трудоемким в случае проведения широкомасштабных экспериментов (large-scale experiments), вовлекающих множество плагинов и определенную логику для автоматической интерпретации полученных результатов.

В данной работе предлагается концепция языка построения моделей извлечения и анализа процессов и описание набора плагинов *DPMine/P* для инструмента ProM, являющихся механизмом реализации этого языка.

Остальная часть работы организована следующим образом. В разделе 2 приводятся общие характеристики языка, рассматривается его основная концепция и указывается связь языка с базовым инструментом (ProM). Раздел 3 раскрывает понятие модели эксперимента, принципа ее интерпретации и представления на разных уровнях абстракции. В разделе 4 дается синтаксико-семантическое описание некоторых блоков и приводятся примеры их использования для решения демонстрационных задач. Наконец, раздел 5 подытоживает результаты и приводит перпективу дальнешей работы по теме.

## 2. ЯЗЫК И ИНСТРУМЕНТ

Сложность предметно-ориентированных экспериментов, подразумевающих задействование большого числа алгоритмов, их итеративного использования и параметризации, нелинейность структуры потоков объектов данных, являющихся выходными результатами одних алгоритмов и входными — для других, — все это продиктовало необходимость введения некоторого языка описания эксперимента, обладающего гибкостью и простотой одновременно.

Разрабатываемый язык нацелен на реализацию объединения отдельных этапов эксперимента в единую последовательность — *модель эксперимента*, поддержку конструкций циклов и других элементов управления потоками исполнения, обладание прозрачной, но гибкой (и что важно — расширяемой) семантикой. Рабочее название проекта инструмента и языка описание экспериментов — *DPMine* [5].

Как и любой язык, DPMine нуждается в носителе, то есть в интерпретаторе или компиляторе. В данной работе рассматривается приложение языка к инструменту ProM, являющимся *базовым*, то есть по сути платформой для исполнения модели (DPModel), описываемоей языком DPMine/P.[1]

### 2.1 Уровни представления языка

Рассмотрение языка осуществляется с двух уровней представления: на нижнем уровне находится инструменто-ориентированная *объектная модель*; на средне-верхнем — собственно язык, базирующийся на XML (*уровень хранения и представления модели*), а также *графическое представление*, позволяющее задавать модель процесса в виде набора строительных элементов (блоков). Графическая модель преобразуется в XML-представление, компилируемое в объектную модель, которая в свою очередь исполняется на базе инструмента Process Mining, в частном случае — ProM (см. рис. 1).

### 2.2 Концепция блоков, портов, коннекторов и схем

Реализация основной семантики языка осуществляется через концепцию блоков, портов, коннекторов и схем. Подразумевается, что расширение функционала языка также должно осуществляться с опорой на эту концепцию. Рассмотрим эти элементы подробнее.

[1] Постфикс «/P» указывает на реализацию для ProM.

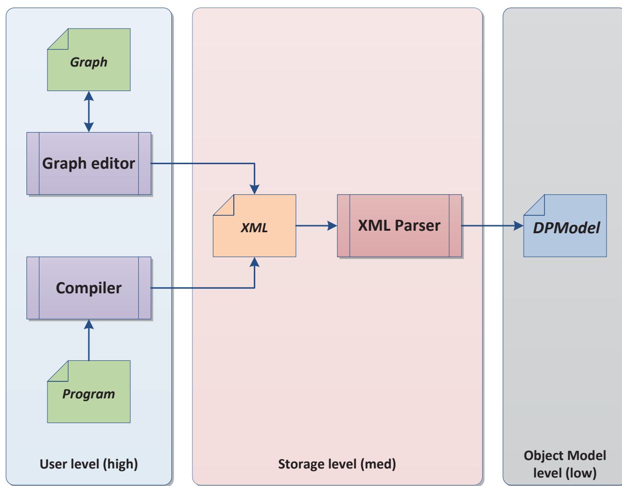


**Рис. 1. Уровни представления языка**

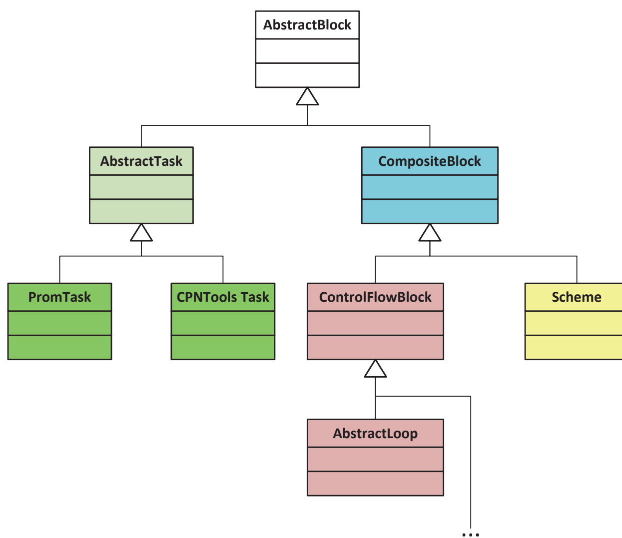


**Рис. 2. Диаграмма иерархии типов блоков**

### 2.2.1 *Блоки*

*Блок* — основной строительный элемент языка, рассматривается как элементарная операция, но необязательно таковой является. Блок, в зависимости от своего типа, реализует одиночную задачу базового инструмента (например путем вызова определенного его плагина), используется для иерархического представления сложных схем (в виде единого блока специального типа «схема»), реализует конструкции управления потоком выполнения, используется как оператор подстановки для передачи какой-то схемы в другую схему в виде параметра (вводя элементы функционального программирования) и др. По выполняемой функции блоки объединяются в иерархию типов, представленную на схеме на рис. 2.

### 2.2.2 *Порты*

*Порт* — объект связи, принадлежащий некоторому блоку, обладающий характеристиками направления (входные, выходные и прокси-порты) и типа данных. Используются для транспортировки объектов заданного типа в блок и

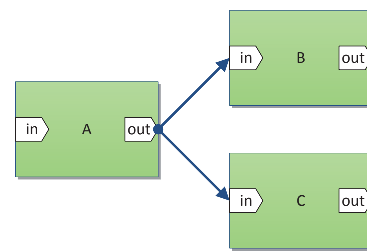


**Рис. 3. Порты и коннекторы**

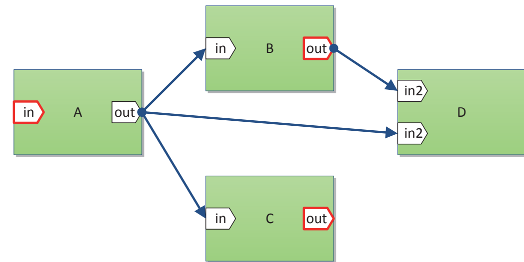


**Рис. 4. Связная схема**

из него. В зависимости от типа блока подразделяются на *пользовательские* и *встроенные* (built-in).

### 2.2.3 *Коннекторы*

*Коннектор* — направленный объект связи, соединяющий два блока через их порты: выходной порт одного блока (своим началом) с входным портом другого (своим концом). Из одного выходного порта может выходить несколько коннекторов, в один входной порт входит всегда только один коннектор (рис. 3).

### 2.2.4 *Схемы*

*Схема* — множество взаимодействующих блоков, связанных между собой коннекторами. Является основным механизмом реализации абстрагирования, изолирования и иерархии подпроцессов.

Блоки, входящие в состав некоторой схемы, называются *изолированными* (isolated), если они не имеют ни одной связи (то есть не соединены коннекторами). Схема $S$ называется *связной* (connected), если она не имеет изолированных блоков (рис. 4).

*Интерфейсом схемы* назовем произвольное подмножество $Ifp$ портов (называемых интерфейсными портами) множества всех портов, образованного блоками, входящими в состав этой схемы. На рис. 4 интерфейсом схемы являются порты $Ifp = \{A.in, B.out, C.out\}$, где под нотацией $A.in$ подразумевается порт *in* блока $A$ (интерфейсные порты обведены красным).

## 3. МОДЕЛЬ: ПРЕДСТАВЛЕНИЕ И ИСПОЛНЕНИЕ

На уровне инструмента ProM язык DPMine/P рассматривается как набор плагинов и объектов данных (являющихся входными и выходными для данных плагинов). Основным объектом является (объектная) модель эксперимента DPModel/P — модель

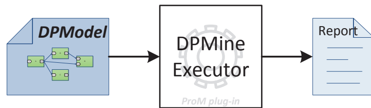


**Рис. 5. Исполнение модели интерпретатором**

workflow, представленная в виде объекта данных (Java-object) в инструменте ProM.

## 3.1 XML-представление

Для структурированного описания модели и входящих в ее состав элементов (схем, блоков, коннекторов и др.) используется расширяемый язык разметки XML.

XML-модель можно разрабатывать как вручную, так и получать в качестве выходного результата работы специализированного редактора, например графического (будущая работа), либо компилятора с какого-то другого языка, например скриптового.

**Листинг 1. Пример описания модели**

```
<model name="MMain">
  <lib id="prom/base.dpmlib"/>
  <lib id="prom/petrinets.dpmlib"/>
  ...
  <scheme name="SMain">
    ...
  </scheme>
  <report filename="..."/>
</model>
```

Пример описания модели в XML-формате представлен в листинге 1. Здесь элемент `model` задает узел описания модели, включающий такие элементы как ссылки на используемые библиотеки (`lib`), схему верхнего уровня (`scheme`), параметры отчета (`report`) и др.

Разные элементы модели (собственно модель, схемы, блоки различных типов) описываются с помощью специальных элементов XML-схемы и рассматриваются в соответствующих разделах.

## 3.2 Исполнение модели

*Исполнение модели* состоит в исполнении главной схемы этой модели (схемы верхнего уровня) с формированием отчета об исполнении (в т.ч. ошибках и др.). Исполнение модели осуществляется специальным агентом — *интерпретатором*, реализация которого тесно связана с базовым инструментом и для ProM и представляется в виде разрабатываемого плагина — *DPMineExecutor* (рис. 5).

Исполнение модели в конечном итоге декомпозируется до исполнения отдельных блоков, входящих в состав главной схемы этой модели. Рассмотрим подробнее, что под этим подразумевается.

*Исполнением блока* является набор действий, выполняемых соответствующим инструментом (например плагином DPMineExecutor) — интерпретатором — по отношению к данному блоку, определяемому его типом и набором входных параметров (на входных портах этого блока).

Для того, чтобы конкретный блок мог быть исполнен интерпретатором, необходимо, чтобы все внешние зависимости этого блока были удовлетворены. Для некоторого блока $B$ его зависимости считаются удовлетворенными, если:

1. блок $B$ не имеет входных портов;
2. блок имеет входные порты и для каждого такого порта выполняются следующие требования:
   (a) для порта не установлен флаг «обязательно подключен», тогда порт может быть не подключен коннектором к другому (выходному) порту другого блока;
   (b) порт подключен коннектором к другому (выходному) порту другого блока и статус такого блока — «исполнен», что означает наличие на его выходных портах данных, соответствующих типам этих выходных портов.

Введем несколько формальных определений для состояний блоков с точки зрения интерпретатора.

*Исполнимый блок* (executable block) — такой блок, для которого удовлетворены внешние зависимости.

*Исполняемый блок* (running block) — исполнимый блок, который в текущий момент исполняется интерпретатором.

*Обозреваемый блок* (observable block) — блок, для которого интерпретатор в текущий момент определяет, удовлетворены ли его входные зависимости.

*Неисполненный блок* (unexecuted block), *исполненный блок* (executed block) — блок который, соответственно, еще не был исполнен либо уже был исполнен интерпретатором.

*Попытка исполнения* (Execution attempt) — выбор интерпретатором очередного блока, определение, является ли он *исполнимым* (при это блок становится *обозреваемым*) и в случае положительного ответа — его исполнение (блок становится *исполняемым*) — это случай «удачной попытки исполнения блока»; иначе блок пропускается и производится переход к другому блоку — случай «неудачной попытки исполнения блока».

Если блок является исполнимым (то есть для него удовлетворены входные зависимости), тогда интерпретатором вызывается соответствующая процедура исполнения блока, которая определяется типом этого блока (а сам блок становится исполняемым).

*Последовательностью исполнения* является такая последовательность, в которой множество блоков исполняются интерпретатором при условии, что они могут быть исполнены, то есть каждый блок в такой последовательности является исполнимым. Интерпретатор может предпринимать несколько попыток исполнить какой-то конкретный блок, и при этом все эти попытки, за исключением последней, будут считаться неудачными в случае, если для данного блока не будут на момент осуществления попытки удовлетворены все его входные зависимости. Иными словами, последовательность исполнения —

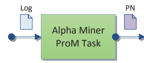


**Рис. 6. Блок типа «задача ProM»**

это последовательность удачных попыток исполнения набора блоков интепретатором.

Для схемы на рис. 4 допустимыми являются следующие (но не только) последовательности исполнения:

- A B C D
- A B D C
- A C B D

На практике последовательность, в которой набор блоков будет исполнен интерпретатором, определяется внутренним представлением объектной модели.

Для схемы вводится понятие исполнения по аналогии с блоком: *исполнением схемы* является последовательность исполнения содержащихся в ней блоков. Если все блоки, содержащиеся в некоторой схеме, могут поменять состояние с «неисполненный» на «исполненный» за конечное число шагов, то такая схема является *исполнимой*. Иными словами, в *исполнимой схеме* все блоки в ее составе схемы могут быть исполнены.

## 4. ПРИМЕРЫ БЛОКОВ И СЦЕНАРИИ ИСПОЛЬЗОВАНИЯ

Рассмотрим подробнее свойства и способы исполнения блоков различных типов.

### 4.1 Блок «Задача» (Task Block)

Выполняет конкретную задачу, связанную с базовым инструментом, например ProM или CPNTool [6]. Например, блок «задача ProM» связан с конкретным плагином ProM посредством аннотации, содержащей имя плагина, сигнатуру метода и другую необходимую информацию. Количество и типы портов блока определяются конкретным методом плагина, к которому (методу) данный блок является привязанным (рис. 6), а исполнение блока такого типа ведет к вызову этого метода, то есть фактически запуску плагина (решающего конкретную задачу Process Mining).

#### *4.1.1 XML-представление*

XML-описание блока такого типа дано в листинге 2.

**Листинг 2. Описание блока «задача ProM»**

```
<promtask name="AM1" other="..."
  plugin="org.processmining...AlphaMiner"
  method="...">
   <ports>
      <in name="log" dtype="XLog" binding="..."/>
      <in name="param1" dtype="int" binding="..."/>
      <out name="pn" dtype="PetriNet" binding="..."/>
   </ports>
</promtask>
```

Описание блока «задача ProM» вводится элементом `promtask`, имеющего в числе атрибутов `name`, задающий имя блока (должно быть уникально в пределах *области видимости* этого блока), `plugin` для указания полного квалификационного имени плагина в рамках базового инструмента (ProM) — плагина Alpha Miner [7], `method` для указания привязки к конкретному методу плагина и пр.

Секция `ports` содержит описание входных (элемент `in`) и выходных (элемент `out`) портов блока. Каждый порт имеет параметр `name` — имя порта, уникальное в пределах одного блока, `dtype` — тип данных порта и параметр привязки (к плагину) `binding`.

#### *4.1.2 Библиотека задач*

Описывать полностью блок «задача» каждый раз, когда он используется в схеме, неудобно. Для оптимизации этого процесса вводится механизм *библиотеки задач*. Такая библиотека содержит описания всех нужных блоков задач в привязке к конкретным плагинам ProM и представляет собой специальный XML-файл библиотеки (листинг 3). Для использования описанного блока в конкретной схеме указывается ссылка на библиотеку и элемент внутри нее (листинг 4).

**Листинг 3. Библиотека блоков «задача ProM»**

```
<?xml version="1.0" encoding="UTF—8" ?>
<lib name = "prom_miners" version = "0.1">
   <promtask name="AlphaMiner1" other="..."
     plugin="org.processmining...AlphaMiner"
     method="...1...">
      <ports>
         <in name="log" dtype="XLog" binding="..."/>
         <in name="param1" dtype="int" binding="..."/>
         <out name="pn" dtype="PetriNet" binding="..."/>
      </ports>
   </promtask>
</lib>
```

**Листинг 4. Модель с подключаемой библиотекой задач**

```
<?xml version="1.0" encoding="UTF—8" ?>
<model name="MMain">
   <lib name="prom_miners"
     id="prom/prom_miners.dpmlib"/>
   ...
   <scheme name="SMain">
      <promtask name="am1" lib="prom_miners"
        libitem="AlphaMiner1">
      ...
   </scheme>
   <report filename="..."/>
</model>
```

### 4.2 Блок «Схема»

*Блок типа «Схема»* (scheme block) — блок, представляющий вложенную систему блоков и коннекторов, то есть попросту схему. Блок используется для иерархической структуризации модели, его можно представить как аналог понятия «процедура» в языках программирования. На *внешнем уровне* блок «Схема» представляет собой (на общих основаниях) обычный исполняемый блок, имеющий внешние порты определенных типов данных (рис. 7).

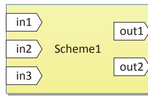


**Рис. 7. Блок типа «Схема» на внешнем уровне**

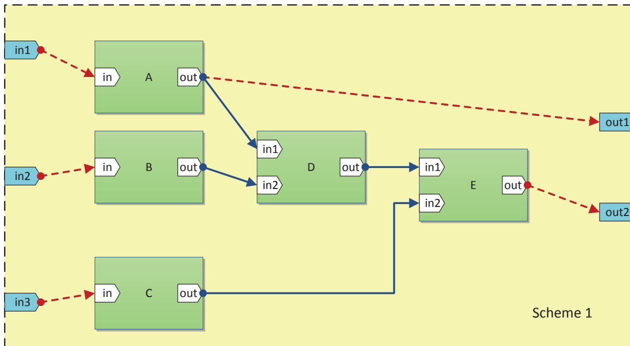


**Рис. 8. Блок типа «Схема» на внутреннем уровне**

*Исполнение* блока «Схема» на внешнем уровне осуществляется по общему для всех блоков принципу: единственным требованием является удовлетворение входных зависимостей. После исполнения блока «Схема» на его выходные порты также кладутся объекты соответствующих типов.

На внутреннем уровне блок «Схема», как это следует из его названия, представляет собой схему, состоящую из блоков и коннекторов, соединяющих их (рис. 8). Блоки, содержащиеся внутри схемы, называются *внутренними блоками* (на рисунке порты таких блоков белые). Коннекторы, которыми внутренние блоки соединяются (строго) между собой, по аналогии называются *внутренними коннекторами* (на рисунке отображены сплошными синими стрелками).

Соединение внешних портов блока «Схема» (на рисунке — синие) с портами внутренних блоков (внутренними портами) так же осуществляется посредством коннекторов, которые, однако, имеют особое назначение и название: это *интерфейсные коннекторы* (или *прокси-коннекторы*; на рисунке обозначены штриховыми линиями красного цвета). Внутренние порты схемы, соединенные прокси-коннекторами с внешними портами, являются *интерфейсом схемы* (см. раздел 2.2.4).

Под *исполнением* блока «Схема» подразумевается последовательное исполнение содержащихся в такой схеме внутренних блоков, для которых удовлетворены входные зависимости. Так, для схемы `Scheme 1` (рис. 8) допустимыми являются следующие (но не только) последовательности исполнения внутренних блоков:

- A B D C E
- A B C D E
- C A B D E

### 4.2.1 XML-представление

XML-описание блока такого типа дано в листинге 5.

Описание блока «Схема» вводится элементом `scheme`, основным атрибутом которого является `name`, задающий уникальное имя блока. Блок «Схема» содержит три основные раздела.

**Листинг 5. Описание блока «Схема»**

```
<scheme name="scheme1">
  <body>
    <block name="A">
      <ports>
        <in name="in" dtype="..."/>
        <out name="out" dtype="..."/>
      </ports>
    </block>
    <block name="B">...</block>
    <block name="C">...</block>
    <block name="D">
      <ports>
        <in name="in1" dtype="..."/>
        <in name="in2" dtype="..."/>
        <out name="out" dtype="..."/>
      </ports>
    </block>
    <block name="E">...</block>
    <connector name="c1" outp="A.out" inp="D.in1"/>
    <connector outp="B.out" inp="D.in2"/>
    <connector outp="C.out" inp="E.in2"/>
    <connector outp="D.out" inp="E.in1"/>
  </body>
  <ports>
    <in name="in1" dtype="..."/>
    <in name="in2" dtype="..."/>
    <in name="in3" dtype="..."/>
    <out name="out1" dtype="..."/>
    <out name="out2" dtype="..."/>
  </ports>
  <proxy>
    <connector name="pc1" outp=".in1" inp="A.in">
    <connector outp=".in2" inp="B.in">
    <connector outp=".in3" inp="C.in">
    <connector outp="A.out" inp=".out1">
    <connector outp="E.out" inp=".out2">
  </proxy>
</scheme>
```

Раздел `body` — тело схемы, содержащее описание входящих в ее состав внутренних блоков и коннекторов. В приведенном выше примере тэгом `block` обозначается условный абстрактный блок, структура которого единственно представляет интерес для данного примера (в реальности это будут блоки конкретных типов, например блоки «Задача»).

Раздел `ports` описывает *интерфейс схемы*, то есть те порты, которые будут видны снаружи этого блока.

Раздел `proxy` содержит описание *прокси-коннекторов*, соединяющих интерфейсные порты блока «Схема» с портами внутренних блоков схемы.

Адресация некоторого порта внутреннего блока осуществляется с использованием нотации *`имя_блока.имя_порта`*. Для указания прокси-коннектору внешнего (интерфейсного) порта

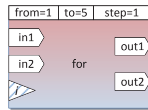


**Рис. 9. Простой блок «for»**

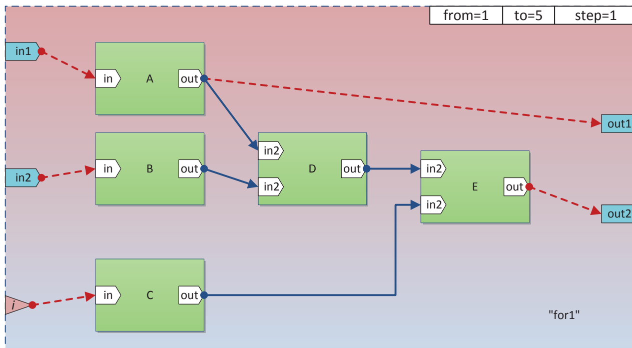


**Рис. 10. Блок «for» на внутреннем уровне**

схемы используется специальная нотация `.имя_порта`, то есть вместо имени блока подразумевается сам блок «Схема».

## 4.3 Простой блок «for»

Язык DPMine/P определяет семантику нескольких блоков, реализующих циклы. Простейшим из них является блок «for», производящий итеративное исполнение вложенной схемы с изменением параметра цикла в пределах *от*, *до* с некоторым *шагом* (рис. 9).

Блок «for» представляет собой специальный вид схемы, выполняющейся повторно (итеративно) с использованием принципа предварительного сброса элементов схемы перед очередной итерацией. Блок имеет тело — схему — и встроенный порт «*i*» (на рисунке показан заштрихованным треугольником), не имеющий возможности подключения извне, но носящий тип входного порта по отношению к внешнему интерфейсу блока. Имя этого порта задается через параметр `iname` и по умолчанию принимает значение «*i*».

Для блока «for» возможно, как и для обычного блока «Схема», определить внешние (пользовательские) интерфейсные порты, и подсоединить их нужным образом к внутренним блокам его тела.

Пример структуры тела блока «for», изображенного на рис. 9, представлен на рис. 10.

### *4.3.1 Семантика*

Блок «for» исполняет содержимое тела (схемы) в зависимости от условий from-to-step некоторое количество раз. Согласно правилам исполнения схемы (см. раздел 3.2), уже после самой первой итерации все блоки схемы (а значит и сама схема) получают признак «исполненный», что исключает повторное исполнение блоков (и схемы) на следующей итерации. Для выхода из складывающейся ситуации при выполнении очередной итерации блока «for» вначале выполняется метод `reset()` (сброс) для всей схемы (и значит содержащихся в ней блоков) перед каждой новой итерацией, в том числе и перед самой первой.

*Сброс блока* состоит в принудительном выставлении для него состояния «неисполнен», что, таким образом, делает его доступным для дальнейшего (повторного) исполнения.

### *4.3.2 Встроенный порт «*i*»*

Переменная цикла *i* учитывается в качестве значения, проключаемого на псевдо-порт «*i*», и может использоваться в качестве входного значения для любого блока, например в качестве параметра discovery-алгоритма. Тип ресурса для этого порта по умолчанию «`int`», однако он может быть изменен, например, на «`double`». Семантика приращения переменной итерации *i* аналогична таковой в цикле «for» для языка Basic, то есть сравнение с верхней границей осуществляется по условию «меньше-или-равно».

### *4.3.3 XML-представление*

Пример XML-описания блока такого типа приведен в листинге 6.

**Листинг 6. Описание блока «for»**

```
<for name="for1" from="1" to="5" step="1" iname="i">
  <body>
    <promtask name="am1" lib="prom_miner"
      libitem="AlphaMiner1">
  </body>
  <ports>
    <in name="log" dtype="org.pro...XLog"/>
    <out name="pn" dtype="org.pro...PetriNet"/>
  </ports>
  <proxy>
    <connector outp=".log" inp="am1.log"/>
    <connector outp=".i" inp="am1.param1"/>
    <connector outp="am1.pn" inp=".pn"/>
  </proxy>
</for>
```

Данный код описывает блок «for» (соответственно, элемент `for`) с именем (`name`) «`for1`», выполняющий в цикле итерации со значением переменной *i* (параметр `iname`), меняющий значения от 1 (`from`) до 5 (`to`) с шагом 1 (`step`). Тело блока состоит из одного единственного блока типа «задача ProM» с именем «`am1`» (блок вводится ссылкой на элемент «`AlphaMiner1`» библиотеки «`prom_miner`»), один из портов («`am1.param1`») которого подключен посредством прокси-коннектора к квазивходному порту «`i`» блока «`for`».

В результате исполнения блока «`for1`» на его выходной интерфейсный порт «`pn`» будет положено значение, являющееся выходным значением блока «`am1`» — результат работы alpha-miner алгоритма, — сеть Петри [7], соответствующая вызову этого алгоритма на последней итерации цикла — с параметром *i*, равным 5. Предыдущие 4 результата также останутся доступными в специальной памяти объектов ProM, где ими можно будет воспользоваться только в ручном режиме. Для того, чтобы все 5 объектов — результатов 5 итераций цикла были доступны для дальнейшей потоковой обработки в модели эксперимента, необходимо вводить дополнительные типы блоков, пример одного из которых дается в следующемм разделе.

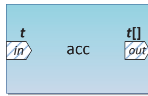


**Рис. 11. Блок-аккумулятор**

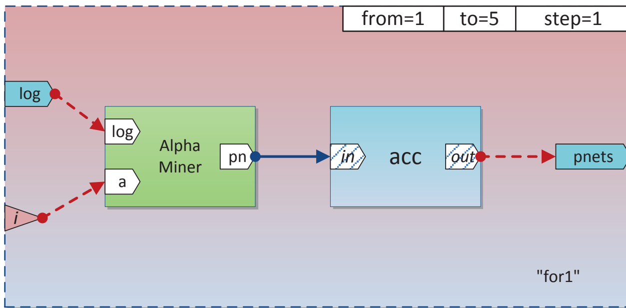


**Рис. 12. Схема тела блока модифицированного цикла «for»**

## 4.4 Блок-аккумулятор

Как и следует из его названия, блок типа «аккумулятор» используется для накапливания поступающих значений в виде массива. Блок имеет два встроенных порта: один входной «`in`» и один выходной «`out`», — имеющие типы `t` и `t[]` (массив) соответственно, где *t* задается (рис. 11).

Семантика блока-аккумулятора проста: при каждом исполнении блока он получает одно (текущее) значение со своего входного порта «`in`» и добавляет его во внутренний массив, после чего статус блока отмечается как «исполненный». При выполнении последующего сброса данного блока накопленные данные из внутреннего массива блока не стираются, что позволяет, с одной стороны, сохранять добавляемые значения, а с другой — позволяет исполнять блок повторно для накопления новых значений.

### *4.4.1 XML-представление*

XML-описание блока-аккумулятора приведено в листинге 7 и выглядит весьма просто.

**Листинг 7. Описание блока «for»**

```
<acc name="acc1" rtype="org.pro...PetriNet"
  iname="in" oname="out"/>
```

Блок задается элементом `acc`, имеет имя «`acc1`» и накапливаемый тип данных (*t*) «`org.pro...PetriNet`» (сокращенный классификатор типа данных «сеть Петри» из соответствующего пакета). Имена входного и выходного встраиваемых портов задаются параметрами `iname` и `oname` соответственно, в примере для них заданы те же значения, что и выставляемые по умолчанию.

### *4.4.2 Пример использования*

В качестве демонстрации использования блока-аккумулятора усовершенствуем пример, рассмотренный выше для блока «for» (см. листинг 6). Для этого добавим в тело цикла на выход блока alpha-miner блок-аккумулятор (см. рис. 12).

XML-представление данной схемы дано в листинге 8.

**Листинг 8. Модифицированный пример с циклом «for» и блоком-аккумулятором**

```
<for name="for1" from="1" to="5" step="1" iname="i">
   <body>
      <promtask name="am1" lib="prom_miner"
        libitem="AlphaMiner1">
      <acc name="acc1" rtype="org.pro...PetriNet"
        iname="in" oname="out">
      <connector name="int_con1" outp="am1.pn"
        inp="acc1.in">
   </body>
   <ports>
      <in name="log" dtype="org.pro...XLog"/>
      <out name="pnets" dtype="org.pro...PetriNet[]"/>
   </ports>
   <proxy>
      <connector outp=".log" inp="am1.log"/>
      <connector outp=".i" inp="am1.a"/>
      <connector outp="acc1.pnets" inp=".pnets"/>
   </proxy>
</for>
```

Алгоритм исполнения блока «for1» теперь выглядит следующим образом. На каждой итерации цикла осуществляется сброс схемы тела блока, то есть сброс блоков, содержащихся в ней: это блок-задача «`am1`» и блок-аккумулятор «`acc1`», который, однако, сохраняет предыдущие накопленные значения. При исполнении тела цикла первым всегда исполняется блок «`am1`», для которого удовлетворены входные зависимости (проключаемые прокси-коннекторами с входных интерфейсных портов блока-цикла). После исполнения этого блока на его единственный выходной порт кладется результат в виде объекта «сеть Петри», после чего исполняется блок-аккумулятор, входные зависимости которого теперь также удовлетворены: на его единственном входном порту находится свежеполученная сеть Петри. Исполнение блока состоит в добавлении во внутренний массив новой сети.

После 5 итераций цикл завершается, а на выходной порт блока «`for`» проключается выходной порт «`out`» блока-аккумулятора, тип данных которого — «массив объектов типа сеть Петри», содержащего ровно пять сетей, по одной на каждую итерацию цикла.

Для дальнейшей работы с массивами объектов необходимо использовать также блоки специальных типов, например «`for_each`», выполняющие последовательную итерацию всех объектов массива с исполнением для каждого из них схемы — тела блока. Однако рассмотрение таких блоков выходит за рамки данной работы.

## 5. ЗАКЛЮЧЕНИЕ

В работе был рассмотрен новый язык DPMine/P моделирования предметно-ориентированных экспериментов Process Mining и инструментальная поддержка этого языка на базе платформы ProM.

В настоящий момент язык находится в стадии активного развития, поэтому многие элементы претерпевают изменения. Особенно этого касается синтаксических моментов и семантики новых блоков, в то время как основная концепция, изложен-

ная в разделе 2.2 является фундаментальной и направляющей с точки зрения введения нового функционала.

Было приведено описание некоторых блоков и примеров их использования. Разумеется, это только малая часть всей палитры функциональных блоков, полное описание которых скорее было бы похоже на справочник по языку.

С точки зрения поддержки реализации языка на базовом инструменте ProM был рассмотрен только один из плагинов, осуществляющих непосредственно исполнение (интерпретацию) моделей. За рамками данной работы остались плагины для создания тестовых моделей, плагины — визуализаторы моделей и отчетов о выполнении, плагины — менеджеры библиотек XML и некоторые другие.

Из работы, которую предстоит сделать в будущем, можно отметить следующее. Во-первых, инструмент, как и язык, находится в постоянной разработке, и на момент написания статьи первая альфа-версия основного набора плагинов ожидает своей сборки. Основной рутинной работой является разработка семантики новых блоков, необходимых для поддержки все более сложных схем обработки данных, и, соответственно, их программная реализация. В частности, планируется введение ссылочной семантики, позволяющей передавать некоторым типам блоков в качестве параметра объект типа «схема» (в виде соответствующего блока «Схема»).

Наконец, наиболее существенной задачей на следующем этапе является разработка графического редактора, позволяющего подняться над уровнем XML-представления и создавать модели с использованием визуальных элементов из библиотеки блоков и коннекторов.

Также изучается потребность в разработке еще более высокоуровневого языка скриптового типа, позволяющего описывать модели с использованием синтаксиса, приближенного к классическим языкам программирования высого уровня. Для такого языка, соответственно, потребуется разрабатывать компилятор в XML-представление модели.

## 6. СПИСОК ЛИТЕРАТУРЫ


[1] Process Mining Manifesto / W. M. P. van der Aalst, A. Adriansyah, A. K. Alves de Medeiros [и др.] // BPM 2011 Workshops, Part I. T. 99. Springer-Verlag, 2012. C. 169–194.

[2] Petri Carl Adam, Reisig Wolfgang. Petri net // Scholarpedia. 2008. T. 3, № 4. c. 6477.

[3] Carmona Josep, Cortadella Jordi, Kishinevsky Michael. A Region-Based Algorithm for Discovering Petri Nets from Event Logs // BPM. 2008. C. 358–373.

[4] Aalst W.M.P. van der. Decomposing Process Mining Problems Using Passages // Applications and Theory of Petri Nets 2012 / под ред. S. Haddad, L. Pomello. T. 7347 из *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2012. C. 72–91.

[5] Shershakov S. DPMine: modeling and process mining tool // Proceedings of the 7th Spring/Summer Young Researchers' Colloquium on Software Engineering, SYRCoSE 2013. 2013.

[6] Jensen K., Kristensen L.M., Wells L. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems // International Journal on Software Tools for Technology Transfer (STTT). 2007. T. 9. C. 213–254.

[7] van der Aalst W.M.P., Weijter A.J.M.M., Maruster L. Workflow Mining: Discovering process models from event logs // IEEE Transactions on Knowledge and Data Engineering. 2003. T. 16. c. 2004.