Goals of the property language

The property language for the MCC'2012 @ Petri Nets has been designed to allow participation of many tools. It allows to write reachability formulæ, CTL formulæ, LTL formulæ and also to express some structural properties. It is tightly related to Petri nets, that are the modeling formalism used during the contest.

This language is designed to evolve in the future editions of the contest. It is also designed to integrate with Petri Net Markup Language in the future. To do so, the language is provided both as a BNF grammar and as an XML Schema.

Contents

1 Description of the langage

The property language is available in two flavors: XML and textual. They are not equivalent, as the XML flavor is richer than the textual one, and proposes some possible evolutions of the language for the next MCC'2012.

We give in this document the EBNF[1] grammar (that can be parsed using a recursive-descent parser) of the textual flavor, together with the equivalent XML schema.

We also provide an implementation in C++ of the parser, that converts the textual flavor to its XML equivalent. APIs to read the XML flavor can be generated, at least for C++, Java and Python.

1.1 Identifiers

An identifier is any nonempty sequence of characters, starting with a letter, a number or underscores. Depending on the context, it should not start with reserved keywords:

```
I, N
A, E
X, U, W, F, G
bound, deadlock,
true, false, xor,
l0, l1, l2, l3, l4
```

The parser provided for the Model Checking Contest can handle in some cases such identifiers.

1.2 Property sets

The properties are given, in the textual flavor, as a file containing several named properties, defined as a (nonempty) sequence of properties.

```
property_set: property+
```

The `property_set` element is the root of the XML representation. It contains several (at least 1) properties.

```
<xs:element name="property_set">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="unbounded" ref="mcc:property"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Note that we do not allow two properties to have the same name. The contents of a `property_set` can thus be seen as a sequence or as a set of properties.

[1] Note that we use a space character instead of "," for concatenation.

1.3 Properties

A property associates a name to a formula. The name is unique, and is used in the answer to identify the property. Each property contains some tags. Currently, only two are available: `ctl` identifies a CTL-only formula, whereas `ltl` identifies a LTL-only formula. Other formulas, such as reachability or structural ones, do not have tags, because they can be computed using several techniques, possibly CTL or LTL.

In the textual flavor, a property is composed of a prefix: nothing, `ltl` or `ltl`, followed by the property name and its body.

```
property: [ "struct" | "reach" | "ctl" | "ltl" ] identifier ":" formula
```

The XML flavor defines a property as having a name, a sequence of tags, and a formula as body.

```
<xs:element name="property">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="id" type="xs:ID"/>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="mcc:property-tag"/>
      </xs:sequence>
      <xs:element ref="mcc:formula"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Note that the XML flavor stores a sequence of `property-tag`, whereas the textual flavor only allows to pick one of the two tags. The same constraint should be put in the XML flavor.

1.4 Tags

We define tags using a hierarchy. The same approach is used for many elements in the language (for instance formulas), as it allows automatic generation of APIs in several languages. The hierarchy is obtained by defining an abstract base type and element (`property-tag`), and then defining types that inherit from the base type (`structural-only`, `reachability-only`, `ctl-only` and `ltl-only`), together with elements that can be substituted to the base element.

```
<xs:complexType name="property-tag" abstract="true"/>
<xs:element     name="property-tag" abstract="true" type="mcc:property-tag"/>

<xs:complexType name="structural-only">
  <xs:complexContent>
    <xs:extension base="mcc:property-tag"/>
  </xs:complexContent>
</xs:complexType>
<xs:element name="structural-only" substitutionGroup="mcc:property-tag" type="mcc:structural-only"/>

<xs:complexType name="reachability-only">
  <xs:complexContent>
    <xs:extension base="mcc:property-tag"/>
  </xs:complexContent>
</xs:complexType>
<xs:element name="reachability-only" substitutionGroup="mcc:property-tag" type="mcc:reachability-only"/>

<xs:complexType name="ctl-only">
  <xs:complexContent>
    <xs:extension base="mcc:property-tag"/>
  </xs:complexContent>
</xs:complexType>
<xs:element name="ctl-only" substitutionGroup="mcc:property-tag" type="mcc:ctl-only"/>

<xs:complexType name="ltl-only">
  <xs:complexContent>
    <xs:extension base="mcc:property-tag"/>
  </xs:complexContent>
</xs:complexType>
<xs:element name="ltl-only" substitutionGroup="mcc:property-tag" type="mcc:ltl-only"/>
```

1.5 Data Types

The properties use data of several different types. They do not appear explicitly in the textual flavor, but are defined as base types in the XML flavor.

- Colors in markings are *currently* handled as tuples, containing values, that refer to the names defined in the PNML file. Change the type

```
<xs:complexType name="value" abstract="true"/>
<xs:element name="value" abstract="true" type="mcc:value"/>
```

- Integers cover all expressions that return integers. They currently represent constant, cardinalities of tokens, and arithmetic operations between them.

```
<xs:complexType name="integer" abstract="true"/>
<xs:element name="integer" abstract="true" type="mcc:integer"/>
```

- Multisets cover all expressions that return multisets. They currently represent place markings and multiset operations between them.

```
<xs:complexType name="multiset" abstract="true"/>
<xs:element name="multiset" abstract="true" type="mcc:multiset"/>
```

- Places are not used directly, expect for restriction of a particular place to some colors.

```
<xs:complexType name="place" abstract="true"/>
<xs:element name="place" abstract="true" type="mcc:place"/>
```

- Transitions are not used directly, expect for restriction of a particular transitions to some bindings.

```
<xs:complexType name="transition" abstract="true"/>
<xs:element name="transition" abstract="true" type="mcc:transition"/>
```

1.6 Formulæ

Formulæ are the body of properties. They define what is expected to hold on the model.

For the textual flavor, we define formulæ using choices, where the order of branches is chosen to allow a compact and readable syntax. We distinguish `complex_formula` and `simple_formula`: the simple formulæ can be combined without the use of parenthesis, contrary to complex formulæ.

```
simple_formula : simple_composite_formula
               | system_formula
               | state_formula
               | simple_path_formula
               | petri_proposition
               | integer_comparison
               | multiset_comparison
               | "(" formula ")"
complex_formula: complex_composite_formula
               | complex_path_formula
formula: complex_formula
       | simple_formula
```

The XML flavor defines a hierarchy of formulæ. So, the `formula` element is abstract. We define its subclasses in the remaining of this document.

```
<xs:complexType name="formula" abstract="true"/>
<xs:element     name="formula" abstract="true" type="mcc:formula"/>
```

The hierarchy is structured around several intermediate types:

- `multiset.comparison` are comparisons over multisets (place markings):

```
<xs:complexType name="multiset.comparison" abstract="true">
  <xs:complexContent>
    <xs:extension base="mcc:formula"/>
  </xs:complexContent>
</xs:complexType>
<xs:element name="multiset.comparison" substitutionGroup="mcc:formula" abstract="true" type="mcc:multiset.comparison"/>
```

- `integer.comparison` are comparisons over integers (cardinalities):

```
<xs:complexType name="integer.comparison" abstract="true">
  <xs:complexContent>
    <xs:extension base="mcc:formula"/>
  </xs:complexContent>
</xs:complexType>
<xs:element name="integer.comparison" substitutionGroup="mcc:formula" abstract="true" type="mcc:integer.comparison"/>
```

- `formula.system` are system-wide formulæ operators, defining invariants:

```
 <xs:complexType name="formula.system" abstract="true">
  <xs:complexContent>
    <xs:extension base="mcc:formula"/>
  </xs:complexContent>
</xs:complexType>
<xs:element name="formula.system" substitutionGroup="mcc:formula" abstract="true" type="mcc:formula.system"/>
```

- `formula.state` are state formulæ operators as defined in CTL*, such as A, E:

```
<xs:complexType name="formula.state" abstract="true">
  <xs:complexContent>
    <xs:extension base="mcc:formula"/>
  </xs:complexContent>
</xs:complexType>
<xs:element name="formula.state" substitutionGroup="mcc:formula" abstract="true" type="mcc:formula.state"/>
```

- `formula.path` are path formulæ operators as defined in CTL*, such as G, F, X:

```
<xs:complexType name="formula.path" abstract="true">
  <xs:complexContent>
    <xs:extension base="mcc:formula"/>
  </xs:complexContent>
</xs:complexType>
<xs:element name="formula.path" substitutionGroup="mcc:formula" abstract="true" type="mcc:formula.path"/>
```

- `formula.composite` are Boolean operators between formulæ, such as disjunction or implication:

```
<xs:complexType name="formula.composite" abstract="true">
  <xs:complexContent>
    <xs:extension base="mcc:formula"/>
  </xs:complexContent>
</xs:complexType>
<xs:element name="formula.composite" substitutionGroup="mcc:formula" abstract="true" type="mcc:formula.composite"/>
```

- `formula.proposition` are atomic propositions, that are usually tightly related to Petri nets:

```
<xs:complexType name="formula.proposition" abstract="true">
  <xs:complexContent>
    <xs:extension base="mcc:formula"/>
  </xs:complexContent>
</xs:complexType>
<xs:element name="formula.proposition" substitutionGroup="mcc:formula" abstract="true" type="mcc:formula.proposition"/>
```

1.7 Composite formulæ

Composite formulæ are build using usual Boolean constants and operators: true, false, negation, conjunction, disjunction, exclusive disjunction, implication and equivalence. They are applied to operands that are subformulæ.

We use `true` and `false` in the textual flavor for Boolean constants. Note that there is *no* aliasing with `0` or `1` as in some languages. Negation is represented by `!`.

```
simple_composite_formula: "true"
                        | "false"
                        | "!" simple_formula
```

In the textual flavor, we provide n-ary operators for conjunction, disjunction and exclusive disjunction. Operators are represented using some almost-standard notations.

```
complex_composite_formula: simple_formula ("&"   simple_formula)+
                         | simple_formula ("|"   simple_formula)+
                         | simple_formula ("xor" simple_formula)+
                         | simple_formula "=>"  simple_formula
                         | simple_formula "<=>" simple_formula
```

Note that there is no operator precedence. The use of `simple_formula` as operands forbids to mix operators without parenthesis. For instance, `!f&g` is invalid and must be written explicitly `!(f&g)` or `(!f)&g`.

In the XML flavor, composite formulæ are defined as elements that inherit from `formula.composite`.

- The Boolean constant `true`:

```
<xs:element name="true" substitutionGroup="mcc:formula.composite">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:formula.composite"/>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- The Boolean constant `false`:

```
<xs:element name="false" substitutionGroup="mcc:formula.composite">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:formula.composite"/>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Negation is a unary operator, corresponding to `!` in the textual flavor:

```
<xs:element name="negation" substitutionGroup="mcc:formula.composite">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:formula.composite">
        <xs:sequence>
          <xs:element ref="mcc:formula"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Conjunction is a n-ary operator (at least binary), corresponding to `&` in the textual flavor:

```
<xs:element name="conjunction" substitutionGroup="mcc:formula.composite">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:formula.composite">
        <xs:sequence minOccurs="2" maxOccurs="unbounded">
          <xs:element ref="mcc:formula"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Disjunction is a n-ary operator (at least binary), corresponding to `|` in the textual flavor:

```
<xs:element name="disjunction" substitutionGroup="mcc:formula.composite">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:formula.composite">
        <xs:sequence minOccurs="2" maxOccurs="unbounded">
          <xs:element ref="mcc:formula"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Exclusive disjunction corresponds to the `xor` operator in the textual flavor:

```
<xs:element name="exclusive_disjunction" substitutionGroup="mcc:formula.composite">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:formula.composite">
        <xs:sequence minOccurs="2" maxOccurs="unbounded">
          <xs:element ref="mcc:formula"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Implication is a binary operator, corresponding to `=>` in the textual flavor:

```
<xs:element name="implication" substitutionGroup="mcc:formula.composite">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:formula.composite">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:formula"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Equivalence is a binary operator, corresponding to `<=>` in the textual flavor:

```
<xs:element name="equivalence" substitutionGroup="mcc:formula.composite">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:formula.composite">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:formula"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

1.8 System-wide formulæ

System-wide formulæ define properties that must hold on all the states of the system (invariants, noted `I` in the textual flavor), or that must never hold in the states of the system (impossibilities, noted `N` in the textual flavor).

```
system_formula: "I" simple_formula
              | "N" simple_formula
```

Note that `I f` is equivalent to `AG f`, while `N f` is equivalent to `! EF f`.

In the XML flavor, these two kinds of formulæ inherit from `formula.system`.

- Invariants take a subformula as parameter:

```
<xs:element name="invariant" substitutionGroup="mcc:formula.system">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:formula.system">
        <xs:sequence>
          <xs:element ref="mcc:formula"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Impossibilities take also a subformula as parameter:

```
<xs:element name="impossibility" substitutionGroup="mcc:formula.system">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:formula.system">
        <xs:sequence>
          <xs:element ref="mcc:formula"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

1.9 State formulæ

State formulæ correspond to the CTL operators A and E.

```
state_formula: "A" simple_formula
             | "E" simple_formula
```

In the XML flavor, these operators inherit from `formula.state`.

- `all_paths` takes a subformula as parameter:

```
<xs:element name="all_paths" substitutionGroup="mcc:formula.state">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:formula.state">
        <xs:sequence>
          <xs:element ref="mcc:formula"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- `exists_path` takes a subformula as parameter:

```
<xs:element name="exists_path" substitutionGroup="mcc:formula.state">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:formula.state">
        <xs:sequence>
          <xs:element ref="mcc:formula"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

1.10 Path formulæ

Path formulæ correspond to the CTL and LTL operators `X`, `G`, `F`, `U` and `W`.

```
simple_path_formula: "X" int? "~"? simple_formula
                   | "G" simple_formula
                   | "F" simple_formula
complex_path_formula: simple_formula "U" simple_formula
                    | simple_formula "W" simple_formula
```

The next `X` operator has two optional parameters. The first one is an integer that indicates the number of steps to perform, for instance `X1 f` is the same as `X f`, `X0 f` is the same as `f`, and `X3 f` is the same as `XXX f`. The second parameter defines what is the result of this operator when a state has no successors: `X f` returns `true` when applied on a state with no successors, while `X~ f` returns `false`. This can be useful for instance to detect deadlocks, as `IX~ true` will return `false` if the system contains at least one deadlock state.

The `W` operator is a weak release as usual.

Note that we may add the release operator in the future.

The XML flavor defines these operators as deriving from `formula.path`.

- `next` takes a subformula, a Boolean constant and an integer as parameters:

```
<xs:element name="next" substitutionGroup="mcc:formula.path">
 <xs:complexType>
   <xs:complexContent>
     <xs:extension base="mcc:formula.path">
       <xs:sequence>
         <xs:element ref="mcc:formula"/>
         <xs:element name="if_no_successor" type="xs:boolean"/>
         <xs:element name="steps" type="xs:integer"/>
       </xs:sequence>
     </xs:extension>
   </xs:complexContent>
 </xs:complexType>
</xs:element>
```

- `globally` takes a subformula as parameter:

```
<xs:element name="globally" substitutionGroup="mcc:formula.path">
 <xs:complexType>
   <xs:complexContent>
     <xs:extension base="mcc:formula.path">
       <xs:sequence>
         <xs:element ref="mcc:formula"/>
       </xs:sequence>
     </xs:extension>
   </xs:complexContent>
 </xs:complexType>
</xs:element>
```

- `finally` takes a subformula as parameter:

```
<xs:element name="finally" substitutionGroup="mcc:formula.path">
 <xs:complexType>
   <xs:complexContent>
     <xs:extension base="mcc:formula.path">
       <xs:sequence>
         <xs:element ref="mcc:formula"/>
       </xs:sequence>
     </xs:extension>
   </xs:complexContent>
 </xs:complexType>
</xs:element>
```

- `until` takes two subformulæ as parameter, together with a constant `weak` or `strong`:

```
<xs:element name="until" substitutionGroup="mcc:formula.path">
 <xs:complexType>
   <xs:complexContent>
     <xs:extension base="mcc:formula.path">
       <xs:sequence>
         <xs:element name="before">
           <xs:complexType>
             <xs:sequence>
               <xs:element ref="mcc:formula"/>
             </xs:sequence>
           </xs:complexType>
```

```
          </xs:element>
          <xs:element name="reach">
            <xs:complexType>
              <xs:sequence>
                <xs:element ref="mcc:formula"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element name="strength">
            <xs:simpleType>
              <xs:restriction base="xs:token">
                <xs:enumeration value="weak"/>
                <xs:enumeration value="strong"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

1.11 Petri net specific propositions

Petri net specific propositions are various atomic propositions that are related to Petri net transitions.

```
petri_proposition: "deadlock"
                 | transition "?" ("l0" | "l1" | "l2" | "l3" | "l4")
                 | transition "?"
```

- `deadlock` returns true if the current state is a deadlock:

```
<xs:element name="is_deadlock" substitutionGroup="mcc:formula.petri">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:formula.petri"/>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- `transition?level` returns true if `transition` is `level`-live[2] from the current state Petri net:

```
<xs:element name="is_live" substitutionGroup="mcc:formula.petri">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:formula.petri">
        <xs:sequence>
          <xs:element ref="mcc:transition"/>
          <xs:element name="level">
            <xs:simpleType>
              <xs:restriction base="xs:token">
                <xs:enumeration value="l0"/>
                <xs:enumeration value="l1"/>
                <xs:enumeration value="l2"/>
                <xs:enumeration value="l3"/>
                <xs:enumeration value="l4"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

  Note that in the MCC 2012, we will only use this formula from the initial state.

- `transition?` returns true if `transition` is fireable in the current state:

```
<xs:element name="is_fireable" substitutionGroup="mcc:formula.petri">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:formula.petri">
        <xs:sequence>
          <xs:element ref="mcc:transition"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

[2] http://en.wikipedia.org/wiki/Petri_net#Liveness

1.12 Integer comparison

Integer comparisons allow to compare integers or expressions that return integers, such as bounds of places, cardinality of markings, . . .

```
integer_comparison: integer "="  integer
                  | integer "!=" integer
                  | integer "<"  integer
                  | integer "<=" integer
                  | integer ">"  integer
                  | integer ">=" integer
```

In the XML flavor, all these operators inherit from `integer.comparison`:

- Integer equality:

```
<xs:element name="integer-eq" substitutionGroup="mcc:integer.comparison">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:integer.comparison">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:integer"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Integer inequality:

```
<xs:element name="integer-ne" substitutionGroup="mcc:integer.comparison">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:integer.comparison">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:integer"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Less than (strict) comparison:

```
<xs:element name="integer-lt" substitutionGroup="mcc:integer.comparison">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:integer.comparison">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:integer"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Less than or equal comparison:

```
<xs:element name="integer-le" substitutionGroup="mcc:integer.comparison">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:integer.comparison">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:integer"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Greater than (strict) comparison:

```
<xs:element name="integer-gt" substitutionGroup="mcc:integer.comparison">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:integer.comparison">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:integer"/>
        </xs:sequence>
```

```
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Greater than or equal comparison:

```
<xs:element name="integer-ge" substitutionGroup="mcc:integer.comparison">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:integer.comparison">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:integer"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

1.13 Integer expressions

An `integer` expression is defined as:

```
simple_integer: int
              | "bound" "(" place ")"
              | "|" multiset "|"
              | "(" integer ")"
complex_integer: simple_integer ("+" simple_integer)+
               | simple_integer ("*" simple_integer)+
               | simple_integer  "-" simple_integer
               | simple_integer  "/" simple_integer
integer: complex_integer
       | simple_integer
```

The `bound` operator returns the bound of the place starting from the current state. In the MCC'2012 edition, it will only be used from the initial state.

The `| |` operator returns the cardinality of a multiset, which is the total number of elements contained in the multiset, *not* the number of different elements.

We allow n-ary versions of addition and multiplications, but as in the rest of this language, there is no operator precedence, and they thus cannot be combined.

In the XML flavor, all operators inherit from `integer`.

- An integer constant is a number:

```
<xs:element name="integer-constant" substitutionGroup="mcc:integer">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:integer">
        <xs:attribute name="value" type="xs:integer"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- The bound of a place:

```
<xs:element name="place-bound" substitutionGroup="mcc:integer">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:integer">
        <xs:sequence>
          <xs:element ref="mcc:place"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- The cardinality of a multiset, which counts all the elements in the multiset:

```
<xs:element name="multiset-cardinality" substitutionGroup="mcc:integer">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:integer">
        <xs:sequence>
          <xs:element ref="mcc:multiset"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

  To count the number of distinct elements in a multiset, we can use `|set(m)|`, where `set(...)` is defined as a multiset expression that removes cardinalities of elements.

- The sum of integers:

```
<xs:element name="sum" substitutionGroup="mcc:integer">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:integer">
        <xs:sequence minOccurs="2" maxOccurs="unbounded">
          <xs:element ref="mcc:integer"/>
        </xs:sequence>
      </xs:extension>
```

```
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- The product of integers:

```
<xs:element name="product" substitutionGroup="mcc:integer">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:integer">
        <xs:sequence minOccurs="2" maxOccurs="unbounded">
          <xs:element ref="mcc:integer"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- The substraction of integer:

```
<xs:element name="substraction" substitutionGroup="mcc:integer">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:integer">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:integer"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- The division of integers:

```
<xs:element name="division" substitutionGroup="mcc:integer">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:integer">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:integer"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

1.14 Multiset comparison

Multiset comparisons are defined as below. Note that < represents inclusion, and > containment. All these comparisons work on multisets, *not* on sets!

```
multiset_comparison: multiset "="  multiset
                   | multiset "!=" multiset
                   | multiset "<"  multiset
                   | multiset "<=" multiset
                   | multiset ">"  multiset
                   | multiset ">=" multiset
```

In the XML flavor, all these operators inherit from `multiset.comparison`:

- Multiset equality:

```
<xs:element name="multiset-eq" substitutionGroup="mcc:multiset.comparison">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:multiset.comparison">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:multiset"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Multiset inequality:

```
<xs:element name="multiset-ne" substitutionGroup="mcc:multiset.comparison">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:multiset.comparison">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:multiset"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Strict inclusion comparison:

```
<xs:element name="multiset-lt" substitutionGroup="mcc:multiset.comparison">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:multiset.comparison">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:multiset"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Inclusion comparison:

```
<xs:element name="multiset-le" substitutionGroup="mcc:multiset.comparison">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:multiset.comparison">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:multiset"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Strict containment comparison:

```
<xs:element name="multiset-gt" substitutionGroup="mcc:multiset.comparison">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:multiset.comparison">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:multiset"/>
        </xs:sequence>
```

```
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- Containment comparison:

```
<xs:element name="multiset-ge" substitutionGroup="mcc:multiset.comparison">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:multiset.comparison">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:multiset"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

1.15 Multiset expressions

Multiset expressions are built from simple or complex multiset expressions. As usual, simple can be combined without the use of parenthesis, contrary to complex expressions.

An expression can be the multiset union `+`, the multiset intersection `*`, or the multiset difference `-`. It can also be `set(m)`, that returns the set built from `m`, or the marking of a place `marking(p)`. We also provide a way to build multiset constants for colors that are tuples, using `<>`.

```
simple_multiset: "<" anonymous_bindings ">"
               | "marking" "(" place ")"
               | "(" multiset ")"
complex_multiset: simple_multiset ("+" simple_multiset)+
                | simple_multiset ("*" simple_multiset)+
                | simple_multiset "-" simple_multiset
multiset: complex_multiset
        | simple_multiset
```

The contents of tuples is a `anonymous_binding` defined later. It uses the names of the colors in the PNML model, *not* their IDs. In the XML flavor, multisets inherit from `multiset`.

- A multiset constant is defined as a `term`:

```
<xs:element name="multiset-constant" substitutionGroup="mcc:multiset">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:multiset">
        <xs:sequence>
          <xs:element ref="mcc:term"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- A multiset can be the marking of a place:

```
<xs:element name="place-marking" substitutionGroup="mcc:multiset">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:multiset">
        <xs:sequence>
          <xs:element ref="mcc:place"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

  Note that the place domain can be colored or not.

- The multiset union (n-ary):

```
<xs:element name="union" substitutionGroup="mcc:multiset">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:multiset">
        <xs:sequence minOccurs="2" maxOccurs="unbounded">
          <xs:element ref="mcc:multiset"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- The multiset intersection (n-ary):

```
<xs:element name="intersection" substitutionGroup="mcc:multiset">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:multiset">
        <xs:sequence minOccurs="2" maxOccurs="unbounded">
          <xs:element ref="mcc:multiset"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

- The multiset difference (binary):

```
<xs:element name="difference" substitutionGroup="mcc:multiset">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:multiset">
        <xs:sequence minOccurs="2" maxOccurs="2">
          <xs:element ref="mcc:multiset"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

1.16 Terms and bindings

Colors are defined as terms, but the MCC models only use tuples. So, we provide a convenient notations for tuples that maps to underlying terms.

A tuple is seen as a generator associated to variables. For instance, the tuple $\langle 1, 2 \rangle$ is seen as `tuple($a, $b)` with `$a = 1` and `$b = 2`. In the notations provided for multisets, the constructor for tuples is `<...>`, and variables are implicitly named `_1`, `_2`, ...

We define them in anonymous bindings as below:

```
anonymous_bindings: ( value ("," value)* )?
```

In the XML flavor, terms are defined as a base abstract element:

```
<xs:complexType name="term" abstract="true"/>
<xs:element name="term" abstract="true" type="mcc:term"/>
```

Currently, the only terms available are tuples:

```
<xs:element name="tuple" substitutionGroup="mcc:term">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:term">
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="mcc:binding"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="binding">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="mcc:value_variable"/>
      <xs:element ref="mcc:value"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

1.17 Values

Values are defined as identifiers that refer to the name in the PNML file, *not* the id:

```
value: IDENTIFIER
```

In the XML flavor, we define an abstract element `value`, from which constants, variables and wildcard inherit:

- Constants are only integers currently:

```
<xs:element name="value-constant" substitutionGroup="mcc:value">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:value">
        <xs:attribute name="value" type="xs:IDREF"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

1.18 Places

Places are referred to by their identifier.

```
place: IDENTIFIER
```

```
<xs:complexType name="place" abstract="true"/>
<xs:element name="place" abstract="true" type="mcc:place"/>

<xs:element name="place_name" substitutionGroup="mcc:place">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:place">
        <xs:attribute name="name" type="xs:IDREF"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

1.19 Transitions

Transitions are referred to by their identifier.

```
transition: IDENTIFIER
```

```
<xs:complexType name="transition" abstract="true"/>
<xs:element name="transition" abstract="true" type="mcc:transition"/>

<xs:element name="transition_name" substitutionGroup="mcc:transition">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="mcc:transition">
        <xs:attribute name="name" type="xs:IDREF"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

1.20 Full grammar

This subsection gives the whole grammar for the property language.

```
property_set              : property+

property                  : [ "ctl" | "ltl" ] IDENTIFIER ":" formula

formula                   : complex_formula
                          | simple_formula

simple_formula            : simple_composite_formula
                          | system_formula
                          | state_formula
                          | simple_path_formula
                          | petri_proposition
                          | integer_comparison
                          | multiset_comparison
                          | "(" formula ")"

complex_formula           : complex_composite_formula
                          | complex_path_formula

simple_composite_formula  : "true"
                          | "false"
                          | "!" simple_formula

complex_composite_formula : simple_formula ("&" simple_formula)+
                          | simple_formula ("|" simple_formula)+
                          | simple_formula ("xor" simple_formula)+
                          | simple_formula "=>"  simple_formula
                          | simple_formula "<=>" simple_formula

system_formula            : "I" simple_formula
                          | "N" simple_formula

state_formula             : "A" simple_formula
                          | "E" simple_formula

simple_path_formula       : "X" INTEGER? "~"? simple_formula
                          | "G" simple_formula
                          | "F" simple_formula

complex_path_formula      : simple_formula "U" simple_formula
                          | simple_formula "W" simple_formula

petri_proposition         : "deadlock"
                          | transition "?" ("l0" | "l1" | "l2" | "l3" | "l4")
                          | transition "?"

integer_comparison        : integer "="  integer
                          | integer "!=" integer
                          | integer "<"  integer
                          | integer "<=" integer
                          | integer ">"  integer
                          | integer ">=" integer

integer                   : complex_integer
                          | simple_integer

simple_integer            : INTEGER
                          | "bound" "(" place ")"
                          | "|" multiset "|"
                          | "(" integer ")"

complex_integer           : simple_integer
                          | simple_integer ("+" simple_integer)+
                          | simple_integer ("*" simple_integer)+
                          | simple_integer "-" simple_integer
                          | simple_integer "/" simple_integer

multiset_comparison       : multiset "="  multiset
                          | multiset "!=" multiset
                          | multiset "<"  multiset
                          | multiset "<=" multiset
                          | multiset ">"  multiset
                          | multiset ">=" multiset

multiset                  : complex_multiset
                          | simple_multiset

simple_multiset           : "<" anonymous_bindings ">"
                          | "marking" "(" place ")"
                          | "(" multiset ")"
```

```
complex_multiset          : simple_multiset ("+" simple_multiset)+
                          | simple_multiset ("*" simple_multiset)+
                          | simple_multiset "-" simple_multiset

anonymous_bindings        : ( value ("," value)* )?

value                     : IDENTIFIER

place                     : IDENTIFIER

transition                : IDENTIFIER
```

1.21 ANTLR grammar

This subsection gives a (very) simple grammar in the ANTLR format. The file is available at `http://mcc.lip6.fr/properties/mccproperty.g`. Note that we make use of backtracking. If you build a better grammar, we would like to replace this one with it.

```
grammar mccproperty { backtrack=true; };

IDENTIFIER:
  ('a'..'z'|'A'..'Z'|'0'..'9'|'_')+;

INTEGER:
  '0'..'9'+;

EOL:
  ('\r' | '\n')+;

property_set:
  property+
  ;

property:
  ( 'ctl' | 'ltl' )? IDENTIFIER ':' complex_formula EOL
  ;

simple_formula:
    'true'
  | 'false'
  | '(' complex_formula ')'
  | '!' simple_formula
  | 'I' simple_formula
  | 'N' simple_formula
  | 'A' simple_formula
  | 'E' simple_formula
  | 'X' INTEGER? '~'? simple_formula
  | 'G' simple_formula
  | 'F' simple_formula
  | 'deadlock'
  | IDENTIFIER '?' ('l0' | 'l1' | 'l2' | 'l3' | 'l4')?
  | complex_expression
    ('='
    | '!='
    | '<'
    | '<='
    | '>'
    | '>='
    ) complex_expression
  ;

complex_formula:
    simple_formula ( ('&'   simple_formula)+
                   | ('|'   simple_formula)+
                   | ('xor' simple_formula)+
                   | ('=>'  simple_formula)
                   | ('<=>' simple_formula)
                   | ('U'   simple_formula)
                   | ('W'   simple_formula)
                   )?
  ;

complex_expression:
    simple_expression
    ( ('+' simple_expression)+
    | ('*' simple_expression)+
    | ('-' simple_expression)
    | ('/' simple_expression)
    )?
  ;

simple_expression:
    INTEGER
  | 'bound' '(' IDENTIFIER ')'
  | '|' complex_expression '|'
  | '<' anonymous_bindings '>'
  | 'marking' '(' IDENTIFIER ')'
  | '(' complex_expression ')'
  ;

anonymous_bindings:
    ( IDENTIFIER (',' IDENTIFIER)* )?
  ;
```

1.22 Restrictions for the 2012 Model Checking Contest

During the Contest, we will provide the following kinds of properties:

- Composites of place bounds using <=, such as:

```
bound(p)<=3
bound(p1)<=3 & bound(p2)<=2
bound(p1)<=3 | bound(p2)<=2
(bound(p1)<=3 & bound(p2)<=2) | bound(p3)<=1
```

- Composites of transition liveness for P/T nets only, such as:

```
t?l2
t1?l2 & t2?l0
t1?l2 | t2?l0
(t1?l2 & t2?l0) | t3?l4
```

- Detection of deadlocks, such as:

```
N deadlock
```

- Reachability formulæ using composite, cardinalities, markings and transition fireability, such as:

```
I( <0,0> < marking(p1) )
N( |marking(p1)|=|marking(p2)| )
I( <0,0> < marking(p1) => |marking(p2)|=|marking(p3)| )
N( marking(p1) != marking(p2) )
```

- CTL formulæ using composite, cardinalities, markings and transition fireability, such as:

```
AG( <0,0> < marking(p1) )
AF( |marking(p1)|=|marking(p2)| )
AG( <0,0> < marking(p1) => EF |marking(p2)|=|marking(p3)| )
AG( marking(p1) != marking(p2) )
```

- LTL formulæ using composite, cardinalities, markings and transition fireability, such as:

```
G( <0,0> < marking(p1) )
F( |marking(p1)|=|marking(p2)| )
G( <0,0> < marking(p1) => F |marking(p2)|=|marking(p3)| )
G( marking(p1) != marking(p2) )
```

Note that we will use integer expressions, such as addition, but not multiset expressions. They are given in this manual to give a preview of the language for the next edition.

2 Tools

We provide a tool to convert properties expressed in the textual flavor to the XML flavor. Moreover, the XML Schema has been designed to allow automatic generation of parsers, for instance in C++, Java or Python. You are free to create your own parser, or to use the provided one, or to use an automatically generated XML parser.

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

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

The textual parser can be downloaded from `http://mcc.lip6.fr/properties/property-language.tar.bz2` using:

```
wget http://mcc.lip6.fr/properties/property-language.tar.bz2
```

Note to the users of the official MCC system image The official MCC system image already contains the XML Schema, classes in C++, java and python to access the XML structure, and a tool to convert the textual flavor to the XML one.

2.1 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 -o 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 \
    mccproperty.xsd
```

It generates two files: `mccproperty.hxx` and `mccproperty.cxx` in the `src/cxx/` directory.

The documentation for these classes is generated using Doxygen (`http://www.stack.nl/~dimitri/doxygen/`), with the following command:

```
doxygen doc/cxx/mccproperty.doxygen
```

You can open the documentation at `doc/cxx/html/index.html`. All generated classes belong to the `mcc` namespace.

2.2 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 -o src/java/
xjc -d src/java/ -p mcc mccproperty.xsd
```

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

As usual with Java code, the documentation is generated using javadoc:

```
javadoc -d doc/java/html/ src/java/*
```

You can open the documentation at `doc/java/html/index.html`. All generated classes belong to the `mcc` namespace.

2.3 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/mccproperty.py mccproperty.xsd
```

2.4 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.

2.5 How to compile and use the C++ textual parser?

To compile these tools, you need the Boost Spirit library, available at `http://www.boost.org`.

```
# Define $CC as your C++ compiler, for instance:
# CC=g++
# or
# CC=clang++
cd src/cxx
$CC -O3 -Ilibxsd -lxerces-c mccproperty.cxx txt-to-xml.cc -o txt-to-xml
cd ../..
```

2.6 How to compile and use the C++ XML parser?

To compile these tools, you need the Boost Spirit library, available at `http://www.boost.org`. The source file given is a dummy one: it only shows how to load the XML file.

```
# Define $CC as your C++ compiler, for instance:
# CC=g++
# or
# CC=clang++
cd src/cxx
$CC -O3 -Ilibxsd -lxerces-c mccproperty.cxx read-xml.cc -o read-xml
mv txt-to-xml ~/MCC-PATH/bin
cd ../..
```