HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

International Laboratory of
Process-Aware Information Systems
(PAIS Lab)

2013
CEE-SEC(R)
Software Engineering
Conference in Russia

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

Sergey Shershakov

Moscow 2013

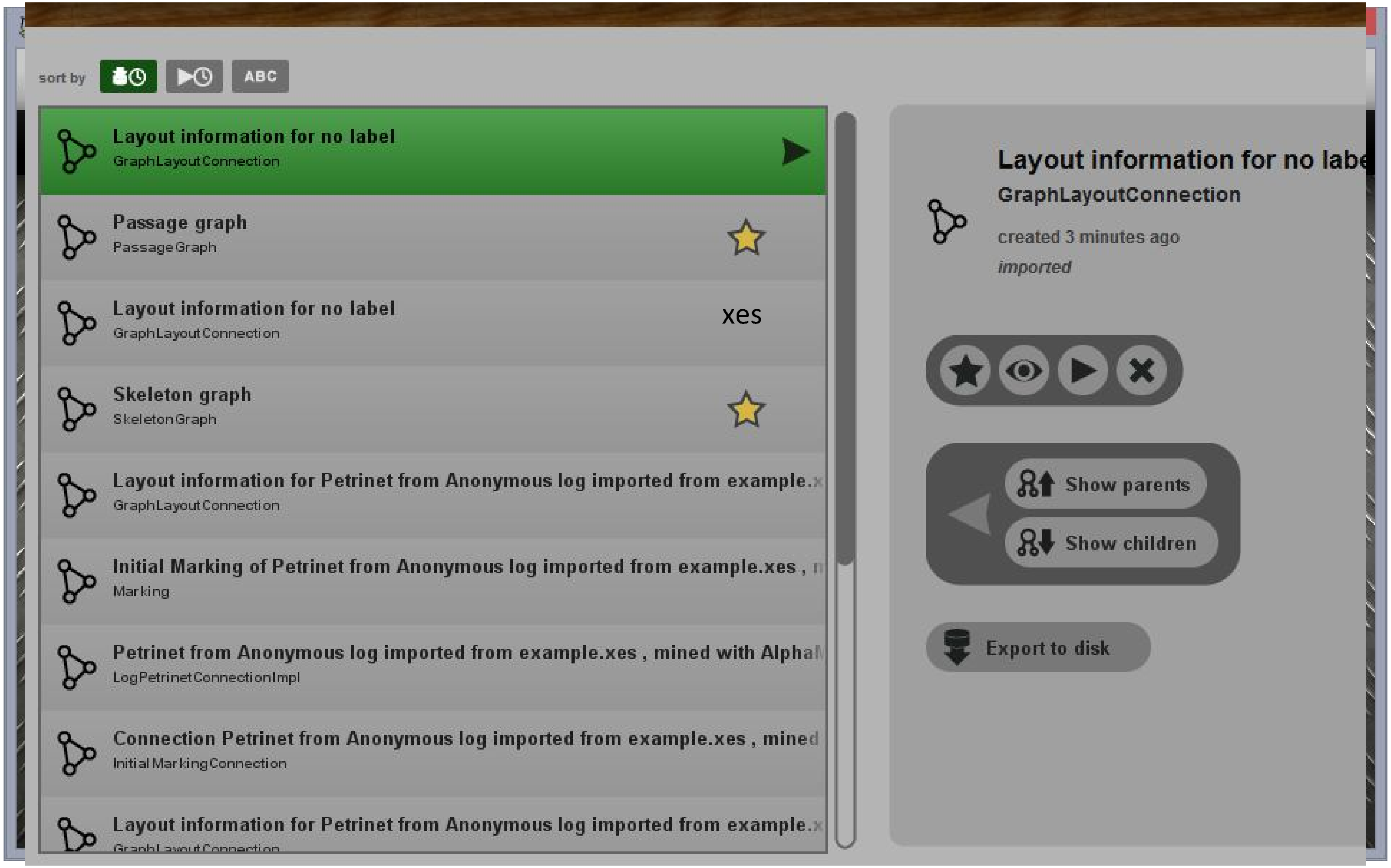
Outline

- Existing Tool: ProM
- Formulation of the Problem
- Approaches to Problem Solving
- Main Concepts
- ProM Representation of Models
- XML-based Language for Model Representation
- Extended Functional Concept
- Some of Block Types and Use Cases
- Work and Progress

Existing Tools

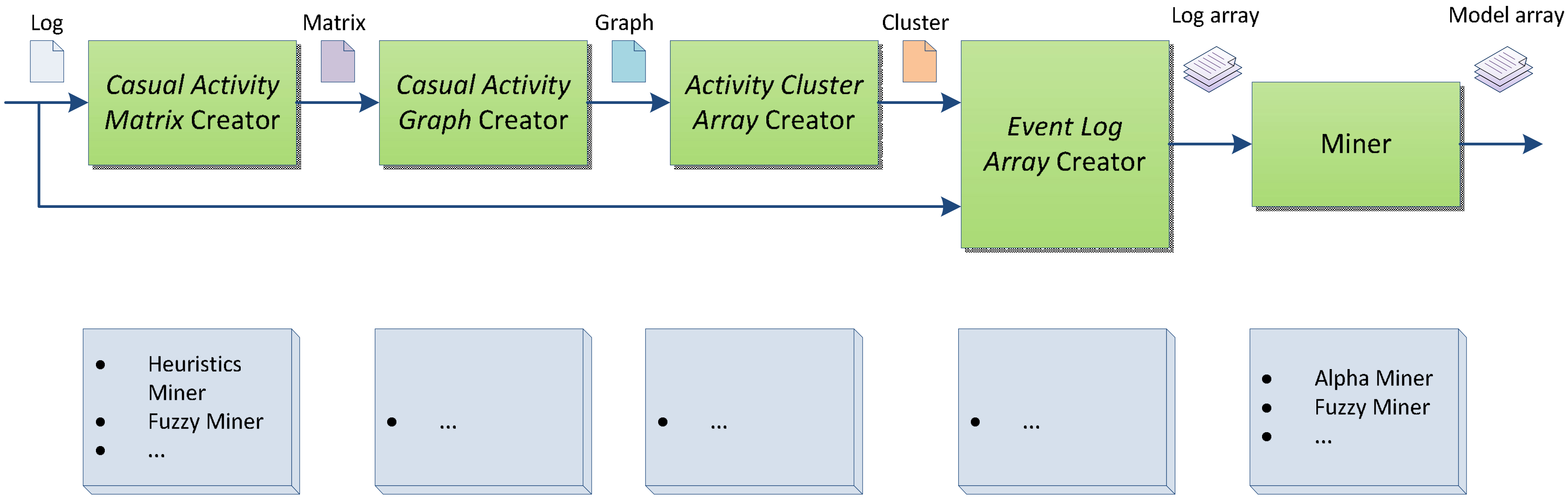
PROM

ProM



sort by
ABC
Layout information for no label
GraphLayoutConnection
Passage graph
PassageGraph
Layout information for no label
xes
GraphLayoutConnection
Skeleton graph
SkeletonGraph
Layout information for Petrinet from Anonymous log imported from example.
GraphLayoutConnection
Initial Marking of Petrinet from Anonymous log imported from example.xes ,
Marking
Petrinet from Anonymous log imported from example.xes , mined with Alpha
LogPetrinetConnectionImpl
Connection Petrinet from Anonymous log imported from example.xes , mined
InitialMarkingConnection
Layout information for Petrinet from Anonymous log imported from example.
Layout information for no labe
GraphLayoutConnection
created 3 minutes ago
imported
Show parents
Show children
Export to disk

Use case: large-scale experiments



Formulation of the Problem

- To develop a descriptive mark-up language for workflow on Process Mining; it should possess the following properties:
  - gathering stages of an experiment into a single sequence;
  - supporting control workflow for implementation of cycles and other required control elements;
  - …
- Implement the language on the basis of ProM tool

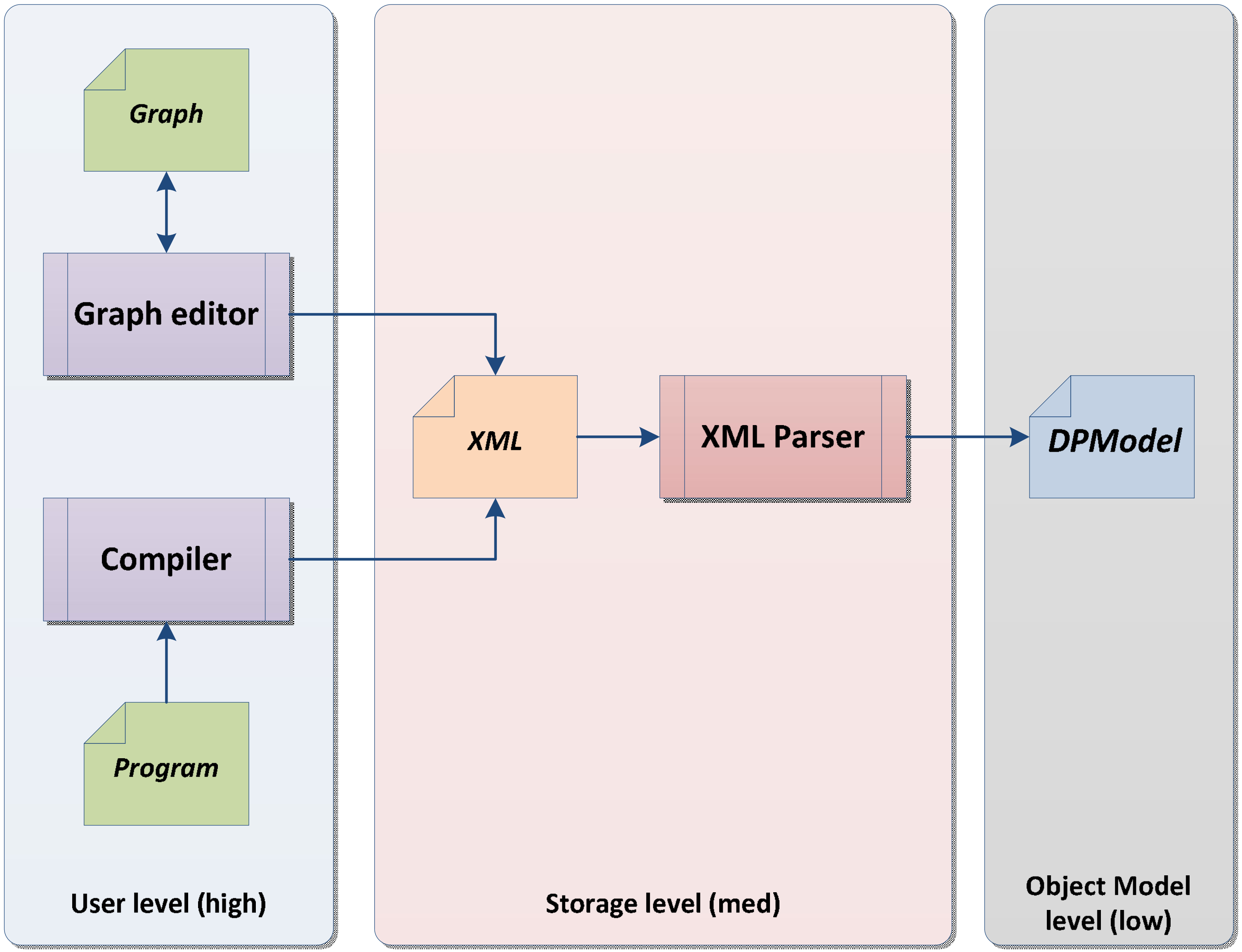
Basic prerequisites

- The language under development is considered from two levels — upper and lower
- Upper/intermediate level:
  - XML-based language itself (storage level);
  - graphics editor enabling the creation of workflow models in the from of graphic block elements and their compilation into an XML representation (user level);
- Lower level:
  - object model

Language representation (viewpoint) levels

- Object model, a representation of the workflow of a task being solved in computer memory, is based on the concept of *blocks* and *connectors*
- XML model is a basis for an object model and is used for its storage in a persistent experiment file
- Tools for derivation of an XML model, e.g. graphics editor

Preparation of an Object Model



Object model

CONCEPT OF BLOCKS, PORTS, CONNECTORS AND SCHEMES

Main idea

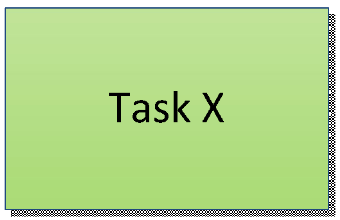
- Implementation of basic language semantics is done through the concept of *blocks*, *ports* and *connectors*
- Expansion of the language's functionality should be based on this very concept

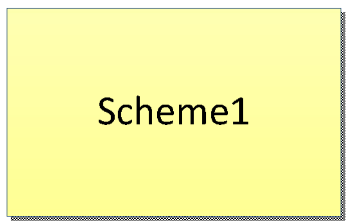
Blocks, ports and connectors

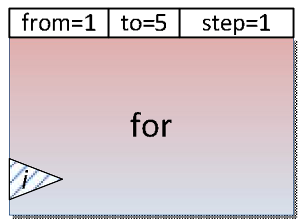
- *Block* — basic language building element; considered as a solitary operation in an external representation but can be complex in an internal one
- *Port* — linking object that belongs to a certain port and has the properties of *direction* and *data type*
- *Connector* — directed linking object connecting two blocks through their ports
- *Scheme* — multitude of interacting blocks connected with each other by connectors

Blocks

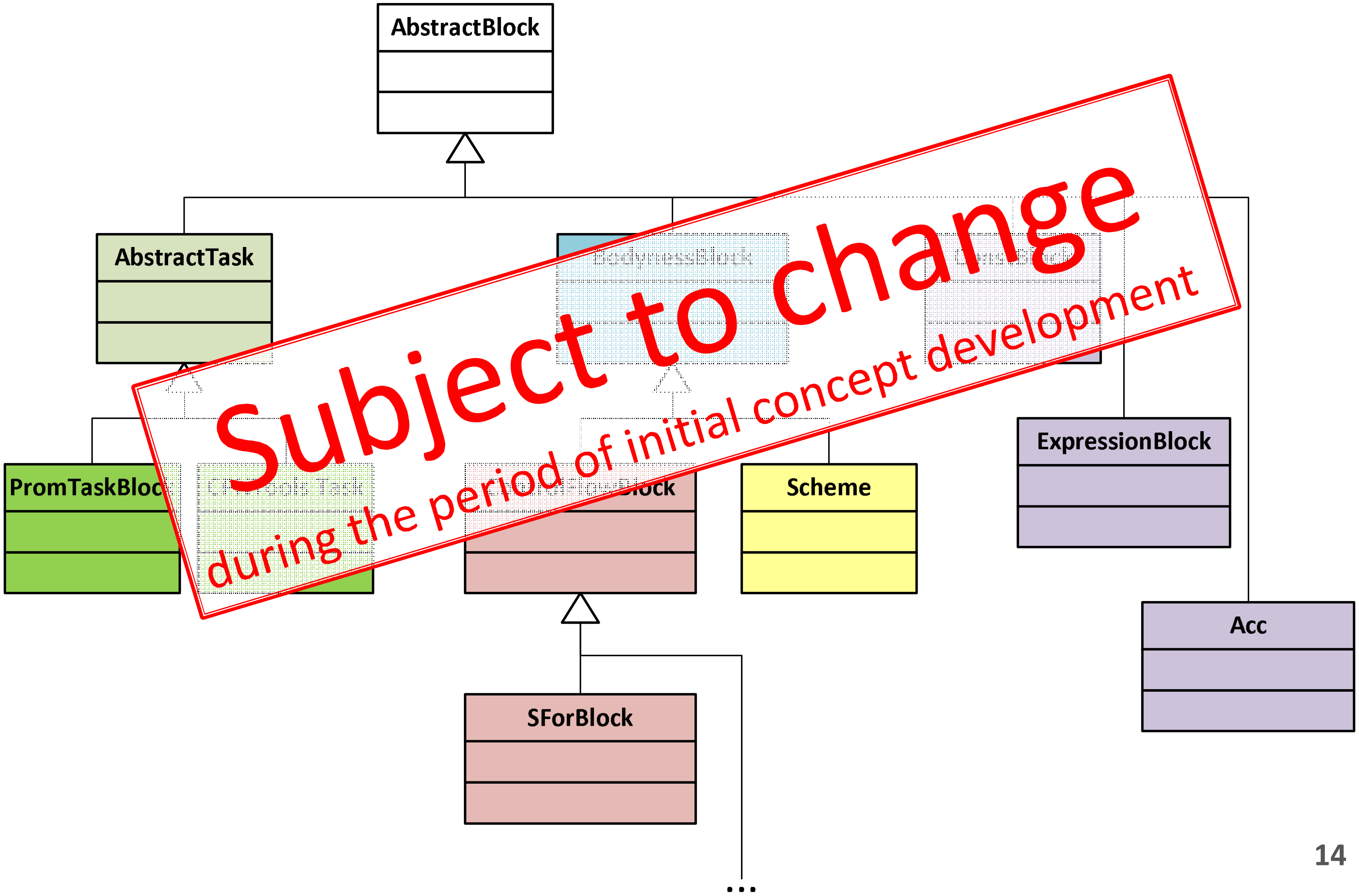
- Basic building element for schemes (and for models respectively)
- Perform a specific task
- Act like a statement in programming languages
- Blocks can have different functionality:
  - perform a single task of a base platform (task blocks);
  - represent complex schemes into single blocks (scheme blocks);
  - implement control workflow (control flow blocks);





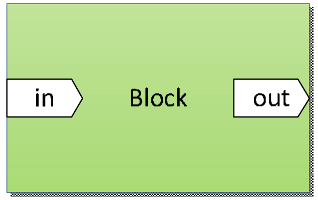


Block types hierarchy



Ports

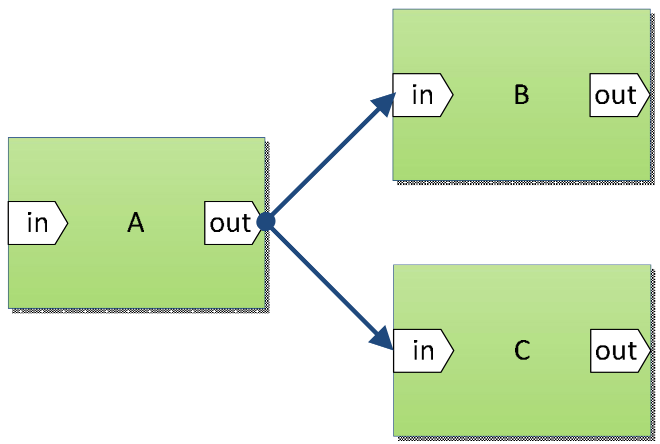
- *Port* — object belonging to a certain block and used for connection and data objects transfer to other ports
- Depending on direction, there are ports:
  - input
  - output
  - proxy (input-output and output-input)



- Transfer objects of a specific *data type*
- Depending on block type, can be either custom or built-in

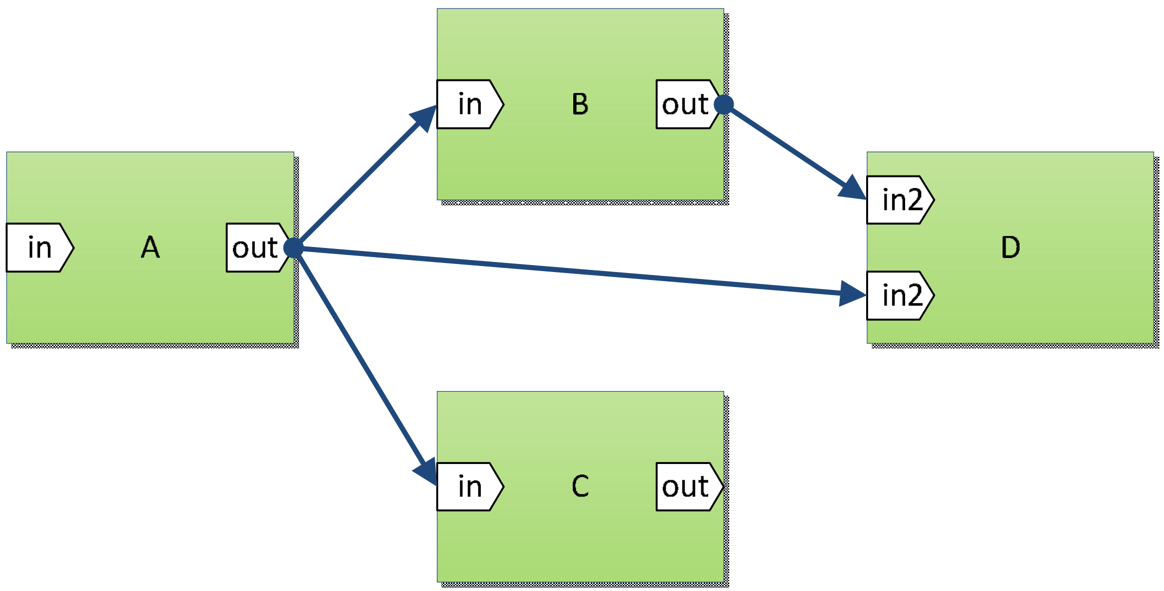
Connectors

- *Connector* — object connecting two blocks through their ports
- It has a *link direction*: a connector (with its *beginning*) always connects an output port of a block with an input port of another one (with its *end*)
- One output port can be linked to several connectors, whereas one input port can have only one connector linked



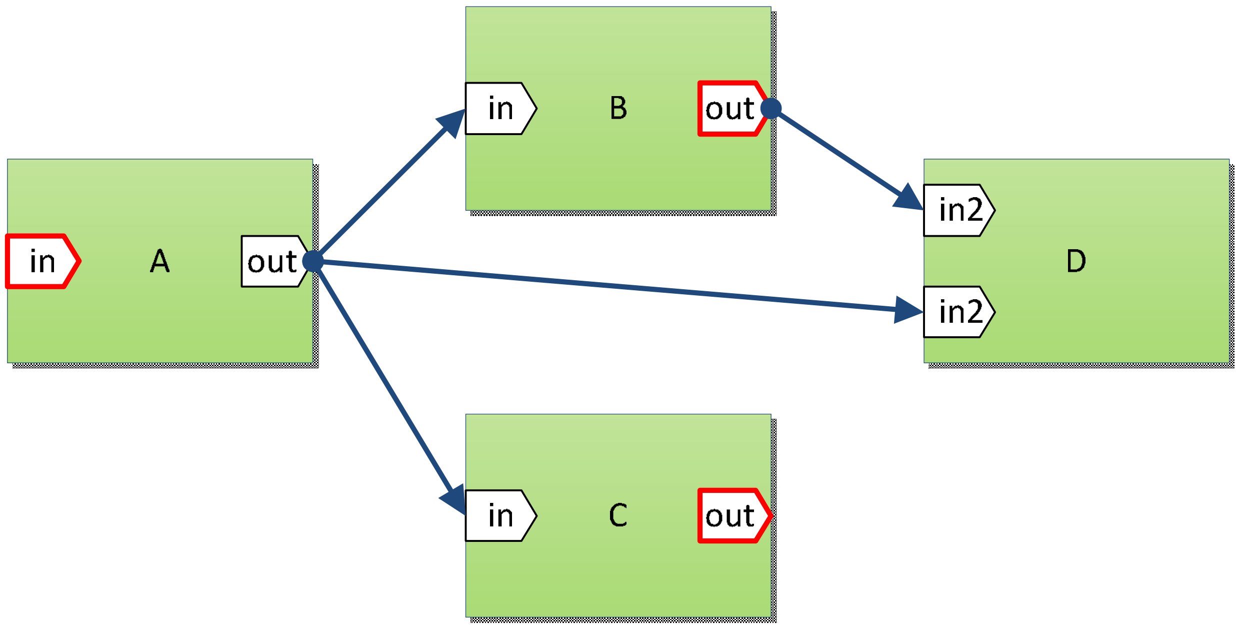
Scheme

- A number of interacting blocks connected with each other by connectors
- It is the main mechanism of implementing abstraction, isolation and hierarchy of sub-processes
- On the figure there is depicted a *connected scheme* consisting of four blocks (A, B, C, D) and four connectors (AB, AC, AD, BD)



Scheme interface

- Let us call *scheme interface* an arbitrary ports subset Ifp (called *interface ports*) within all the blocks' ports of the scheme
- On the figure below a scheme interface is as follows: $Ifp = \{A.in,\ B.out,\ C.out\}$ (whereby port in of block A is denoted $A.in$); the interface ports are in red

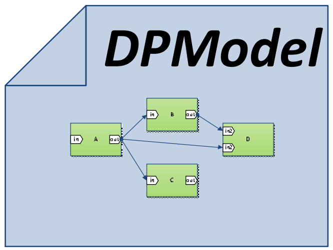


Implementation in ProM

MODEL AND ITS EXECUTION

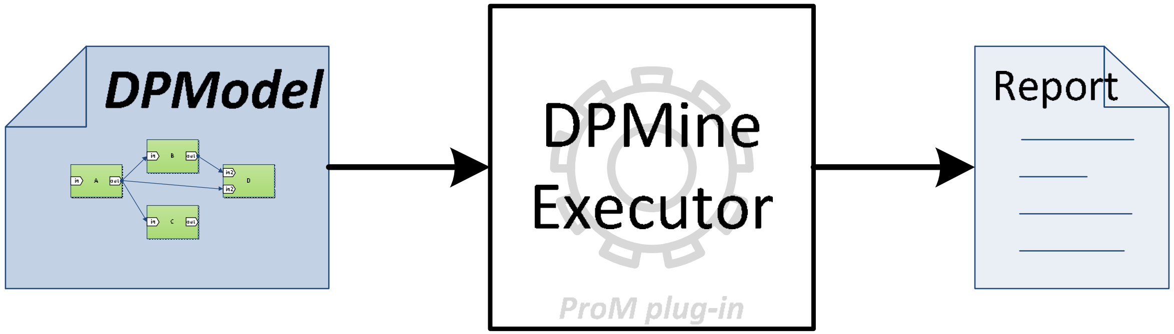
DPModel

- DPMine DPModel(/P) — workflow model represented by a data object (Java object) in ProM tool
- Contains an upper level scheme to be executed by *Executor* (ProM plugin)



Model execution

- *Model execution* consists in executing the main scheme of the model (upper level scheme) and producing an execution report (about errors, etc.)
- Model execution is done by a special agent — *Executor*, implementation of which is closely related to the base tool
  - In ProM tool, **DPMineExecutor** ProM plugin is the model *Executor*



Block “execution” concept

- *Block execution* is a sequence of operations done by an appropriate tool (e.g., DPMineExecutor plugin), that is *Executor*, with regard to a given block in conformity with its type and set of input parameters (at input ports of the block)

Block dependencies

- In order for the *Executor* to be able to execute a given block it is necessary that all the external dependencies of the block be satisfied
- For a given block *B* its dependencies are considered satisfied if:

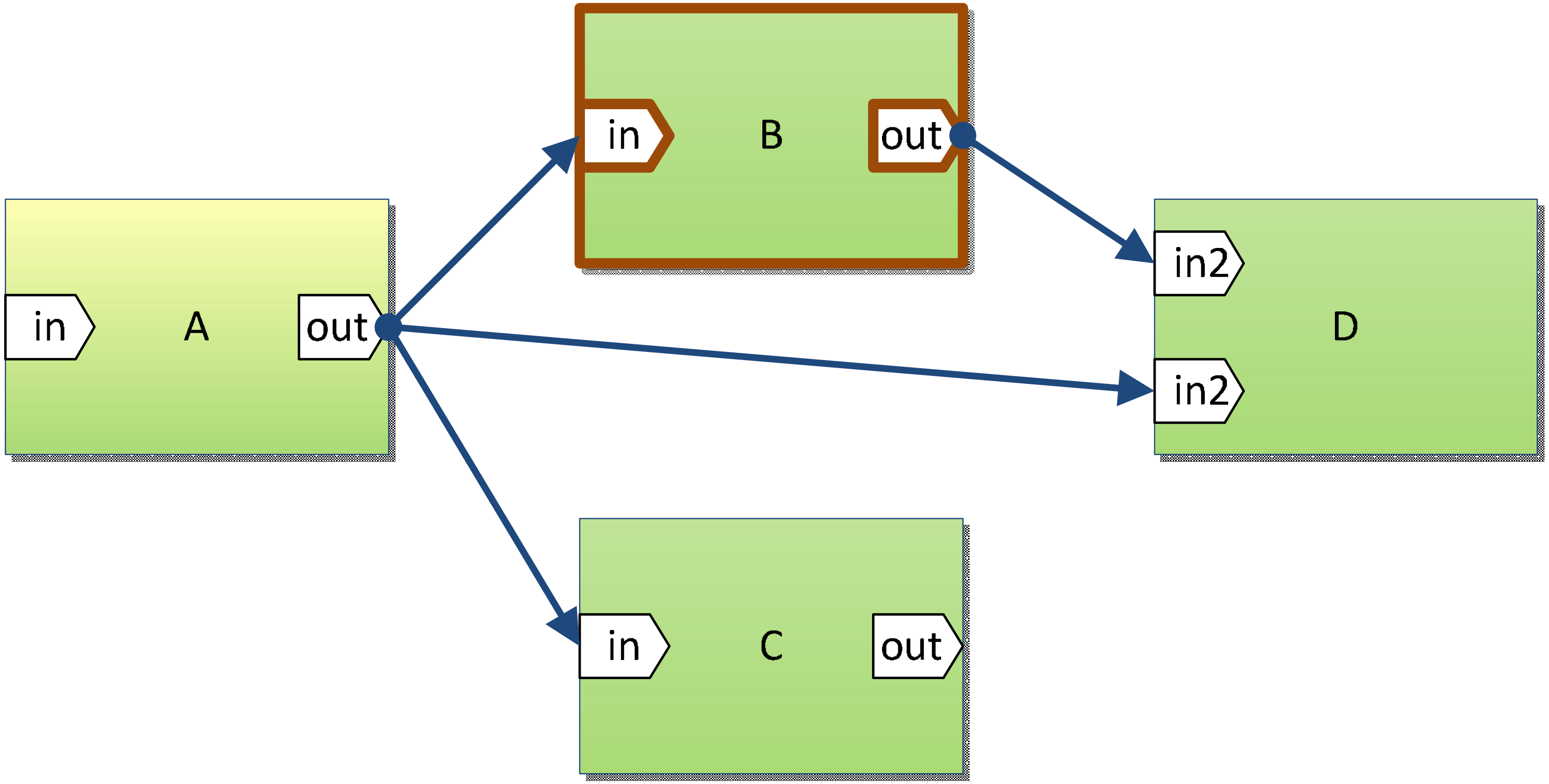
1. the block does not have input ports;
2. the block has input ports and for each port the following conditions are met:
   a) there is no “must be connected” flag for the port set, this way the port can be not connected by a connector to another (output) port of another block;
   b) the port is connected by a connector to another (output) port of another block and the status of this block is “executed”, which means there are data on its output ports that correspond to the types of these output ports

Some definitions regarding block execution

- *Executable block* — block for which the external dependencies are satisfied
- *Running block* — executable block which is currently executed by the *Executor*
- *Observable block* — block for which the *Executor* determines whether its input dependencies are satisfied
- *Unexecuted block* — block which has not yet been executed by the *Executor*; *executed block* — correspondingly, an executed block
- *Execution attempt* — selecting next block by the *Executor*, determining if it is executable (at that the block becomes *observable*) and in the positive case executing it (the block becomes *running*) — this is “*Block execution hit*” case; alternatively, the block is skipped and another block is passed on — “*Block execution fail*” case

Block execution

- If a block is *executable* (that is the input dependencies are satisfied), then the *Executor* calls a **corresponding block execution procedure** which is determined by the **block type** (and the block itself becomes *running*)



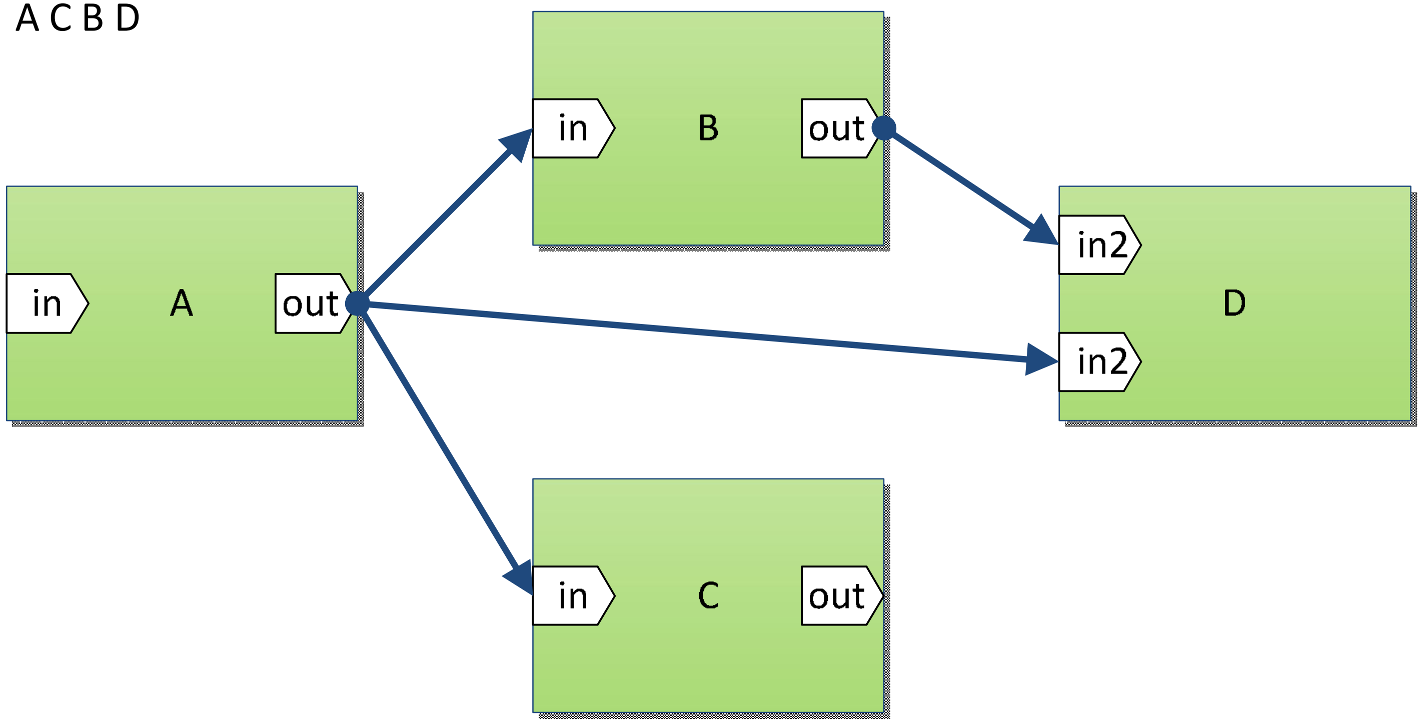
Execution sequence

- *Execution sequence* is a sequence determined by the order blocks are executed by the *Executor* under the condition they can be executed. The *Executor* can undertake several attempts to execute a given block, and in this case all these attempts, except for the last one, are considered failed in case for this block not all its input dependencies are satisfied during the attempts

- In other words,an *execution sequence* is a sequence of block execution *hits* made by the *Executor*

Execution sequence: examples

- For the given scheme the following executions (but not only they) are acceptable:
  - A B C D
  - A B D C
  - A C B D



- A sequence of blocks executed by the *Executor* is determined by the internal representation of the object model

Scheme execution

- The notion of execution is introduced for schemes by analogy with the block
- *Scheme execution* is a *sequence of execution* of scheme blocks
- If all the blocks contained in a scheme can change their state from “*unexecuted*” to “*executed*” in a finite number of steps, the scheme is considered *executable*
  - in other words, if all the blocks within a scheme can be executed, this scheme is *executable*

Language elements

XML DESCRIPTION

XML as language of model description

- XML is a means of stucturized description of a model and elements it comprises: schemes, blocks, connectors, etc.
- XML document of a model can be composed both manually and as output of a special editor, for instance that of a graphics one (work to be done)
- XML representation is also used for storing models in stand-alone files

XML model: example

```xml
<?xml version="1.0" encoding="UTF-8" ?>

<model name="MMain">
   <lib id="prom/base.dpmlib"/>
   <lib id="prom/petrinets.dpmlib"/>
   ...

   <scheme name="SMain">
      ...
   </scheme>

   <report filename="..."/>
</model>
```

Model description

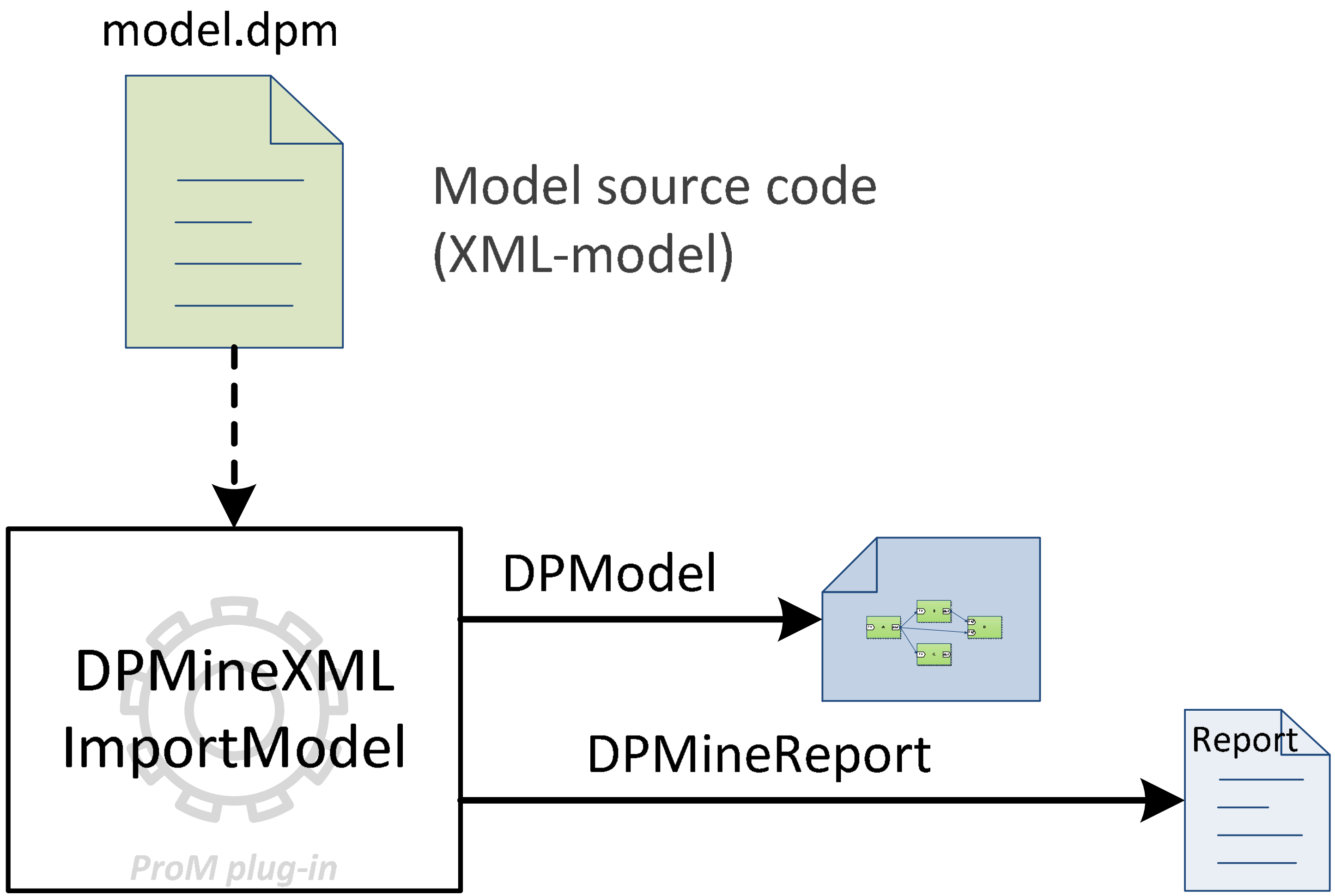
Including libraries related to tasks dealing with specific ProM plug-ins

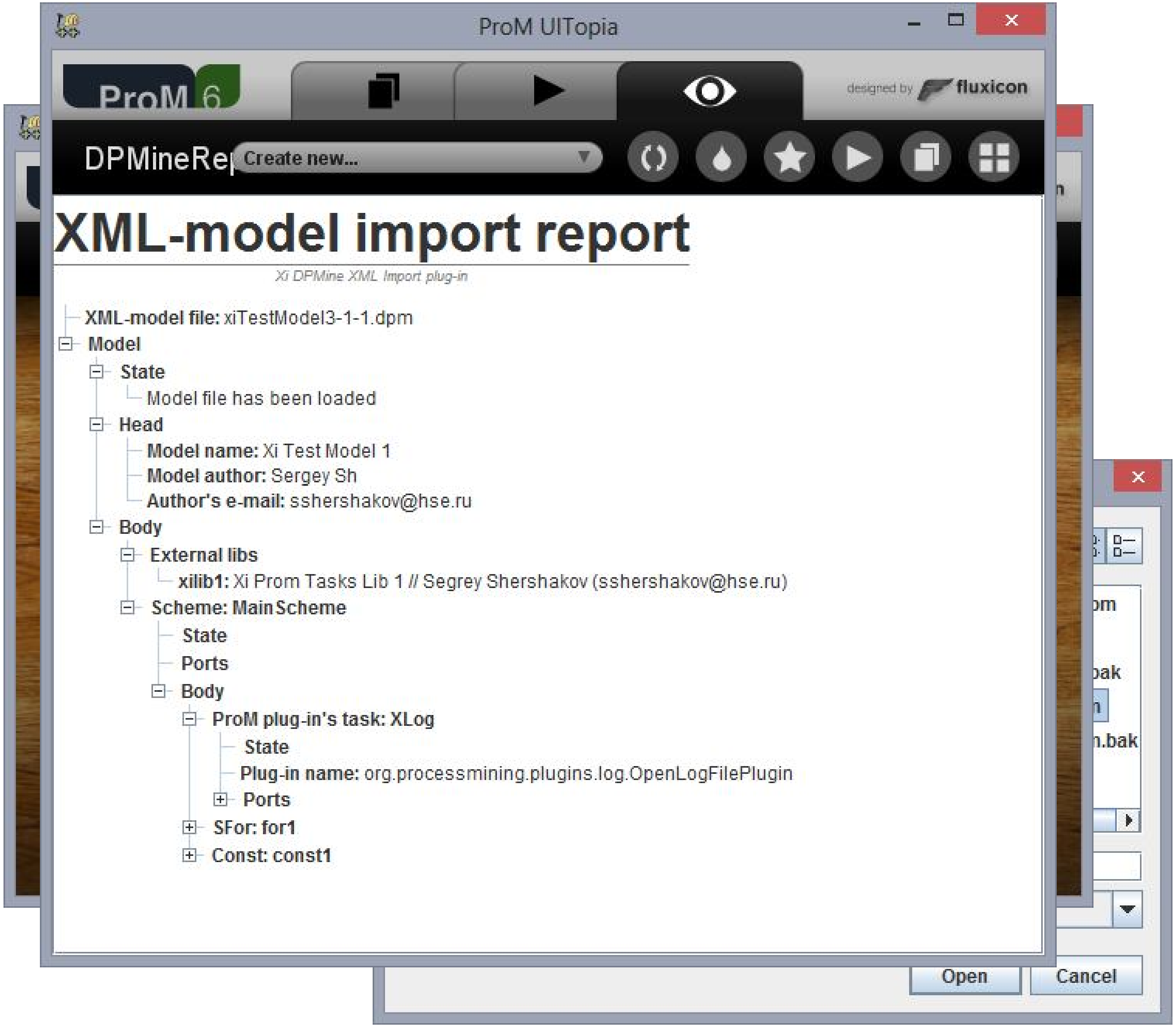
Main executable scheme

Model's additional parameters: report, etc.

XML description of specific blocks will be given while considering them

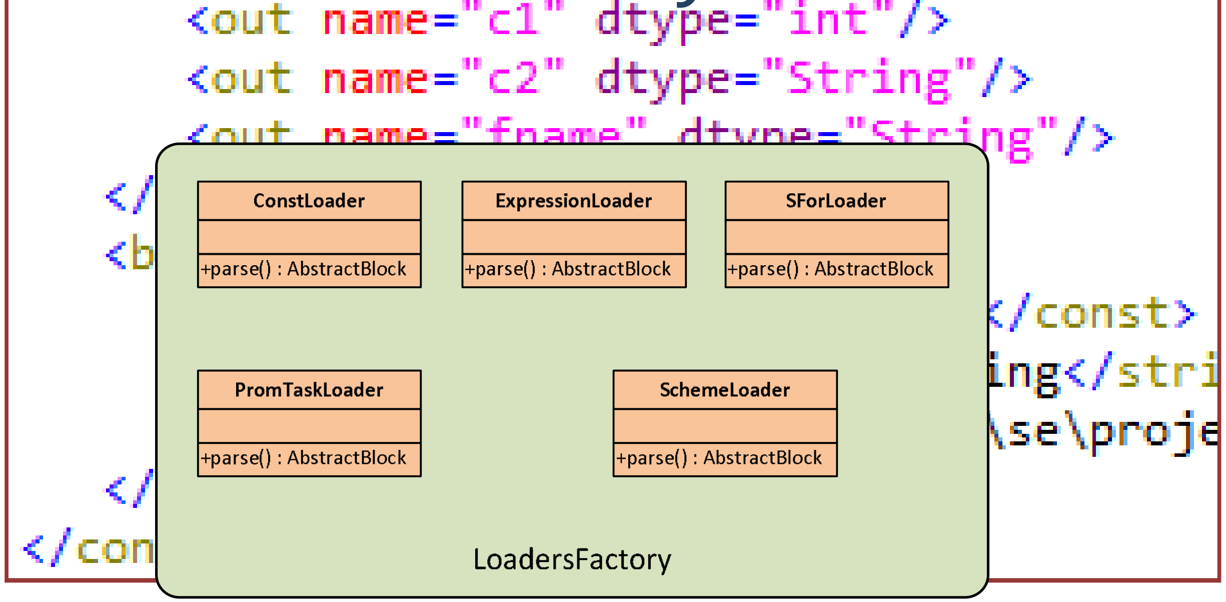
Model import

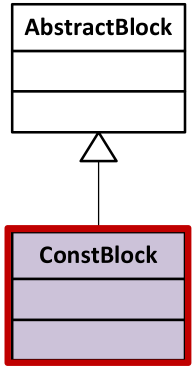


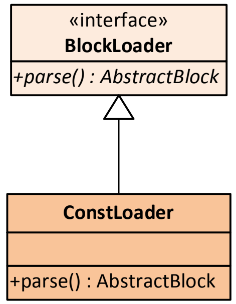

ProM UITopia
ProM 6
designed by fluxicon
DPMineRe
Create new...
XML-model import report
Xi DPMine XML Import plug-in
XML-model file: xiTestModel3-1-1.dpm
Model
State
Model file has been loaded
Head
Model name: Xi Test Model 1
Model author: Sergey Sh
Author's e-mail: sshershakov@hse.ru
Body
External libs
xilib1: Xi Prom Tasks Lib 1 // Segrey Shershakov (sshershakov@hse.ru)
Scheme: MainScheme
State
Ports
Body
ProM plug-in's task: XLog
State
Plug-in name: org.processmining.plugins.log.OpenLogFilePlugin
Ports
SFor: for1
Const: const1
Open
Cancel

Extended Functional Concept

1. Creating a new block class (`ConstBlock`)
2. Creating a block XML-description loader (`ConstLoader`)
3. Registering `ConstLoader` in `LoadersFactory`





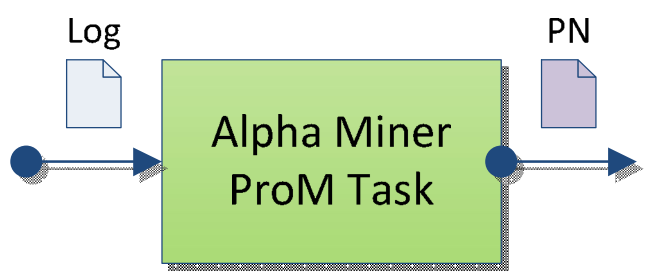


Language elements

BLOCK TYPES AND USE CASES

Task blocks

- Perform specific tasks related to the base tool, for instance ProM or CPNTool
- ProM Task block:
  - is bound with a specific ProM plug-in using annotation which contains plug-in’s name, method signature (or number) and so on;
  - contains ports according to plug-in’s invariant annotation;
  - execution of such a block leads to invocation of a ProM plug-in bound with it



ProM task: example 1

- XML description of a task calling AlphaMiner ProM plug-in:

Block type

Binding with ProM plug-in by FQI (fully qualified identifier)

Binding with a specific plug-in method (one of them)

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

Description of the input and output ports of the block with binding them with the input and output parameters

Tasks library

- Desc
- A me
  speci
  only

lib

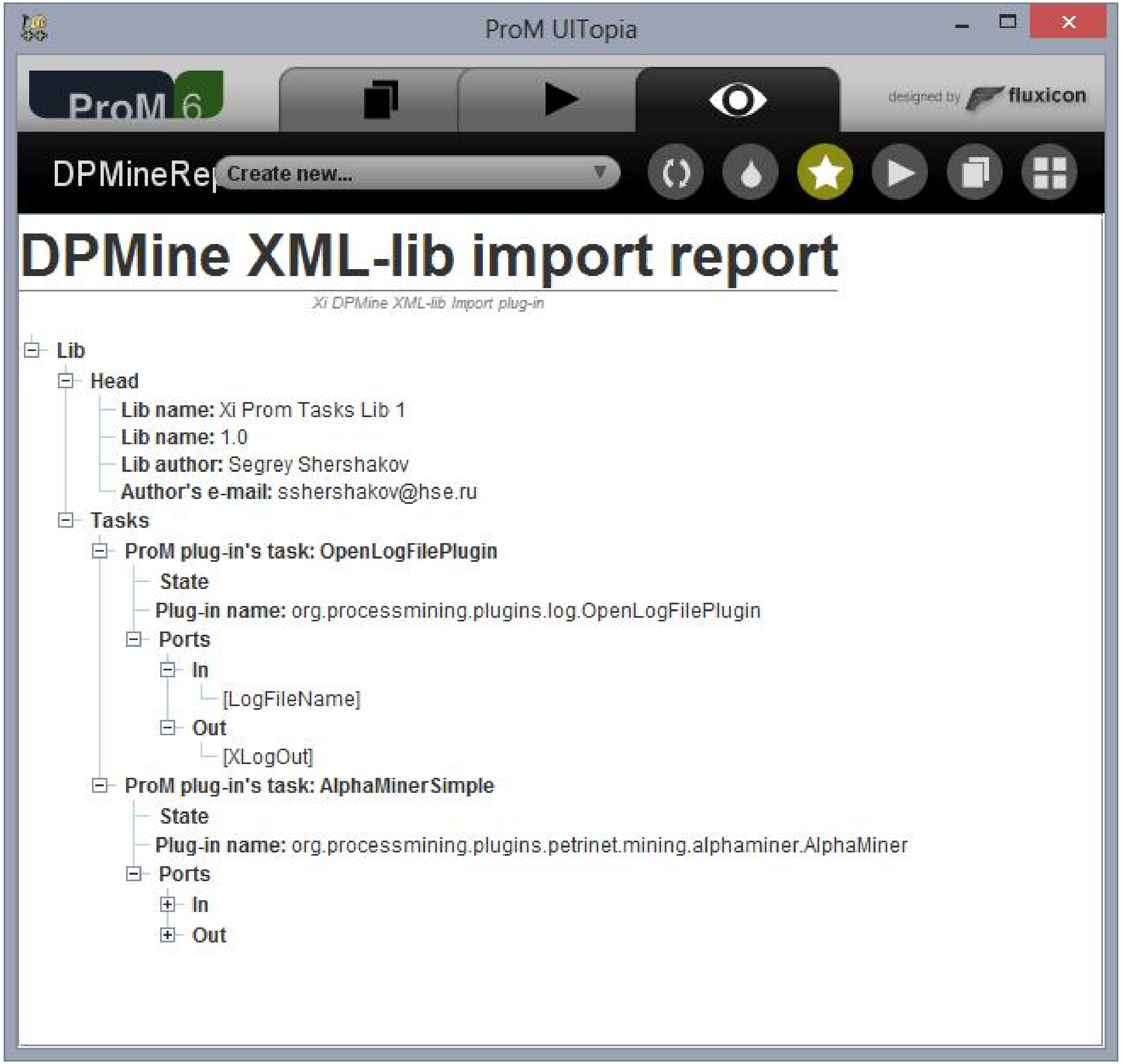
```xml
<?xml version="1.0" encoding="UTF-8" ?>

<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>
```

scheme

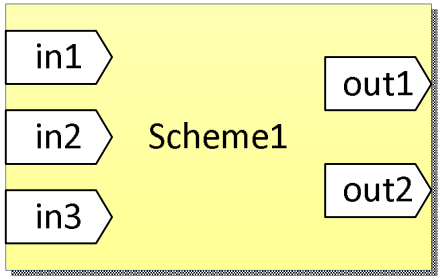
```xml
<?xml version="1.0" encoding="UTF-8" ?>

<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>
```

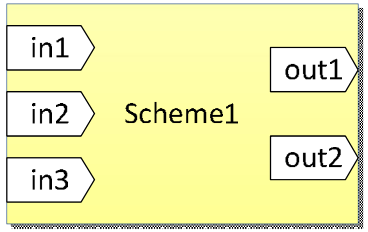

ProM UITopia
ProM 6
designed by fluxicon
DPMineRe|
Create new...
DPMine XML-lib import report
Xi DPMine XML-lib Import plug-in
Lib
Head
Lib name: Xi Prom Tasks Lib 1
Lib name: 1.0
Lib author: Segrey Shershakov
Author's e-mail: sshershakov@hse.ru
Tasks
ProM plug-in's task: OpenLogFilePlugin
State
Plug-in name: org.processmining.plugins.log.OpenLogFilePlugin
Ports
In
[LogFileName]
Out
[XLogOut]
ProM plug-in's task: AlphaMinerSimple
State
Plug-in name: org.processmining.plugins.petrinet.mining.alphaminer.AlphaMiner
Ports
In
Out

Scheme blocks

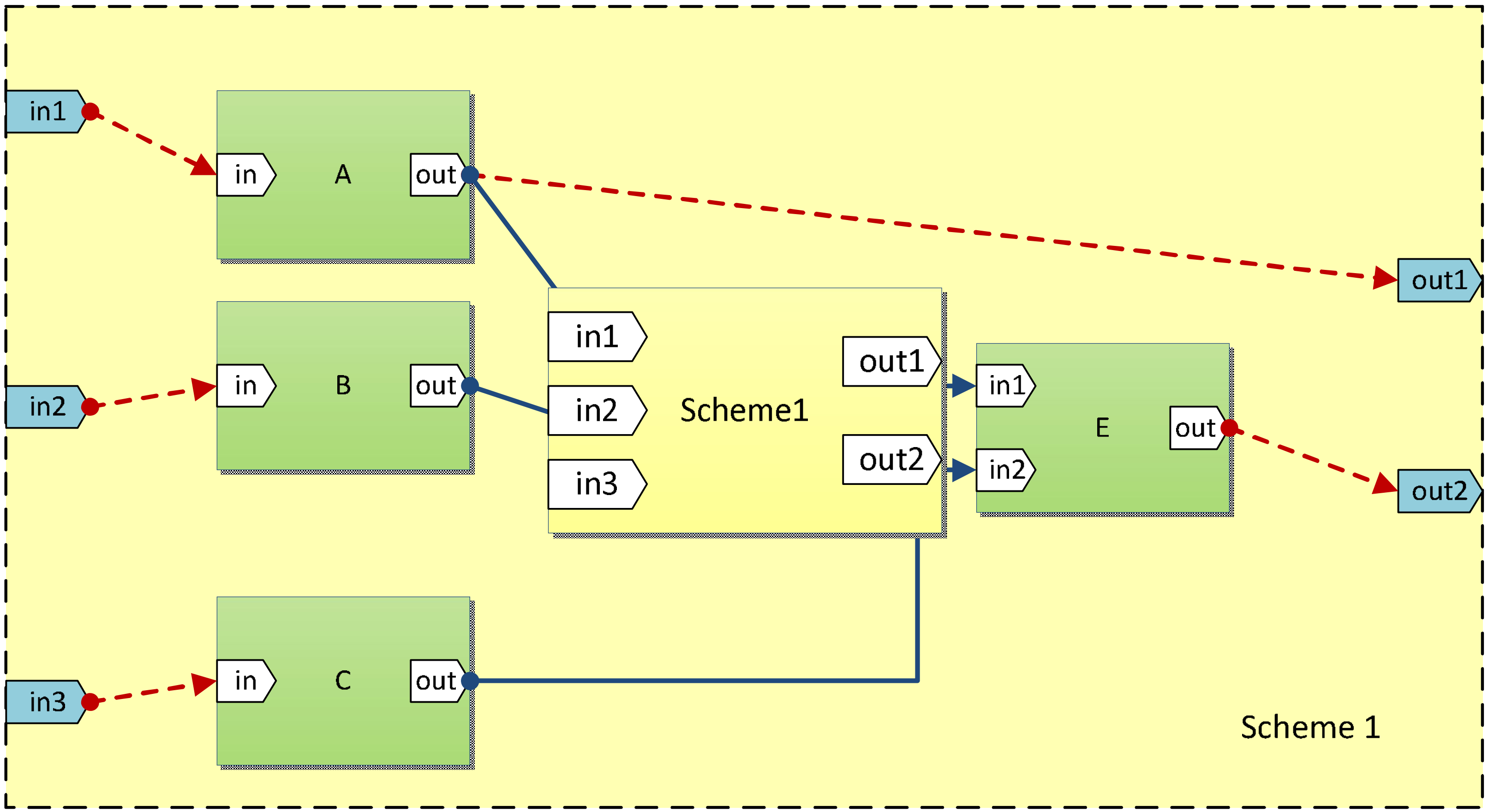
- *Scheme block* — block representing a nested system of blocks and connectors
- Necessary for hierarchical structurization of a model
- It is an analogue of the term “procedure” in programming languages
- At the **external level** a scheme block is a usual executable block that has *external ports* of given data types
- *Execution* of a scheme block at the external level is done according to the following principle common for all blocks: the only condition is the satisfaction of the input dependencies
  - after having executed a scheme block objects of corresponding types are put to its output port



Scheme blocks

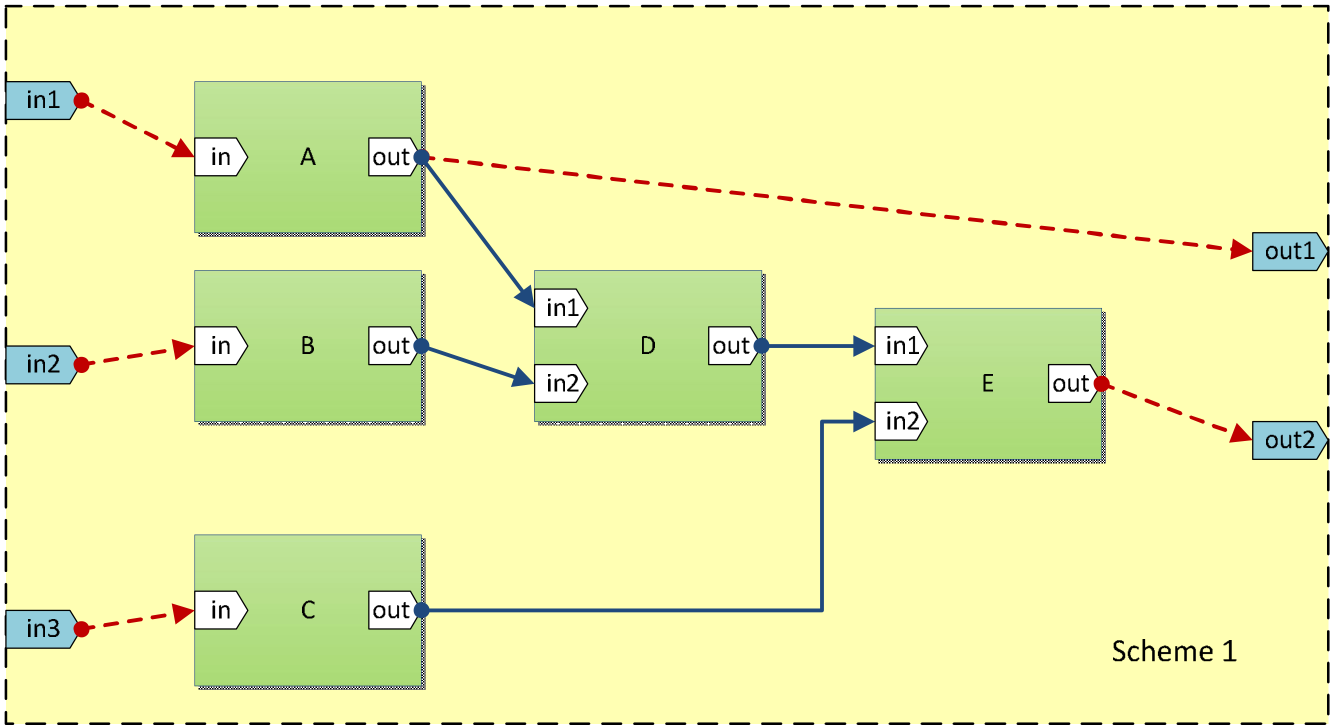


- At the **internal level** a scheme block, as it comes from its name, is a scheme consisting of blocks and connectors that connect them:



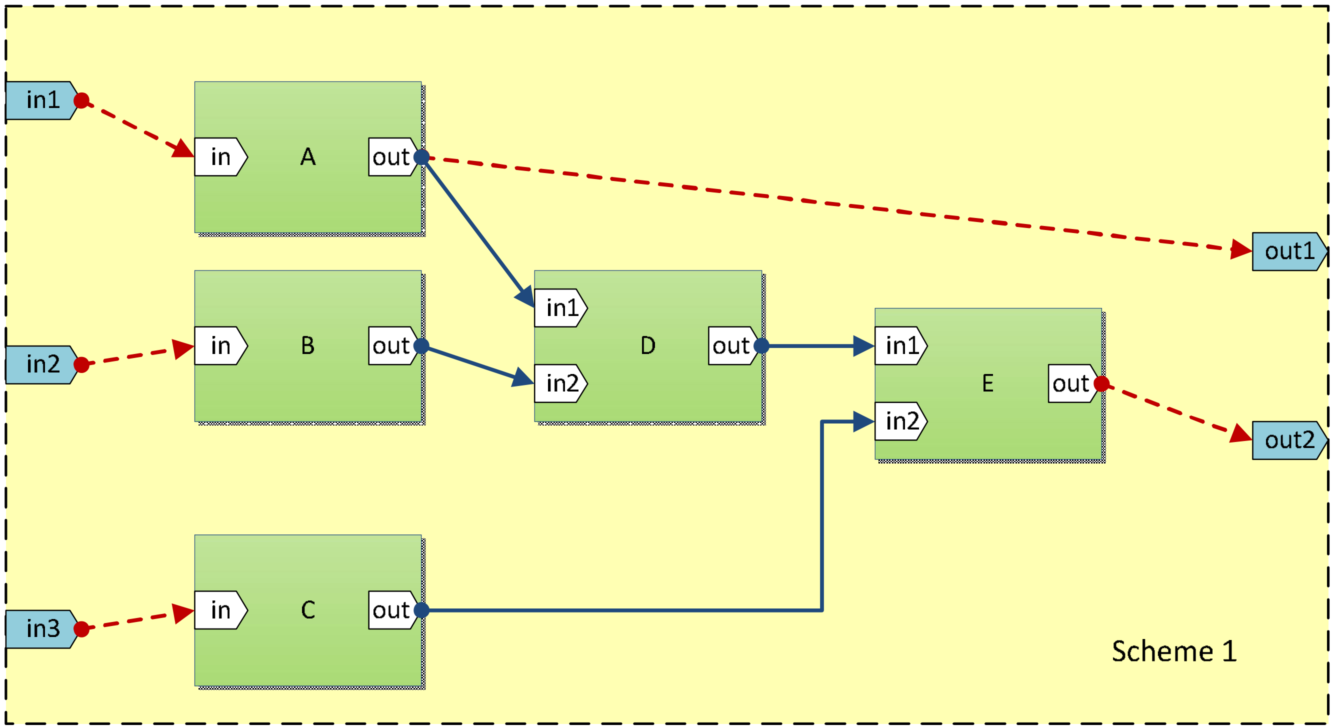
Scheme blocks: some definitions

- The blocks within a scheme are called *internal blocks*
  - the ports of such blocks are in white on the figure
- The connectors by which the internal blocks are connected strictly with each other are similarly called *internal connectors*
  - they are depicted on the figure by solid dark blue arrows



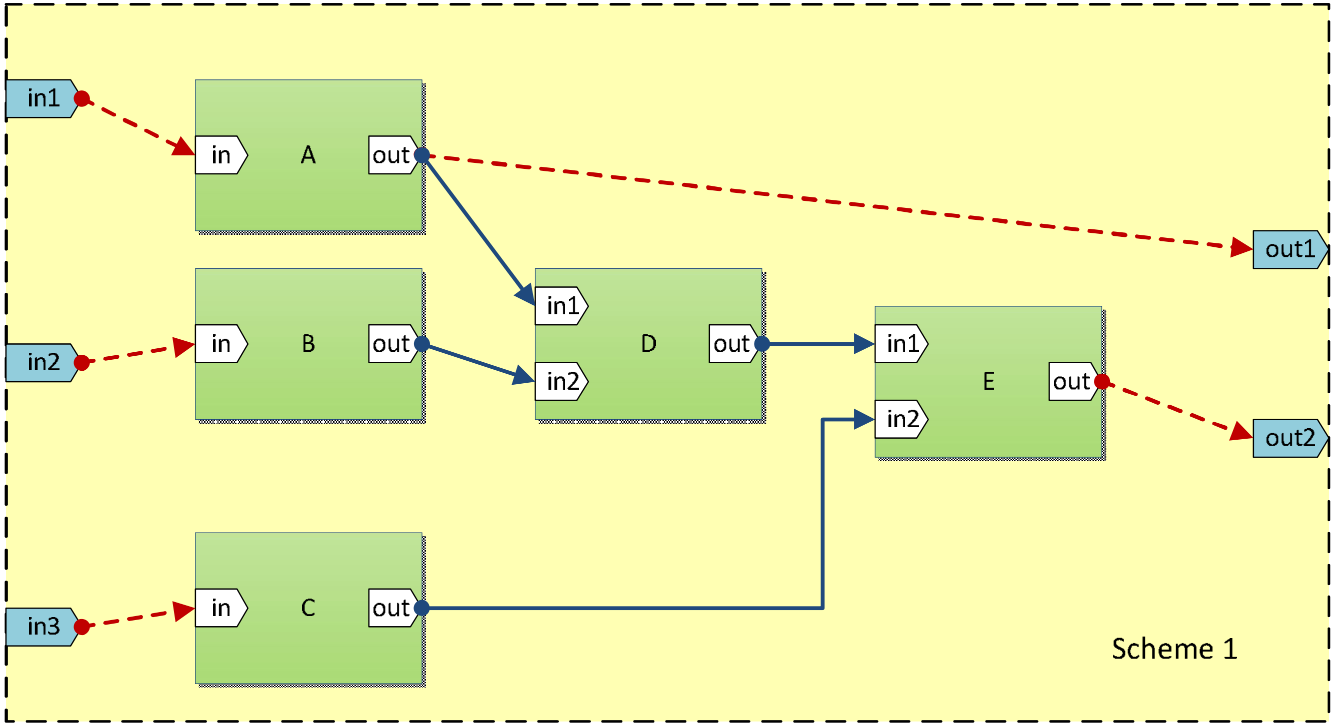
Scheme blocks: some definitions

- Connection of *external ports* of a scheme block (dark blue on the figure) with ports of *internal blocks* (*internal ports*) is also done by connectors, which, however, have a special function and name: *interface connectors* (or *proxy connectors*)
  - *proxy connectors* are depicted on the figure by red dashed lines
- The internal ports connected by proxy connectors with external ports are the [*scheme interface*]



Scheme blocks: execution semantics

- By *scheme block execution* one means a consecutive execution of the scheme's *internal blocks* for which the input dependencies are satisfied
- On the figure, for Scheme 1 the following sequences of execution of the internal blocks (but not only they) are acceptable:
  - A B D C E
  - A B C D E
  - C A B D E



Scheme blocks: XML representation

Scheme block

Connectors for inner blocks

Scheme body start

```
    <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>
```

blocks: kind
udo-blocks
specific
le

Scheme external (interface) ports

and the section to which they belong

Special section for proxy connectors

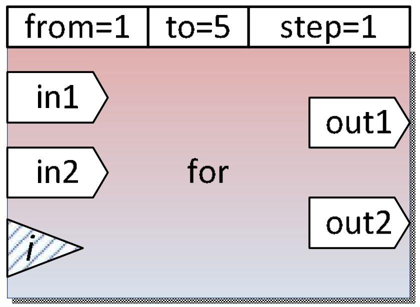
Proxy connectors: “name” attribute is optional

Notation “Block_name-dot-Port_name” for internal port referencing

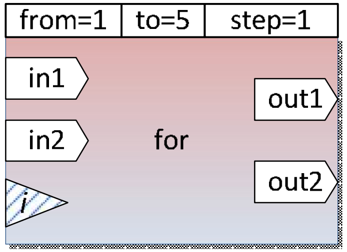
Special notation “dot-Port_name” for interface ports

Simple “for” block

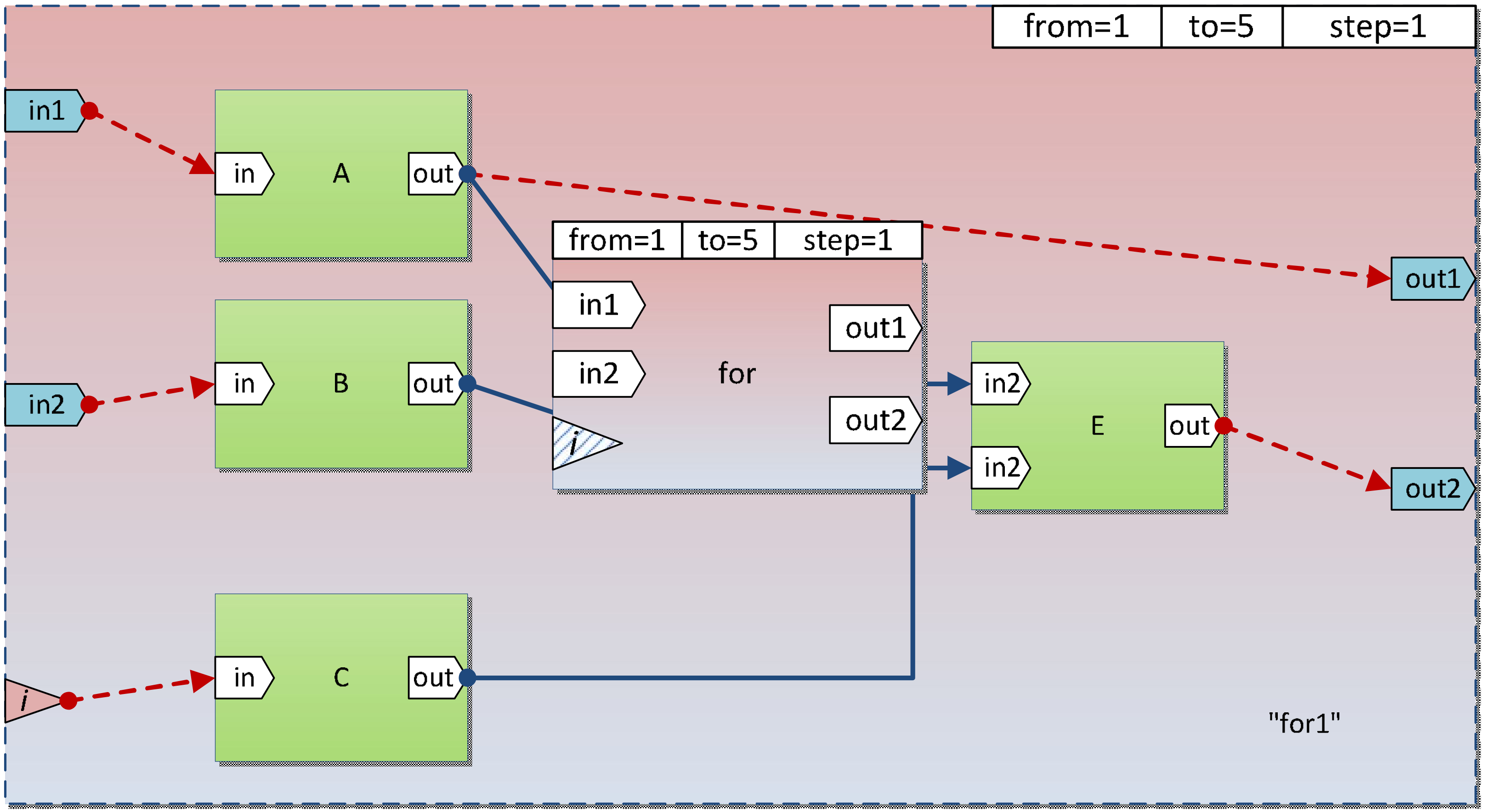
- It is used (in the simplest implementation) for integrating integral values *from*, *to*, with a specified *step*
- “For” block is a special scheme that is run repeatedly (iteratively) and according to the principle of the prior *reset* of the scheme’s elements before the next iteration
- It has a scheme body and “*i*” built-in port (depicted on the scheme by a dashed triangle) that has no external connection but is an input port with regard to the external interface of the block
  - port name is set through “iname” parameter, by default it has “*i*” value
- It enables (as in every scheme) to determine external (custom) interface ports and connect them in a required order with blocks of the scheme body



Simple "for" block



- Example of internal representation of a "for" block in a body with several blocks:



Simple “for” block: semantics

- “For” block runs the content of the scheme body a number of times depending on from-to-step conditions
- Each iteration means application of the *execute*() method for the blocks within the scheme body
- Already after the first iteration all the blocks of the scheme (and therefore the scheme as a whole) get to the “executed” state, which precludes the repetitive execution of the blocks (and the scheme) at next iterations
  - In order to avoid this while executing the next iteration of the “for” block the *reset*() method is applied for the whole scheme (and therefore all the block within) **before** each new iteration (including the very first one)

Simple “for” block: “i” built-in port

- Loop counter *i* is taken as a value that is through-connected to a pseudo-port "*i*" and can be used as an input value for any block (e.g., as a parameter or a mining algorithm)
- Resource type for this port is by default "int“ but choosing other types is possible, for instance "double“
- The incrementation semantics of iteration value *i* is similar to “for” cycle in Basic language (that is comparison with the upper limit is made according to “less-or-equal” condition, <=)

Simple “for” block: XML representation

Loop parameters

Built-in port – iterator name

```
<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>
```

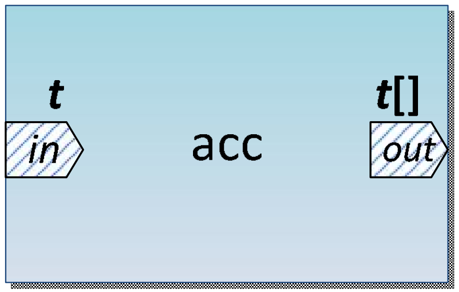
[AlphaMiner task]()

Interface ports

Proxy connectors. Port “i” is a built-in one

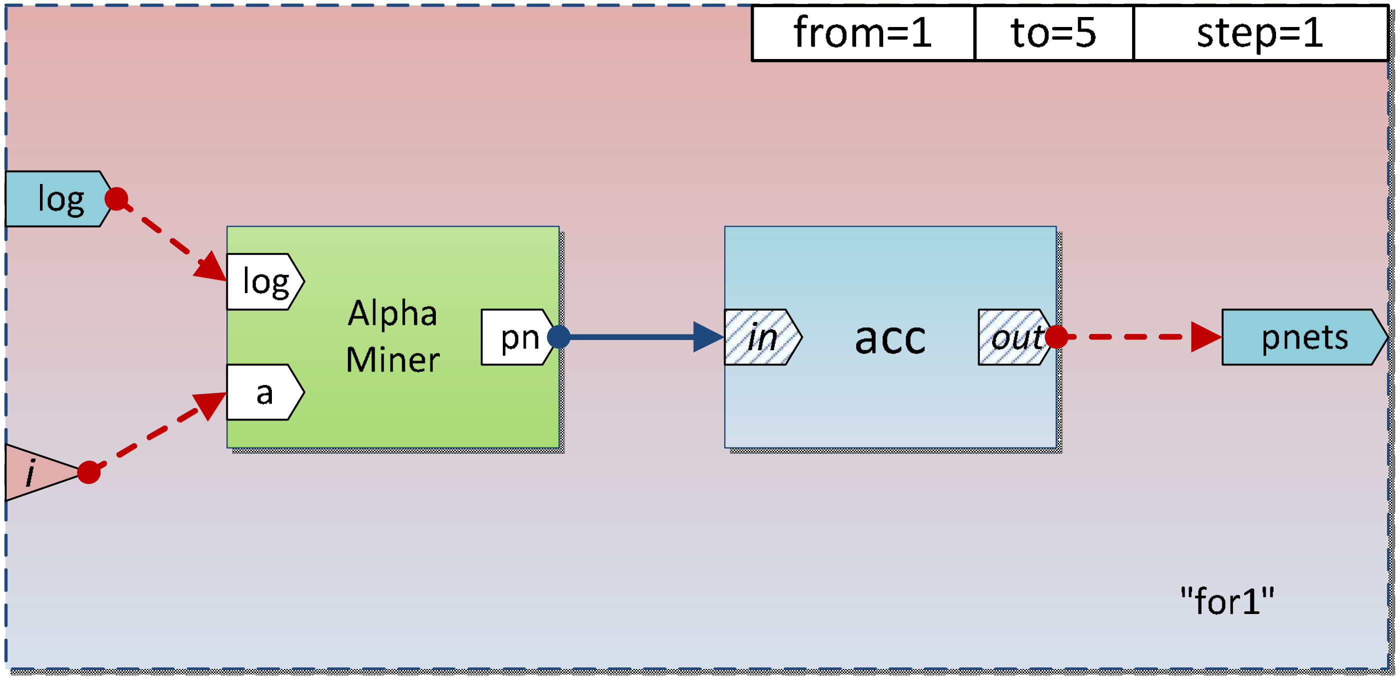
Acc block

- It is used for accumulating incoming values in an array
- The block has two built-in ports , one input "*in*" and one output "*out*", that have the types *t* and *t*[] (array) respectively, where *t* is to be set
- At each execution of the block (*execute*() method) it takes one value from *in* input port and appends it to an internal array, after that the block is marked as "executed"
  - at successive applications of *reset*() for the block the accumulated data from the internal array are not deleted, which enables the accumulation of the appended values
- XML representation:
  - ```
    <acc name="acc1" rtype="org.pro...PetriNet"
      iname="in" oname="out"/>
    ```


    ```

Use case 2

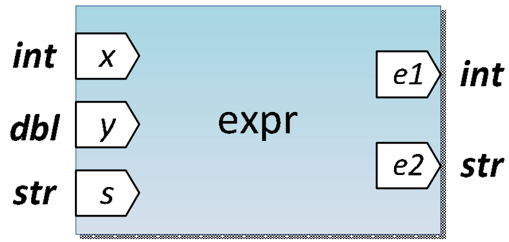
- Modified example from [this slide]:
  - Petri nets which are obtained after each iteration of a “for” cycle are stored by using an acc block

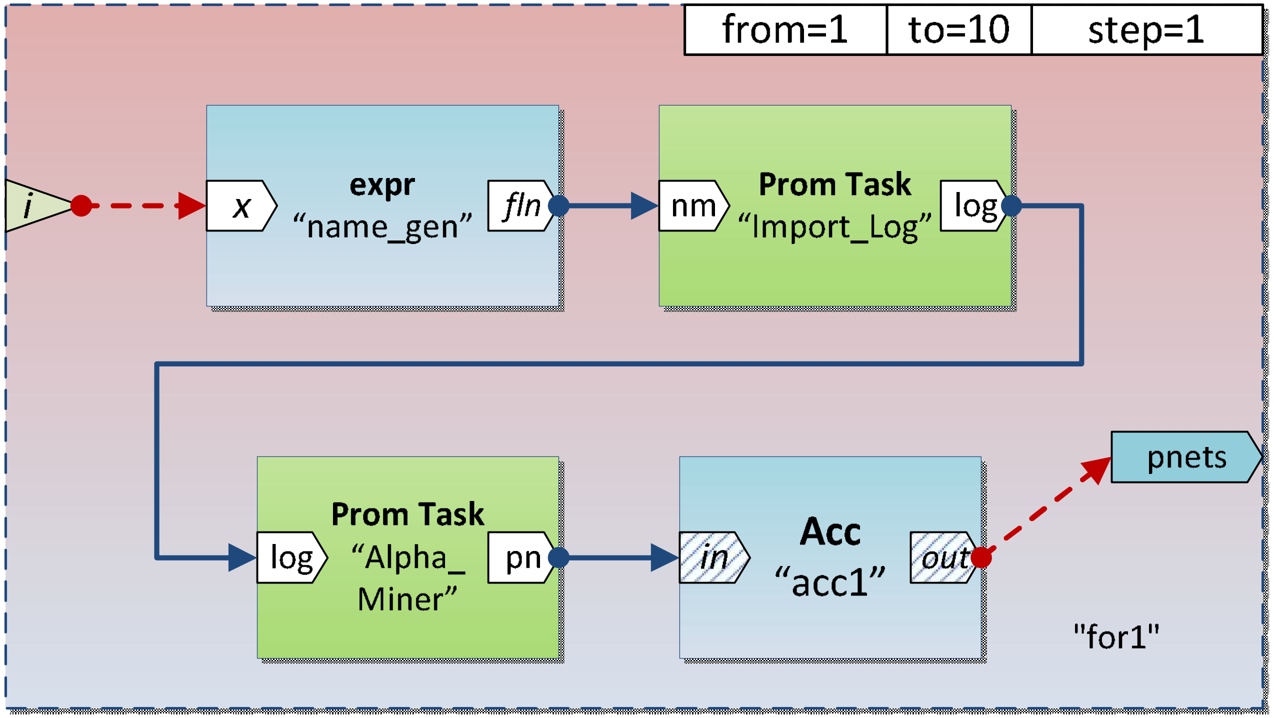


Use case 2

```xml
<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>
```

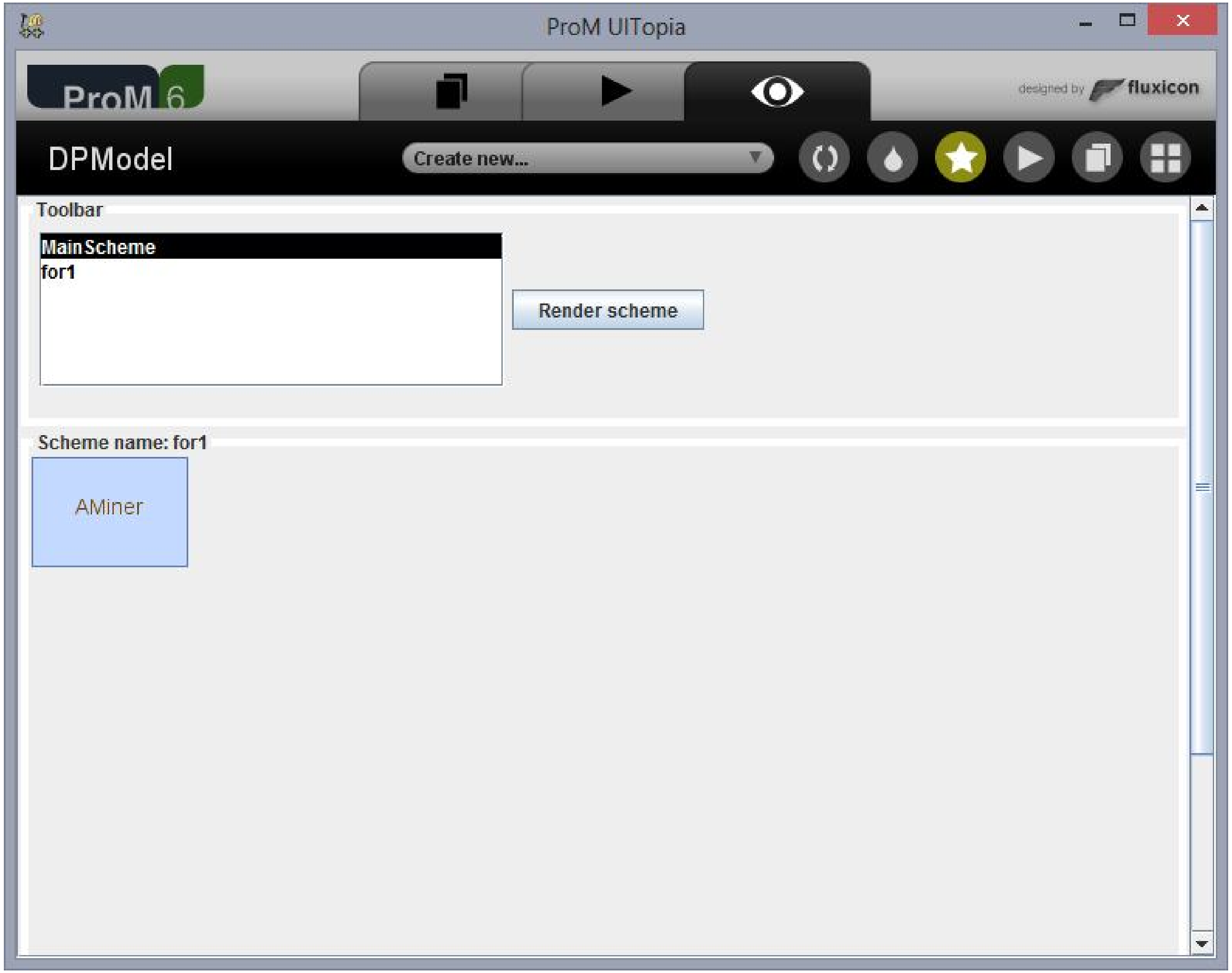
Expression Block





BLOCK EDITOR

Yet Another Graphical Tool



WORK AND PROGRESS

What has been done

- A concept of DPMine language has been developed
- The language has been implemented for ProM tool
- A number of basic blocks has been developed and implemented
- Work on the graphical model editor has been started

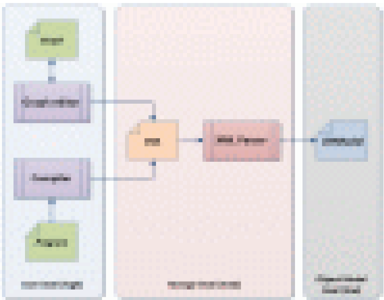
What to do next

- Comprehensive design of the graphical model editor
- Extending functionality by adding new blocks (as soon as need may be)
- Conducting a wide range of DPMine-based experiments
- Profiling the reporting subsystem

Contacts

http://www.hse.ru/staff/sshershakov

sshershakov@hse.ru



http://pais.hse.ru/research/projects/dpmine

Any questions?

THANK YOU!