SEPIA Tutorial

Documentation & Code Examples for the Security-oriented Petri Net Framework

IIG Telematics, University of Freiburg

Contents

1 Introduction

SEPIA stands for "Security-oriented Petri Net Framework" and provides implementations for various types of Petri nets. Along Place/Transition-Nets, it supports Petri nets with distinguishable token colors. To support information flow analysis of processes, SEPIA defines so-called IF-Nets, tailored for security-oriented workflow modelling, which enable users to assign security-levels (high, low) to transitions, data elements and persons/agents participating in the process execution.

For the usage in editors, Petri nets can be put in graphical containers, which hold visualization information. To preserve compatibility, Petri nets from other frameworks can be imported with the parser functionalities and also be exported for other frameworks using the serializing functionalities.

Additionally, the framework comes with classes for the traversal of Petri nets.

This document is a mix of programming tutorial and library documentation, where the features are demonstrated with some visualizations and minimal programming code examples.

The examples in this document have been tested against SEPIA release version 1.0.0, which can be found at `https://github.com/iig-uni-freiburg/SEPIA`. This document was last updated on July 29, 2015.

1.1 Library Dependencies

SEPIA builds upon the following tools. To use SEPIA, make sure all these libraries are included in the classpath.

- TOVAL, located at `https://github.com/GerdHolz/TOVAL`
- JAGAL (Java Graph Library), located at `https://github.com/iig-uni-freiburg/JAGAL`
- SEWOL (Security-oriented Workflow Framework), located at `https://github.com/iig-uni-freiburg/SEWOL`
- XML Schema Object Model (xsom), located at `https://xsom.java.net/`
- XML Datatypes Library (xsdlib)
- isorelax, located at `http://iso-relax.sourceforge.net/`
- hamcrest, located at `https://github.com/hamcrest`
- Multi Schema Validator (MSV), located at `https://msv.java.net/`
- relaxng-Datatype, located at `https://sourceforge.net/projects/relaxng/`

1.2 Package Structure

The packages in `de.uni.freiburg.iig.telematik.sepia` are logically divided into the following sub-packages. Methods of data structure defining classes like the Petri nets are always defined in abstract classes with Java Generics to keep constraints between data types and to allow users to easily define their own sub-classes. By using Java Generics, the types of the places, transitions, flow relations, and markings can be set as needed. Their concrete subclasses like `PTNet` or `IFNet` only contain their constructors and don't need to specify any data type over Java Generics. This way, clarity in the data structure and an intuitive depth of inheritance are ensured.

- The `petrinet` package contains both abstract Petri net definitions and concrete subclasses for P/T-Nets, CPNs, and IF-Nets.
- With the help of the container classes in the package `graphic`, Petri nets get complemented with graphical information, e.g. for the visualization in editors.
- The `traversal` package offers Petri net traversal functionalities.
- With the classes in the package `serialize`, Petri net objects can be exported in different file formats. This way Petri nets can also be edited in different editors.
- Serialized Petri nets can be parsed with the help of classes in the `parser` package.
- The `util` package contains helper classes for common Petri net properties/techniques such as reachability.

- The package `mg` contains marking graph implementations for all supported Petri net types.
- To replay traces on a Petri net, the package `replay` contains the appropriate classes.

2 Supported Petri Net Types

With P/T-Nets, CPNs, and IF-Nets, the SEPIA framework supports different Petri net types. P/T-Nets are basic Petri nets consisting of places and transitions. CPNs extend P/T-Nets by distinguishable token colors and IF-Nets add subjects to the net's context and add security clearances for token colors, subjects, and transitions.

An explicit definition of all supported net types can be found in the following publication:

> Thomas Stocker, Frank Böhr: *IF-Net: A Meta-Model for Security-Oriented Process Specification.* Security and Trust Management, Springer, 2013.

However, all of them build upon the same abstract class `AbstractPetriNet`. Using Java Generics the Petri net component types can be defined as subclasses of `AbstractPlace`, `AbstractTransition`, `AbstractFlowRelation`, and `AbstractMarking`. Figure 1 shows the inheritance structure of all Petri net classes.

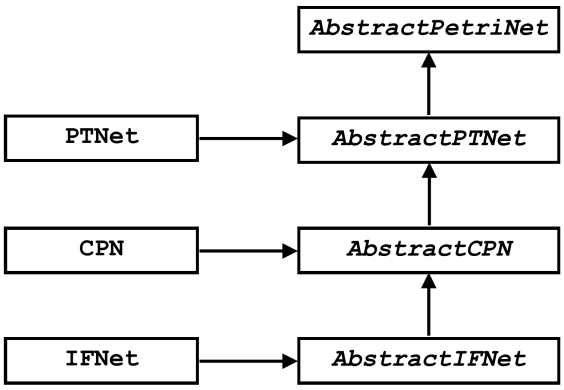

Figure 1: Inheritance hierarchy of the Petri net classes.

2.1 Petri Nets in SEPIA

All Petri net types have common properties and methods, which are outlined in this section.

To create new places, transitions, or flow relations, no new objects need to be instantiated "by hand". Instead net components are added to Petri nets by providing unique identifiers (name of places and transitions). The appropriate methods (`net.addPlace()`, `net.addTransition()`, ...) are called in the Petri net with the element's name as parameter. The instance of the net creates the respective element object. Flow relations are added by providing the identifiers of corresponding places and transitions. There are different methods for adding flow relations, depending on the direction (place to transition of vice versa). If a component's name doesn't exist or doesn't belong to the expected component type, an exception is thrown.

```
PTNet net = new PTNet();

// add elements
net.addPlace("p1");
net.addPlace("p2");
net.addTransition("t1");
net.addFlowRelationPT("p1", "t1");
net.addFlowRelationTP("t1", "p2");
```

In case additional properties of places or transitions have to be set, internally generated place/transition-objects can be requested from the Petri net itself (again with their identifier).

```
// set properties
net.getPlace("p1").setCapacity(5);
net.getTransition("t1").setLabel("Transition 1");
```

The initial state of a Petri net is set using a marking. Marking elements have different data types depending on the Petri net type. Markings of P/T-Nets have integer values representing the number of tokens per place, where markings of CPNs and IF-Nets have multisets of strings with the token colors. The markings are created, filled, and assigned to the Petri net object. The following code shows the marking of a P/T-Net, which assigns a number of tokens to specified places:

```
PTMarking marking = new PTMarking();
marking.set("p1", 2);
marking.set("p2", 1);
net.setInitialMarking(marking);
```

Flow relations have constraints, which specify the number of required tokens for a transition to fire for flow relations from places to transitions or the number of tokens to produce for flow relations from transitions to places. If all places in front of a transition contain enough tokens s.th. all constraints of the flow relations are satisfied, the transition is enabled. Enabled transitions can fire, where the specified number of tokens in the input places are consumed and tokens are produced in the output places. If the `fire()`-method of a disabled transition gets called an exception is thrown. The following code fires transition `t1` if it is enabled:

```
if (net.getTransition("t1").isEnabled())
    net.getTransition("t1").fire();
```

Each Petri net type defines validity and soundness properties, whereas validity refers to a correct net structure (i.e. a net specification which makes sense somehow) and soundness typically refers to workflow-specific properties of Petri nets. The soundness property implies a valid Petri net.

2.2 P/T-Nets: Place/Transition Nets

P/T-Nets are basic Petri nets, which can be represented as graphs with places and transitions as two different kinds of vertices. The vertices are connected with flow relations,

where places can only be connected to transitions and vice versa. Places can contain tokens and the assignment of tokens to places is called a marking.

The dynamic behaviour of a Petri net lies in the firing of transitions, which removes tokens from input places and generates new tokens in places connected via outgoing flow relations. The number of required tokens in input places for firing and generated tokens in output places are specified via constraints (number and type of transported tokens) of corresponding flow relations.

The following code sample shows how a P/T-Net consisting of two transitions and two places linearly connected by flow relations can be created.

```
PTNet ptnet = new PTNet();
ptnet.addPlace("p1");
ptnet.addPlace("p2");
ptnet.addTransition("t1");
ptnet.addTransition("t2");
ptnet.addFlowRelationPT("p1", "t1", 2);
ptnet.addFlowRelationTP("t1", "p2", 1);
ptnet.addFlowRelationPT("p2", "t2", 1);
```

As specified in the flow relation the transition requires two tokens in the place to be able to fire. For this some tokens must be added to the place by defining a marking.

```
PTMarking ptmarking = new PTMarking();
ptmarking.set("p1", 2);
ptnet.setInitialMarking(ptmarking);
```

Now the transition `t1` is enabled and can be fired:

```
if (ptnet.getTransition("t1").isEnabled())
    ptnet.getTransition("t1").fire();
```

This leads the place `p1` to contain no more tokens. Instead the transition created a new token in `p2`, s.th. the transition `t2` can be fired now.

Figure 2 shows the example P/T-Net.

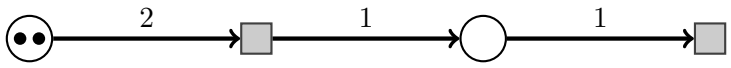

Figure 2: Visualization of the P/T-Net used in the example.

We can check for validity and get some topological information of the net. A P/T-Net is valid, if its structure makes sense, e.g. flow relations don't connect two places or vice versa. A valid P/T-Net is also sound. These properties can be verified using the corresponding property checker. They are either located in the sub-package `properties` in the package `petrinet` or in the sub-package of the respective Petri net type. The following code checks the validity and soundness of a Petri net and returns its boundedness. If the Petri net is not valid or not sound, an exception is thrown.

```
// Check validity (throws exception if invalid)
PTNetValidity.checkValidity(ptnet);
// Check soundness (throws exception if invalid)
```

```
PTNetSoundness.checkSoundness(ptnet, false);
BoundednessCheckResult boundedness = BoundednessCheck.getBoundedness(ptnet);
System.out.println(boundedness);
// Number of outgoing relations from the place:
System.out.println(ptnet.getPlace("p1").outDegree());
// Are there enough tokens in the input places?
System.out.println(ptnet.getTransition("t1").isEnabled());
```

The computation of some of the properties can be costly and tedious. Therefore some of the property checks can be executed asynchronously, by what it is not necessary anymore to wait for the programs response. For that the Observer Pattern is used. The asynchronous boundedness check can be performed by creating an appropriate listener, which is shown in the following code:

```
BoundednessCheck.initiateBoundednessCheck(ptnet, new
        ExecutorListener<BoundednessCheckResult<PTPlace,PTTransition,
        PTFlowRelation,PTMarking,Integer>>() {

    @Override
    public void executorStarted() {}

    @Override
    public void executorStopped() {}

    @Override
    public void executorFinished(BoundednessCheckResult<PTPlace,
            PTTransition,PTFlowRelation,PTMarking,Integer> result) {
        System.out.println( result.getBoundedness() );
    }

    @Override
    public void executorException(Exception e) {}

    @Override
    public void progress(double progress) {}
});
```

The method `executorStart` is called when the computation of the property gets started. If the execution was successful, the method `executorFinished` gets called.

2.3 CPNs: Colored Petri Nets

As an extension of the P/T-Net, tokens are differentiated with respect to their type. Different types of tokens are represented by different colors. According to that the resulting Petri nets are called colored Petri nets (CPN). CPNs define a default token color which is used for creating default constraints on flow relations. Figure 3 shows an example CPN, where a blue token is generated by the first transition and consumed by the second. The default token color in the example is `black`.

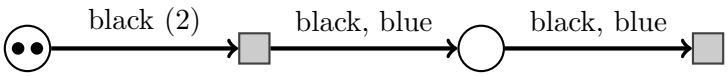

Figure 3: Visualization of the CPN used in the example.

This CPN can be created using the `CPN` class from the SEPIA framework. Note that a marking no longer contains an integer value per place, but a multiset in order to distinguish the number of tokens per token type.

Additional to the sensible structure, CPNs require flow relation effectiveness for validity. This means, that each relation must move at least one token from a place to a transition or vice versa. For the soundness property no more conditions must be met, s.th. a valid CPN is also sound.

In the same way constraints on flow relations contain a multiset of token colors which define the token colors that must flow. Firing rules make the creation of constraints more convenient. They are created per transition and contain multisets of tokens for the requirements and productions of a transition. Internally firing rules are translated into constraints. The following example creates a CPN and adds a constraint to one flow relation and uses a firing rule to set constraints to flow relations concerning a single transition.

```
CPN cpn = new CPN();
cpn.addPlace("p1");
cpn.addPlace("p2");
cpn.addTransition("t1");
cpn.addTransition("t2");
cpn.addFlowRelationPT("p1", "t1");
cpn.addFlowRelationTP("t1", "p2");
Multiset<String> c1 = new Multiset<String>();
c1.add("blue");
cpn.addFlowRelationPT("p2", "t2", c1);

FiringRule frT1 = new FiringRule();
frT1.addRequirement("p1", "black", 2);
frT1.addProduction("p2", "blue", 1);
cpn.addFiringRule("t1", frT1);

CPNMarking cpnmarking = new CPNMarking();
cpnmarking.set("p1", new Multiset<String>("black", "black"));
cpn.setInitialMarking(cpnmarking);
```

By default, all transitions consume and produce one token of the default token color, so this constraint can be omitted. Constraints with default token colors only need to be declared if more than one token should be consumed or produced.

Since `p1` initially contains two black tokens and `t2` requires two black tokens at its incoming places to be enabled, it can be fired.

2.4 IF-Nets: Information Flow Nets

The IF-Net complements the CPN by different security levels, access modes for transitions and tokens, and subjects, which are assigned to activities (transitions). We distinguish

between two security levels low and high. Subjects can be assigned a security clearance, transitions have a security classification, and tokens have an attribute classification represented as security level. Tokens with a high security classification are only allowed to be passed by activities with a high classification and only subjects with a high clearance are allowed to be assigned to such transitions as executor. This information is deposited in a so called labelling and access control model inside the IF-Net's analysis context. There's also a new type of declassification transition, which eases the security level by expecting a token with high classification and producing a token with low classification.

For example, a token with high classification can represent a construction plan in a company. Parts of the resulting product are produced by a third-party supplier, who shouldn't be able to see the complete construction plan. For him an adjusted version of the construction plan with low classification is generated, which can be modelled in an IF-Net via a declassification transition.

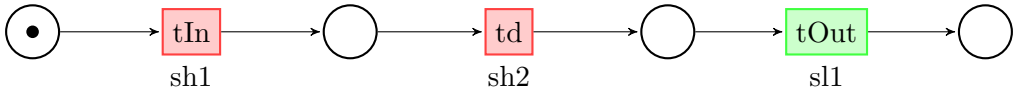

Figure 4: Visualization of the IF-Net used in the example. The transition `td` is a declassification transition. Both the red tokens and the red transitions have a high security clearance, where the green tokens and green transitions have a low security clearance. The subjects `sh1` and `sh2` have a high security clearance and the subject `sl1` has a low clearance.

As an example, we build the IF-Net shown in figure 4. It contains a declassification transition and two regular transitions, where one has a high and the other has a low security clearance. The corresponding Java code using the SEPIA framework looks like the following:

```
IFNet ifnet = new IFNet();

ifnet.addPlace("pIn");
ifnet.addPlace("p1");
ifnet.addPlace("p2");
ifnet.addPlace("pOut");

ifnet.addTransition("tIn");
ifnet.addDeclassificationTransition("td");
ifnet.addTransition("tOut");

IFNetFlowRelation f1 = ifnet.addFlowRelationPT("pIn", "tIn");
IFNetFlowRelation f2 = ifnet.addFlowRelationTP("tIn", "p1");
IFNetFlowRelation f3 = ifnet.addFlowRelationPT("p1", "td");
IFNetFlowRelation f4 = ifnet.addFlowRelationTP("td", "p2");
IFNetFlowRelation f5 = ifnet.addFlowRelationPT("p2", "tOut");
IFNetFlowRelation f6 = ifnet.addFlowRelationTP("tOut", "pOut");
```

It is possible to add constraints to the flow relations to define the required and produced tokens and to set a maximum place capacity for specified token colors:

```
f2.addConstraint("red", 1);
```

```
f3.addConstraint("red", 1);
f4.addConstraint("green", 1);
f5.addConstraint("green", 1);
ifnet.getPlace("p2").setColorCapacity("green", 1);
```

Again an initial marking must be defined:

```
IFNetMarking sm = new IFNetMarking();
sm.set("pIn", new Multiset<String>("black"));
ifnet.setInitialMarking(sm);
```

The transitions can be assigned the access modes *create*, *write*, *read*, and *delete* for specified token colors. By default, they don't have any of them assigned. Access modes can be set like follows:

```
RegularIFNetTransition tIn = ifnet.getTransition("tIn");
tIn.addAccessMode("red", AccessMode.CREATE);
```

For the assignment of security clearances for subjects, objects, and activities an analysis context is needed. It consists of a labeling and an access control model, which again contains a `SOABase`. The SOA base contains all names of subjects, objects, and activities:

```
SOABase context = new SOABase("DataUsage");
context.setSubjects(Arrays.asList("sh1", "sh2", "sl0"));
context.setObjects(Arrays.asList("red", "green", "black"));
context.setActivities(Arrays.asList("tIn", "td", "tOut"));
```

An access control model defines which activities subjects are allowed to perform, based on the formerly defined SOABase. Different access control models can be found in the SEWOL project (see Library Dependencies in section 1.1).

```
ACLModel acl = new ACLModel("ACL", context);
acl.addActivityPermission("sh1", "tIn");
acl.addActivityPermission("sh2", "td");
acl.addActivityPermission("sl0", "tOut");
```

In the other direction, the analysis context assigns subjects to transitions.

```
AnalysisContext ac = new AnalysisContext("ac", acl, true);
// Assign subjects to transitions
ac.setSubjectDescriptor("tIn", "sh1");
ac.setSubjectDescriptor("td", "sh2");
ac.setSubjectDescriptor("tOut", "sl0");
```

Based on this analysis context a labeling can be defined, which assigns security levels to transitions, subjects, and token colors:

```
Labeling l = ac.getLabeling();

// Set subject clearance
l.setSubjectClearance("sh1", SecurityLevel.HIGH);
```

```
l.setSubjectClearance("sh2", SecurityLevel.HIGH);
l.setSubjectClearance("sl0", SecurityLevel.LOW);
// set transition classification
l.setActivityClassification("tIn", SecurityLevel.HIGH);
l.setActivityClassification("td", SecurityLevel.HIGH);
l.setActivityClassification("tOut", SecurityLevel.LOW);
// set token color classification
l.setAttributeClassification("red", SecurityLevel.HIGH);
l.setAttributeClassification("green", SecurityLevel.LOW);
```

In the end the analysis context is given to the IF-Net:

```
ifnet.setAnalysisContext(ac);
```

The dependencies between analysis context, labeling, access control model, and SOA base can be seen in figure 5.

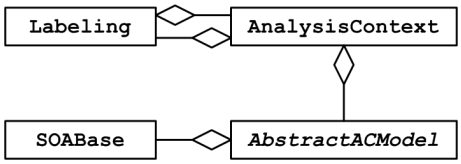

Figure 5: Relationships between analysis context, labeling, access control model, and SOA base.

For a valid IF-Net, the analysis context of an IF-Net must be valid, additionally to the conditions for validity of CPNs. This is the case, if the analysis context contains all token colors and activities of an IF-Net. Also a subject must be assigned to every activity and the security clearances of activities, objects, and subjects must be consistent. For a declassification transition producing a token color, no other transition in the Petri net is allowed to produce a token with the same color. A valid IF-Net must also meet some workflow conditions. At least one token of the control flow token color, by default black, must be consumed and produced by all transitions. The IF-Net must have the option to complete, i.e. the process can enter an end state. The end states must be proper, s.th. there are no remaining control flow tokens in the Petri net when entering the end state. Also there must be no dead transition, so every activity can be executed in at least one trace.

3 Petri Net Traversal

Enabled transitions should not always be fired manually. For this the SEPIA framework supports automated traversal of Petri net activities. Different from the traversal interface in the JAGAL framework, which is also implemented by all Petri nets, traversal of Petri net activities works by choosing an arbitrary enabled transition to fire. A transition to fire gets selecting randomly (`RandomPNTraverser`) or stochastically (`StochasticPNTraverser`). The random traverser fires an enabled transition randomly, where the stochastic traverser considers manually specified probabilities for the choice of enabled transitions to fire.

The following code randomly fires an enabled transition until no more transitions are available:

```
RandomPTTraverser t = new RandomPTTraverser(ptnet);
for (int i = 1; ptnet.hasEnabledTransitions(); i++) {
    System.out.println(i + ": " + ptnet.getEnabledTransitions().size());
    t.chooseNextTransition(ptnet.getEnabledTransitions()).fire();
}
System.out.println("no more enabled transitions");
```

An example for the usage of Petri net traversal can be found in the `PNTraversalUtils` class. It contains a method `testTraces`, which simulates a given Petri net a given number of times and returns the observed sequences of activities (traces). The following method call simulates the Petri net `ptnet` 10 times and prints out all distinct traces. Thereby it limits the number of activities per sequence to 100.

```
PNTraversalUtils.testTraces(ptnet, 10, 100, true, false, false);
```

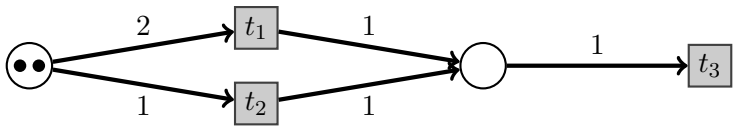

Figure 6: P/T-Net example for the Petri net traversal.

The output for ten simulations on the P/T-Net shown in figure 6 contains the following distinct traces:

```
[t1, t3]
[t2, t2, t3, t3]
[t2, t3, t2, t3]
```

4 Reachability

The marking graph of a Petri net describes all reachable states (markings) in terms of a transition system. Transitions of a transition system represent the firing of Petri net transitions, transition system events denote markings. Each sequence which reaches a final state (drain) represents a complete sequence of activities in the Petri net. Figure 7 shows the marking graph of the Petri net visualized in figure 6.

To build the marking graph of a Petri net the following method can be used:

```
PTMarkingGraph mg = MGConstruction.buildMarkingGraph(ptnet);
```

To get the corresponding sequences the sequence generator can be used:

```
SequenceGenerationCallableGenerator generator =
      new SequenceGenerationCallableGenerator(ptnet);
MGTraversalResult res = SequenceGeneration.getFiringSequences(generator);
```

The sequences which result in a final state can be retrieved by

```
res.getCompleteSequences();
```

The result is the same as in the traversal example in section 3:

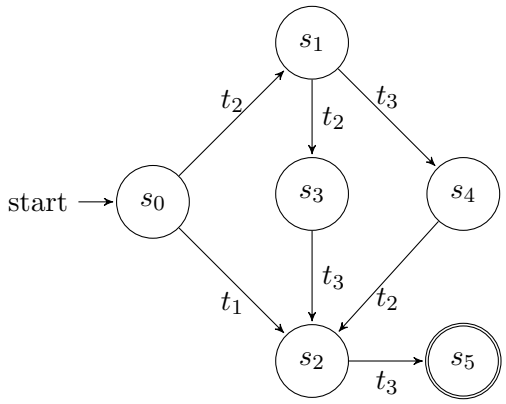

Figure 7: Marking graph of the P/T-Net example from the Petri net traversal in figure 6.

```
[t2, t2, t3, t3], [t2, t3, t2, t3], [t1, t3]
```

All possible sequences, including those not resulting in a final state, are returned by

```
res.getSequences();
```

The corresponding sequences are

```
[t2, t3], [t2, t2, t3, t3], [t2, t2], [t2], [t1], [t2, t3, t2, t3],
[t1, t3], [t2, t2, t3], [t2, t3, t2]
```

The marking graph is also used to check for the boundedness property on a Petri net. If the number of possible sequences is finite or smaller than the maximum possible value for the integer data type, the Petri net is bounded. Otherwise the Petri net is assumed to be unbounded.

5 Replaying

The replaying classes in the package `replay` check if logs are fitting on the specified Petri net. In doing so three different termination criteria are considered:

Possible firing sequence: The trace relates to a possible sequence in given Petri net. All activities can be fired according to their order within the trace.

No enabled transitions: The trace relates to a complete firing sequence in the given Petri net. All activities can be fired according to their order within the trace. After firing the last activity, there must not be any enabled transitions.

Escapable with silent transitions: The trace relates to a complete firing sequence in the given Petri net. All activities can be fired according to their order within the trace. After firing the last activity, only the firing of silent transitions is allowed to complete the sequence.

The following code creates a log for the formerly defined P/T-Net. The traces 1, 2, and 3 are complete sequences on the net, trace 6 is incomplete, and traces 4 and 5 are non-fitting traces for the given Petri net.

```
List<LogTrace<LogEntry>> log = new ArrayList<LogTrace<LogEntry>>();
log.add(LogTraceUtils.createTraceFromActivities(1, "t1","t3"));
log.add(LogTraceUtils.createTraceFromActivities(2, "t2","t3","t2","t3"));
log.add(LogTraceUtils.createTraceFromActivities(3, "t2","t2","t3","t3"));
log.add(LogTraceUtils.createTraceFromActivities(4, "t2","t1","t2","t3"));
log.add(LogTraceUtils.createTraceFromActivities(5, "t1","t2","t3"));
log.add(LogTraceUtils.createTraceFromActivities(6, "t2","t2","t3"));
```

The class `ReplayCallableGenerator` contains static methods to replay traces on a Petri net. The following example replays the log on the P/T-Net with the termination criterion *possible firing sequence*:

```
ReplayCallableGenerator gen = new ReplayCallableGenerator(ptnet);
gen.setLogTraces(log);
gen.setTerminationCriteria(TerminationCriteria.POSSIBLE_FIRING_SEQUENCE);
ReplayResult<LogEntry> result = Replay.replayTraces(gen);
```

The corresponding output is the following, where the percentage of fitting and non-fitting traces is shown:

```
Replaying log on model "PetriNet"... done [fitting=0.6666666666666666,
not fitting=0.3333333333333333] [9 milliseconds]
```

Which traces were fitting and which traces were non-fitting can be retrieved from the object `result` using the following methods:

```
System.out.println( result.getFittingTraces() );
System.out.println( result.getNonFittingTraces() );
```

6 Petri Net Graphics

The Petri net classes as they are don't contain any visualization information like object positioning, object dimension, or border style. To keep logic and visualization separated, the graphical information is available in graphical container classes. Figure 8 visualizes the inheritance hierarchy of the graphical Petri net classes with the corresponding Petri net classes. Note that `AbstractGraphicalCPN` and `AbstractCPNGraphics` are direct subclasses of `AbstractGraphicalPN` and `AbstractPNGraphics` instead of inheriting from the P/T-Net classes. The reason for this is the difference in the marking and the flow relation constraints, which are represented as strings in the CPN and the IF-Net.

The graphical attributes in the graphical classes are motivated by the styling possibilities given by the PNML standard[1]. Basically we differentiate between graphical information of nodes (places and transitions), arcs (flow relations), and annotations (labels). The corresponding attributes are listed in the classes `NodeGraphics`, `ArcGraphics`, and `AnnotationGraphics`.

The graphic objects are assigned to elements of the Petri net using an associative array with the name of the node, arc or annotation as key. In the following example we set the text alignment for all arc annotations to *right*. At first we create the graphic objects and assign colors to the color names. The token clearances and the subject clearances also need a positioning:

[1] see `http://www.pnml.org/version-2009/version-2009.php`

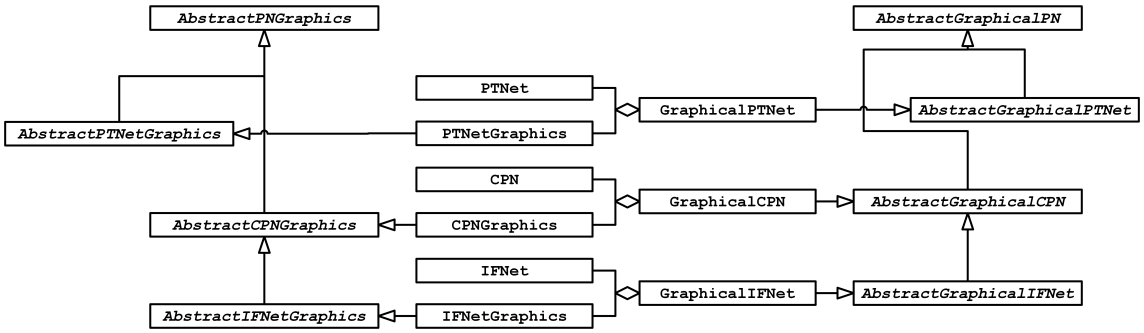

Figure 8: Inheritance hierarchy of the graphical Petri net classes with the aggregated Petri net classes.

```
IFNetGraphics ifNetG = new IFNetGraphics();
GraphicalIFNet gIFNet = new GraphicalIFNet(ifnet, ifNetG);

Map<String, Color> colors = new HashMap<String, Color>();
colors.put("black", Color.BLACK);
colors.put("red", Color.RED);
colors.put("green", Color.GREEN);
ifNetG.setColors(colors);

ifNetG.setClearancesPosition(new Position(20, 30));
ifNetG.setTokenLabelsPosition(new Position(40, 30));
```

Now the text alignment can be set:

```
Font annotationFont = new Font();
annotationFont.setAlign(Align.RIGHT);
AnnotationGraphics arcAnnotation = new AnnotationGraphics();
arcAnnotation.setFont(annotationFont);

ifNetG.getArcAnnotationGraphics().put(f1.getName(), arcAnnotation);
ifNetG.getArcAnnotationGraphics().put(f2.getName(), arcAnnotation);
ifNetG.getArcAnnotationGraphics().put(f3.getName(), arcAnnotation);
ifNetG.getArcAnnotationGraphics().put(f4.getName(), arcAnnotation);
ifNetG.getArcAnnotationGraphics().put(f5.getName(), arcAnnotation);
ifNetG.getArcAnnotationGraphics().put(f6.getName(), arcAnnotation);
```

The object `ifNetG` now contains the associations between Petri net components and graphical information. The object's `toString()`-method returns the following string:

```
            placeGraphics# 0
       transitionGraphics# 0
              arcGraphics# 0
            tokenGraphics# 0
    arcAnnotationGraphics# 6:
                              arcTP_tOutpOut: [Font( align:right )]
                              arcTP_tInp1: [Font( align:right )]
```

```
                                    arcTP_tdp2: [Font( align:right )]
                                    arcPT_p1td: [Font( align:right )]
                                    arcPT_pIntIn: [Font( align:right )]
                                    arcPT_p2tOut: [Font( align:right )]
     placeLabelAnnotationGraphics# 0
transitionLabelAnnotationGraphics# 0
                       tokenColors# 3:
                                    red: java.awt.Color[r=255,g=0,b=0]
                                    green: java.awt.Color[r=0,g=255,b=0]
                                    black: java.awt.Color[r=0,g=0,b=0]
           accessFunctionGraphics# 0
                  subjectGraphics# 0
              clearancesPosition#:  Position( 20.0 / 30.0 )
             tokenLabelsPosition#:  Position( 40.0 / 30.0 )
```

7 Serializing & Parsing of Petri Nets

Serializing Petri Nets

To store Petri nets and make them available for other users and tools, the SEPIA framework comes with a serialization functionality. It supports the file formats PNML and Petrify. To launch the serialization process, an instance of a graphical Petri net, the declaration of the output file format, and the path to the target file are needed.

```
PNSerialization.serialize(gIFNet, PNSerializationFormat.PNML,
                          "/arbitrary/path/ifnet.pnml");
```

Parsing Petri Nets

Petri nets can also be parsed using SEPIA's built-in parser. It supports the PNML and the Petrify file formats and recognizes the file type by the file extension. After parsing it returns an instance of `AbstractGraphicalPN`.

```
AbstractGraphicalPN gNet = PNParsing.parse(
                               new File("/arbitrary/path/ifnet.pnml"));
```

To determine the type of the parsed Petri net, the class type must be checked top-down.

```
if (gNet instanceof GraphicalIFNet)
    System.out.println("IF-Net");
else if (gNet instanceof GraphicalCPN)
    System.out.println("CPN");
else if (gNet instanceof GraphicalPTNet)
    System.out.println("P/T-Net");
else
    System.out.println("unknown net type");
```

If the input file type is already known an appropriate parser can be called manually. Thus a user can specify if a Petri net should be validated against its net type definition or if an exception should be thrown if the net type is not known.

```
PNMLParser p = new PNMLParser();
AbstractGraphicalPN gNet = p.parse(
                              new File("/arbitrary/path/ifnet.pnml"),
                              true,  // require net type
                              true   // validate
                            );
```