The Property Language Manual

Model Checking Contest @ Petri Nets 2015

Contact:

The MCC formula team,
César Rodríguez, Loïg Jezequel, Emmanuel Paviot-Adet

v. Feb. 11, 2015

Contents

1 Introduction

1.1 Scope

This document is intended as a guide for the developers of the tools participating in the Model Checking Contest @ Petri Nets, edition 2015. It specifically contains:

- The definition of the property language used in the contest this year.
- The definition of the language subset presented in each category of the contest (new!).
- Guidance on how to parse the XML files containing the properties.

1.2 Categories of the contest

The Model Checking Contest @ Petri Nets compares the competing verification tools on four categories of verification goals:

1. State Space generation
2. Reachability analysis
3. LTL analysis
4. CTL analysis

To maximize tool participation, we have further divided each category into several subcategories, where only restricted kinds of formulas will be found. Overall, we propose 12 subcategories:

Subcategory	Description
ReachabilityDeadlock	Existence of deadlock states
ReachabilityFireabilitySimple	Firability of transitions
ReachabilityFireability	Boolean combinations of propositions checking firability of transitions
ReachabilityCardinality	Boolean combinations of propositions comparing the number of tokens in places
ReachabilityBounds	Boolean combinations of propositions comparing the bounds of places
ReachabilityComputeBounds	Computing the bounds of places
LTLFireabilitySimple	LTL properties with only one linear-time operator and no boolean operator
LTLFireability	Full LTL with atomic propositions checking firability of transitions
LTLCardinality	Full LTL with atomic propositions comparing the number of tokens in places
CTLFireabilitySimple	CTL properties with only one temporal operator and no boolean operator
CTLFireability	Full CTL with atomic propositions checking firability of transitions
CTLCardinality	Full CTL with atomic propositions comparing the number of tokens in places

Each subcategory designates the name of a XML file providing, for every benchmark, a set of formulæ for tools competing in this subcategory to solve.

The 12 subcategories are this year the same as the ones proposed the last year. However, the class of formulas (i.e., the specific fragment of the BNF grammar presented in Section 2) contained in each of them has varied from last year.

Most of the formulas presented to tools have been randomly generated. Only very few of them were written by hand. Unfortunately, most of the benchmarks submitted to the contest came without example verification properties. Writing them by hand is a very time consuming task (there are currently more than 250 models!). This is a very unfortunate situation but we have not found any way to overcome it.

1.3 Goals and evolution of the property language

The property language for the Model Checking Contest @ Petri Nets is designed to maximize the tools participating in the competition. It is based on XML and allows to write structural, reachability, CTL, LTL formulæ. It is tightly related to Petri nets, that are the modeling formalism of the MCC.

This language is designed to evolve in the future editions of the contest. It is also designed to integrate with the *Petri Net Markup Language* (PNML) in the future.

The property language has been extensively simplified with respect to the previous edition of the MCC. We kept compatibility in mind. Any tool that participated in the previous edition of the contests should only require minimal changes to accept the new version of the language.

1.4 Outline

This document presents the specification language in, conceptually, two steps.

- We first use a BNF grammar to describe the overall specification language. This grammar is also used to formally define the class of formulas that will be presented to the tools participating on each subcategory of the contest (Section 2).
- Second, the document defines the XML syntax of the files that will actually be passed to the tools (Section 3).

Section 4 provides some help on how to automatically generate code in various languages to parse the XML files containing the formulas.

2 The BNF grammar of the property language

Every property presented to a tool participating in any subcategory of the contests can be derived from the following general grammar of the property language:

⟨*formula*⟩	::=	⟨*boolean-formula*⟩
	|	⟨*integer-formula*⟩
⟨*boolean-formula*⟩	::=	**A** ⟨*boolean-formula*⟩
	|	**E** ⟨*boolean-formula*⟩
	|	**G** ⟨*boolean-formula*⟩
	|	**F** ⟨*boolean-formula*⟩
	|	**X** ⟨*boolean-formula*⟩
	|	⟨*boolean-formula*⟩ **U** ⟨*boolean-formula*⟩
	|	$\neg$ ⟨*boolean-formula*⟩
	|	⟨*boolean-formula*⟩ $\wedge$ ⟨*boolean-formula*⟩
	|	⟨*boolean-formula*⟩ $\vee$ ⟨*boolean-formula*⟩
	|	⟨*atom*⟩
⟨*atom*⟩	::=	**deadlock**
	|	**is-fireable**$(t_1, \ldots, t_n)$
	|	⟨*integer-expression*⟩ $\leq$ ⟨*integer-expression*⟩
⟨*integer-formula*⟩	::=	⟨*integer-expression*⟩
⟨*integer-expression*⟩	::=	⟨*integer-expression*⟩ $+$ ⟨*integer-expression*⟩
	|	⟨*integer-expression*⟩ $-$ ⟨*integer-expression*⟩
	|	*Integer constant*
	|	**place-bound**$(p_1, \ldots, p_n)$
	|	**token-count**$(p_1, \ldots, p_n)$

The property language considers two kinds of formulæ: boolean formulæ, evaluating to a boolean value, and integer formulæ, which evaluate to a natural number. Only the subcategory *ReachabilityComputeBounds* employs integer formulæ, all other subcategories use only boolean formulæ.

Operators **A** and **E** are the standard CTL operators. Operators **G**, **F**, **X**, **U** are the standard LTL operators. We define that the **X** operator evaluate to false if no successor state exists. A boolean formula can also contain boolean combinations of boolean formulas. Atomic propositions are either

- **deadlock**, asking that the current state enables no transition,
- **is-fireable**$(t_1, \ldots, t_n)$, which holds iff either t_1, or t_2, or ... or t_n are enabled at the current state, or
- inequalities of the form ⟨*expr*⟩ $\leq$ ⟨*expr*⟩, where ⟨*expr*⟩ is an integer expression.

Integer expressions can contain additions and subtractions of *Integer constants*, always taken from the set $\{1, 2, 3\}$, and the following operators:

- **place-bound**$(p_1, \ldots, p_n)$, which returns the exact or estimated maximum number of tokens that any reachable marking can put in all n places at the same time.
- **token-count**$(p_1, \ldots, p_n)$, which returns, for the current marking, the exact number of tokens contained in the set $\{p_1, \ldots, p_n\}$ of places.

Note that, in fact, in the current edition of the MCC no subcategory of the contest will contain formulas using additions and subtractions of integer expressions. These operators remain here only for future use.

This grammar defines a large class of formulæ. Each subcategory of the contest contains formulas of only some specific fragment of this grammar. The following subsections define the specific syntax of the formulæ that tools participating in a given subcategory shall expect to be provided with.

Section 3 presents the correspondence between the above BNF grammar and the syntax of the XML files that will actually be passed to the tools.

2.1 Subcategory *ReachabilityDeadlock*

Tools participating in this subcategory will only find one formula to solve for every benchmark:

⟨*formula*⟩ ::= ⟨*boolean-formula*⟩

⟨*boolean-formula*⟩ ::= **E F deadlock**

2.2 Subcategory *ReachabilityFireabilitySimple*

Tools participating in this subcategory will only find two kinds of formulæ to solve for every benchmark:

⟨*formula*⟩ ::= ⟨*boolean-formula*⟩

⟨*boolean-formula*⟩ ::= **E F is-fireable**$(t_1, \ldots, t_n)$
| **A G is-fireable**$(t_1, \ldots, t_n)$

2.3 Subcategories *ReachabilityFireability* and *ReachabilityCardinality*

Both subcategories contain formulas produced using the same grammar, the only difference being the atomic propositions allowed in each one. The grammar is the following:

⟨*formula*⟩ ::= ⟨*boolean-formula*⟩

⟨*boolean-formula*⟩ ::= **E F** ⟨*state-formula*⟩
| **A G** ⟨*state-formula*⟩

⟨*state-formula*⟩ ::= $\neg$ ⟨*state-formula*⟩
| ⟨*state-formula*⟩ $\wedge$ ⟨*state-formula*⟩
| ⟨*state-formula*⟩ $\vee$ ⟨*state-formula*⟩
| ⟨*atom*⟩

Now, the subcateory *ReachabilityFireability* only contains **is-fireable**$(\cdot)$ atoms:

⟨*atom*⟩ ::= **is-fireable**$(t_1, \ldots, t_n)$

while the formulas in the *ReachabilityCardinality* subcategory uniquely contain integer inequalities:

⟨*atom*⟩ ::= ⟨*integer-expression*⟩ $\leq$ ⟨*integer-expression*⟩

⟨*integer-expression*⟩ ::= *Integer constant*
| **token-count**$(\boldsymbol{p_1, \ldots, p_n})$

Recall that the *Integer constant* will be a number in $\{1, 2, 3\}$.

2.4 Subcategory *ReachabilityBounds*

⟨*formula*⟩ ::= ⟨*boolean-formula*⟩

⟨*boolean-formula*⟩ ::= $\neg$ ⟨*boolean-formula*⟩
| ⟨*boolean-formula*⟩ $\wedge$ ⟨*boolean-formula*⟩
| ⟨*boolean-formula*⟩ $\vee$ ⟨*boolean-formula*⟩
| ⟨*atom*⟩

⟨*atom*⟩ ::= ⟨*integer-expression*⟩ $\leq$ ⟨*integer-expression*⟩

⟨*integer-expression*⟩ ::= *Integer constant*
| **place-bound**$(\boldsymbol{p_1, \ldots, p_n})$

2.5 Subcategory *ReachabilityComputeBounds*

Observe that this is the only subcategory where formulas evaluate to an integer number rather than a boolean value.

⟨*formula*⟩ ::= ⟨*integer-formula*⟩

⟨*integer-formula*⟩ ::= **place-bound**$(p_1, \ldots, p_n)$

2.6 Subcategory *LTLFireabilitySimple*

⟨*formula*⟩ ::= ⟨*boolean-formula*⟩

⟨*boolean-formula*⟩ ::= **A** ⟨*path-formula*⟩

⟨*path-formula*⟩ ::= **G** ⟨*atom*⟩
| **F** ⟨*atom*⟩
| **X** ⟨*atom*⟩
| ⟨*atom*⟩ **U** ⟨*atom*⟩

⟨*atom*⟩ ::= **is-fireable**$(t_1, \ldots, t_n)$

2.7 Subcategories *LTLFireability* and *LTLCardinality*

Both subcategories contain formulas produced using the same grammar, the only difference being the atomic propositions allowed in each one. The grammar is the following:

⟨*formula*⟩ ::= ⟨*boolean-formula*⟩

⟨*boolean-formula*⟩ ::= **A** ⟨*path-formula*⟩

⟨*path-formula*⟩ ::= **G** ⟨*path-formula*⟩
| **F** ⟨*path-formula*⟩
| **X** ⟨*path-formula*⟩
| ⟨*path-formula*⟩ **U** ⟨*path-formula*⟩
| ⟨*atom*⟩

Now, the subcategory *LTLFireability* only contains **is-fireable**($\cdot$) atoms:

⟨*atom*⟩ ::= **is-fireable**($t_1, \ldots, t_n$)

while the formulas in the *LTLCardinality* subcategory only contain integer inequalities:

⟨*atom*⟩ ::= ⟨*integer-expression*⟩ $\leq$ ⟨*integer-expression*⟩

⟨*integer-expression*⟩ ::= *Integer constant*
| **token-count**($\boldsymbol{p_1, \ldots, p_n}$)

2.8 Subcategory *CTLFireabilitySimple*

⟨*formula*⟩ ::= ⟨*boolean-formula*⟩

⟨*boolean-formula*⟩ ::= **A** ⟨*path-formula*⟩
| **E** ⟨*path-formula*⟩

⟨*path-formula*⟩ ::= **G** ⟨*atom*⟩
| **F** ⟨*atom*⟩
| **X** ⟨*atom*⟩
| ⟨*atom*⟩ **U** ⟨*atom*⟩

⟨*atom*⟩ ::= **is-fireable**($t_1, \ldots, t_n$)

2.9 Subcategories *CTLFireability* and *CTLCardinality*

Both subcategories contain formulas produced using the same grammar, the only difference being the atomic propositions allowed in each one. The grammar is the following:

⟨*formula*⟩ ::= ⟨*boolean-formula*⟩

⟨*boolean-formula*⟩ ::= **A** ⟨*path-formula*⟩
| **E** ⟨*path-formula*⟩
| $\neg$ ⟨*boolean-formula*⟩
| ⟨*boolean-formula*⟩ $\wedge$ ⟨*boolean-formula*⟩
| ⟨*boolean-formula*⟩ $\vee$ ⟨*boolean-formula*⟩
| ⟨*atom*⟩

⟨*path-formula*⟩ ::= **G** ⟨*boolean-formula*⟩
| **F** ⟨*boolean-formula*⟩
| **X** ⟨*boolean-formula*⟩
| ⟨*boolean-formula*⟩ **U** ⟨*boolean-formula*⟩

The subcategory *CTLFireability* only contains **is-fireable**$(\cdot)$ atoms:

$\langle$*atom*$\rangle$ ::= **is-fireable**$(t_1, \ldots, t_n)$

while the formulas in the *CTLCardinality* subcategory contain integer inequalities:

$\langle$*atom*$\rangle$ ::= $\langle$*integer-expression*$\rangle \leq \langle$*integer-expression*$\rangle$

$\langle$*integer-expression*$\rangle$ ::= *Integer constant*
| **token-count**$(p_1, \ldots, p_n)$

3 XML format of the property language

During the competition, tools will be provided, for every benchmark and every subcategory of the contest, with one XML file containing the set of formulas to solve for that subcategory and that benchmark.

The purpose of this section is defining the XML Schema describing the structure of the XML files containing the formulas. In fact, we describe a RelaxNG representation of the XML Schema.

Each XML file will be provided together with a plain text file describing the formula in a more readable way. The format of this text file is not guaranteed to be preserved in future editions of the contest. This file is just provided for readability.

As an introducing example, consider the following property, given in XML (left) and the corresponding to a set of formulas containing only one formula (right) generated by the grammar in Section 2.

```
<?xml version="1.0"?>
<property–set xmlns="http://mcc.lip6.fr/">
 <property>
  <id>Dekker–PT–010–test1</id>
  <description>
   Automatically generated formula.
  </description>
  <formula>
   <exists–path>
    <until>
     <before>
      <negation>
       <integer–le>
        <integer–constant>1</integer–constant>
        <tokens–count>
         <place>p3_0</place>
        </tokens–count>
       </integer–le>
      </negation>
     </before>
     <reach>
      <is–fireable>
       <transition>withdraw_4_8</transition>
      </is–fireable>
     </reach>
    </until>
   </exists–path>
  </formula>
 </property>
</property–set>
```

E $((\neg\ (1 \leq$ **token-count**(p3_0))) U **is-fireable**(withdraw_4_8))

What follows is the description of the RelaxNG grammar describing the XML files provided to the tools.

3.1 Property sets

The `property–set` element is the root of the XML representation. It contains one or more properties.

```
default namespace = "http://mcc.lip6.fr/"
start = property–set

property–set = element property–set {
  property*
}
```

3.2 Properties

A property is composed of three mandatory parts: a unique identifier, a textual description of the property, and the formula itself.

```
property = element property {
  element id {
    xsd:ID
  } &
  element description {
    text
  } &
  element formula {
    formula
  }
}
```

3.3 Formulæ

Formulæ are the body of properties. They define what is expected to hold on the model. Formulæ are currently of two main types: formulæ that return integers, and formulæ that return Booleans.

In the following, for each rule of the grammar defining formulæ we give the RelaxNG representation (on the left) as well as the corresponding part of the BNF grammar (on the right).

```
formula =
    boolean–formula
  | integer–formula
```

$$
\begin{array}{lll}
\langle \textit{formula} \rangle & ::= & \langle \textit{boolean-formula} \rangle \\
 & | & \langle \textit{integer-formula} \rangle
\end{array}
$$

3.3.1 CTL state operators

These operators correspond to the **A** and **E** operators of CTL.

```
boolean–formula =
    ...
  | element all–paths {
      boolean–formula
    }
  | element exists–path {
      boolean–formula
    }
  | ...
```

$$
\begin{array}{lll}
\langle \textit{boolean-formula} \rangle & ::= & \mathbf{A}\ \langle \textit{boolean-formula} \rangle \\
 & | & \mathbf{E}\ \langle \textit{boolean-formula} \rangle \\
 & | & \ldots
\end{array}
$$

3.3.2 Path operators

These operators are the **G**, **F**, **X**, and **U** operators of CTL and LTL Recall that the `next` operator should evaluate to false if no successor state exists.

```
boolean–formula =
    ...
  | element globally {
      boolean–formula
    }
  | element finally {
      boolean–formula
    }
  | element next {
      boolean–formula &
  | element until {
      element before {
        boolean–formula
      } &
      element reach {
        boolean–formula
      } &
    }
  | ...
```

$$
\begin{array}{lll}
\langle \textit{boolean-formula} \rangle & ::= & \mathbf{G}\ \langle \textit{boolean-formula} \rangle \\
 & | & \mathbf{F}\ \langle \textit{boolean-formula} \rangle \\
 & | & \mathbf{X}\ \langle \textit{boolean-formula} \rangle \\
 & | & \langle \textit{boolean-formula} \rangle\ \mathbf{U}\ \langle \textit{boolean-formula} \rangle \\
 & | & \ldots
\end{array}
$$

3.3.3 Boolean operators

These are usual Boolean operators.

```
boolean–formula =
    ...
  | element negation {
      boolean–formula
    }
  | element conjunction {
      boolean–formula,
      boolean–formula+
    }
  | element disjunction {
      boolean–formula,
      boolean–formula+
    }
  | ...
```

$$\begin{array}{lcl} \langle\textit{boolean-formula}\rangle & ::= & \neg\ \langle\textit{boolean-formula}\rangle \\ & | & \langle\textit{boolean-formula}\rangle \wedge \langle\textit{boolean-formula}\rangle \\ & | & \langle\textit{boolean-formula}\rangle \vee \langle\textit{boolean-formula}\rangle \\ & | & \ldots \end{array}$$

3.3.4 Atomic propositions

There is three types of atomic propositions. Recall that `deadlock` evaluates to true if the current state is a deadlock (has no successor) and `is–fireable` evaluates to true if one of the set of transitions given is fireable from the current state.

```
boolean–formula =
    ...
  | element deadlock { empty }
  | element is–fireable {
      transition+
    }
  | element integer–le {
      integer–expression,
      integer–expression
    }
  | ...
```

$$\langle\textit{boolean-formula}\rangle ::= \langle\textit{atom}\rangle \mid \ldots$$

$$\begin{array}{lcl} \langle\textit{atom}\rangle & ::= & \textbf{deadlock} \\ & | & \textbf{is-fireable}(t_1, \ldots, t_n) \\ & | & \langle\textit{integer-expression}\rangle \leq \langle\textit{integer-expression}\rangle \end{array}$$

3.4 Integer formulæ

An integer formula is an integer expression. The tool must return the integer, that is the result of the expression.

```
integer–formula =
    integer–expression

integer–expression = ...
```

$$\langle\textit{integer-formula}\rangle ::= \langle\textit{integer-expression}\rangle$$

3.4.1 Arithmetic operators

These are usual arithmetic operators for integers.

```
integer–expression =
    ...
| element integer–constant {
    xsd:integer
  }
| element integer–sum {
    integer–expression,
    integer–expression+
  }
| element integer–difference {
    integer–expression,
    integer–expression
  }
| ...
```

$$\begin{array}{lcl} \langle\textit{integer-expression}\rangle & ::= & \textit{Integer constant} \\ & | & \langle\textit{integer-expression}\rangle + \langle\textit{integer-expression}\rangle \\ & | & \langle\textit{integer-expression}\rangle - \langle\textit{integer-expression}\rangle \\ & | & \ldots \end{array}$$

3.4.2 Token-counting integer operators

```
integer−expression =
    ...
  | element place−bound {
      place+
    }
  | element tokens−count {
      place+
    | ...
```

$$\langle \textit{integer-expression} \rangle ::= \textbf{place-bound}(p_1, \ldots, p_n) \mid \textbf{token-count}(p_1, \ldots, p_n) \mid \ldots$$

Recall that

- place−bound returns the exact of estimated bound of a set of places; for several places, it means the maximum number of tokens in all these places at the same time;
- tokens−count returns the exact number of tokens in a set of places.

3.5 Places and transitions

Places and transitions are uniquely identified. The identifiers occurring at the formula XML files are the id's present in the PNML model.

```
place =
    element place {
      xsd:IDREF
    }

transition =
    element transition {
      xsd:IDREF
    }
```

4 Parsing the XML formula files

The RelaxNG grammar can be downloaded from http://mcc.lip6.fr/properties/mcc-properties.rnc using:

```
wget http://mcc.lip6.fr/properties/mcc−properties.rnc
```

The XML schema file can be downloaded from http://mcc.lip6.fr/properties/mcc-properties.xsd using:

```
wget http://mcc.lip6.fr/properties/mcc−properties.xsd
```

4.1 How to generate the XML schema from the RelaxNG grammar?

The Trang tool is able to transform the RelaxNG grammar into an XML Schema. Visit http://www.thaiopensource.com/relaxng/trang.html to install this tool.

```
trang −I rnc −O xsd mcc−properties.rnc mcc−properties.xsd
```

4.2 How to generate C++ classes from the XML Schema?

Generation of C++ classes requires Code Synthesis' xsd tool (http://www.codesynthesis.com/products/xsd/). This tool converts the XML Schema of the property language to a set of C++ classes, an XML validating parser, and an XML output. This tool is free software, and is available for numerous platforms. It is available at http://www.codesynthesis.com/products/xsd/download.xhtml. Parsing and validating XML also requires to install Xerces-C++, available at http://xerces.apache.org/xerces-c/.
After installing the xsd tool, you have to fix the file xsd/cxx/zc−istream.txx:

```
35c35
<       setg (b, b, e);
−−−
>       this−>setg (b, b, e);
```

The conversion from the XML Schema to C++ classes is then performed using the following command:

```
mkdir -p src/cxx/
xsd cxx-tree \
    --generate-serialization \
    --generate-doxygen \
    --generate-ostream \
    --generate-comparison \
    --generate-detach \
    --generate-default-ctor \
    --generate-polymorphic --polymorphic-type-all \
    --namespace-map http://mcc.lip6.fr=mcc \
    --output-dir src/cxx/ \
    --root-element property-set \
    mcc-properties.xsd
```

4.3 How to generate Java classes from the XML Schema?

Conversion from the XML Schema to Java classes requires the Java Architecture for XML Binding (JAXB – http://jaxb.java.net/). It is included in recent Java distributions.

To generate the classes, use the following command:

```
mkdir -p src/java/
xjc -d src/java/ -p mcc mcc-properties.xsd
```

It generates a set of Java files in the `java` directory.

4.4 How to generate Python classes from the XML Schema?

The python script generateDS (http://www.rexx.com/~dkuhlman/generateDS.html) generates Python code from the XML Schema.

```
mkdir -p src/python/
python generateDS.py -m -f --silence -o src/python/mcc-properties.py mcc-properties.xsd
```

4.5 How to generate C# classes from the XML Schema?

There seems to be also tools for C# developers:
http://stackoverflow.com/questions/386155/comparison-of-xsd-codegenerators-c. We did not test them, but are interested by feedback if you use one.