Draft. The revised and slightly shortened version of the text appeared as Piotr SZWED XQPN – colored Petri nets for processing XML data with XQuery language — Electrical Review; ISSN 0033-2097. — 2010 R. 86 nr 9 s. 221–225.

XQPN – colored Petri nets for processing XML data with XQuery language

Piotr SZWED

AGH - University of Science and Technology

Abstract

The paper presents basic concepts of a new class of colored Petri nets XQPN (XQuery Petri Nets) designed for processing XML data assigned to places. Tokens in XQPN nets correspond to nodes of XML documents. Token values that are deleted, read or created by transitions are specified by XQuery expressions assigned to arcs, whereas their number is determined by multiplicity parameters of input arcs. We describe the syntax of XQPN nets, algorithms of binding calculation and net execution and discuss design decisions accompanying the prototype implementation.

Introduction

The paper presents XQPN nets (XQuery Petri Nets), a class of colored Petri nes allowing operations on XML documents assigned to places. XQPN nets are proposed as a modeling and prototyping tool for a large class of applications that internally process XML data.

Petri net is a bipartite directed graph whose nodes belong to two disjoint sets: places and transitions. Arcs can only connect places with transitions and transitions with places. Transitions represent actions that can be performed in the system modeled by a net, whereas places represent its resources. Execution of a transition is followed by a consumption of resources in its input places and creation of new resources in the output places. Depending on the current assignment of resources to places, i.e. the *marking*, some transitions are enabled and can be executed. Petri nets have a traditional graphical form: places are symbolized by circles and transitions by rectangles or bars. Elements stored in places are called tokens.

The most popular classes of Petri nets are Place Transition nets (PT-nets [1]) and Colored Petri Nets (CPN)[2]. In PT nets tokens are indistinguishable. In colored nets each place is assigned with a type (color) and may contain multiple different values of the type (a multiset over the place type). The net syntax is enhanced by adding expressions to arcs. They are responsible for selecting tokens in transitions' input places or contain a recipe for how to create tokens in output places.

In XQPN nets places store data in form of XML documents [3]. Places contents may be treated as in CPN nets, as sets of distinguishable objects that in the case of XQPN correspond to various nodes in the document's hierarchy. Another shared property with colored nets is the presence of expressions assigned to arcs. In XQPN they are expressions of XQuery [4] language being a standard for querying and manipulation of XML data. In PT-nets arcs can be attributed with weights specifying how many tokens a transition will remove from its input places or produce in its output places. In XQPN nets input arcs are attributed with multiplicity parameter playing similar role as weights.

The inspiration for development of XQPN nets was the idea of using colored nets as a tool for formal specification of test cases for internet applications and web services. It was assumed, that tests would be driven by transitions executed in a net serving as the model [of the system and its environment]. We were searching for a modeling language that would rather be loosely typed, provide direct support of XML processing and have certain primitives missing in CPN, like atomic access to multiple tokens contained in a place. The performance issues (e.g. number of transitions executed in a time unit) were considered as not essential in this case. A successful implementation of this approach was described in [11]

The paper is organized as follows. In Section 2 some basic constructs of XQuery language are presented; the main concepts of XQPN nets are shown in Section 3. Section 4 describes methods of binding calculation and execution of a transition. In Section 5 design decisions preceding prototype implementation are discussed. Finally, we present some concluding remarks in Section 6.

XQuery Language

XQuery (XML Query Language) is a language for querying XML data, that can appear either as a standalone XML document or data stored in a data base provided with an XML interface. The language provides means for data extraction preserving independence of the physical data representation. The works on language definition started in 2001, finally in January 2007 its specification became an official recommendation of W3C organization.

The data model for XML content, that is used internally by XQuery takes the form of a tree containing several kinds of nodes, among them document nodes, elements, attributes and text nodes. XQuery uses XPath[] expressions to address and select nodes inside XML documents. As the language was intended only for querying data, it is not capable of modifying the document content. (However, there are efforts to define an improved version of the language XQuery Update [6], allowing data modification.)

In XQuery a value is a sequence of XML nodes or values of atomic types. There is no syntactical difference between a sequence containing one element and its single element. XQuery sequences can be strongly typed, if a sequence contains elements whose type is determined by a schema definition or loosely typed if no schema is referenced. Values represented by sequences are returned from XQuery instructions and can be assigned to variables.

Apart from the capability of extracting information from source documents, XQuery can dynamically create nodes and use built in functions to change their values; this feature can be used for transforming the content and structure of nodes in the resulting sequence. The other capability offered by XQuery is joining data originating from different sources.

XQuery is a programming language. It allows declaring variables, defining functions and even calling external functions, defined outside its execution environment.

XQuery language constructs include loops, assignments and conditionals. Loops are often referred as FLWOR statements (FLWOR is an acronym for keywords `for`, `let`, `where`, `order by`, `return`). An example of FLWOR statement is:

```
for $x in /bookstore/book
where $x/price>30
order by $x/title
return $x/title
```

This expression iterates over the sequence of nodes returned by the XPath [@] expression `bookstore/book` and returns ordered sequence of nodes `title` being children of nodes `book` that match the criterion `price>30`.

The resulting sequence can be returned from a query, but it also can be used as a value in an assignment, e.g.

```
let $y := for ….
```

and further processed in subsequent instructions. Another XQuery statement is a conditional:

```
if(condition)then exp1 else exp2.
```

Both expressions exp1 and exp2 are obligatory; they however can be defined as an empty sequence value ().

The FLWOR and if –then-else statments can not be treated as classical structural instructions of high level languages. They rather resemble operators that return values, e.g. the if-then-else statemant is similar to the ternary operator ? : appering in C, C++, Java and C# languages.

We have selected XQuery as the basic element of XQPN because it is a widely recognized standard of XML processing recommended by W3C organization, moreover, many vendors offer commercial and free XQuery engines suitable for processing XML documents in textual form (located in memory or in a file) [@, @]. as well as stored in a data base, e.g. Microsoft SQL Server, [], IBM DB2 [@] or Oracle [@].

Definition of XQPN nets

We will start the presentation of XQPN nets by providing their definition enumerating components of the nets. While referring to such concepts as XML document or a query we will not give their definitions. An XML document can be treated either as a textual form (a sequence of tags) or as a tree-structured data model [@], whose specification is used to give a semantics to XQuery expressions. Similarly, a query in some contexts can be treated as an expression, whereas in other as a function that maps a particular instance of data model to a sequence o elements resulting from the evaluation of a query expression. When speaking of XML documents we will refer to well formed documents. It is a natural consequence of the assumption that a document is represented by a data model.

Definition 1

XQPN net is defined as the tuple $XQPN = (P; T; A; Q;W; I;G)$, where

- P - is a finite set of places,
- T - is a finite set of transitions, $P \cap T = \emptyset$;
- $A \subseteq P \times T \cup T \times P$ is a set of arcs,
- Q: $A \rightarrow \mathbf{Q_N} \subset \mathbf{Q}$, is a function, assigning XQuery queries to arcs; we will denote a set of all queries by $\mathbf{Q}$; $\mathbf{Q_N} \subset \mathbf{Q}$ is a set of queries returning sequence of nodes,
- W: $P \times T \rightarrow (\mathbf{N} \cup \{*\}) \times (\mathbf{N} \cup \{*\}))$ is a function that assigns multiplicity to input arcs of transitions,
- I: $P \times T \rightarrow \{delete; read\}$ is a function that assigns an input mode to input arcs, for *delete* mode the transition linked by the arc will remove tokens from the input place, for *read* mode it will only query their values and leave them intact,
- G: $T \rightarrow \mathbf{Q_B} \subset \mathbf{Q}$ is a function assigning guards to transitions (guards belonging to the set $\mathbf{Q_B}$ are queries returning Boolean values).

Definition 2

The marking in XQPN is a function $M: P \to \mathbf{M}$ assigning XML documents from a set M to places. By M(p) we will denote a document assigned to the place $p \in P$. For technical reasons we assume that the name of the root of a dokument M(p) can be used as the identifier of the place p. The names of documents roots belonging to the set M(P) must be unique.

Arcs

Arcs connect transitions with places and specify transformations of the data stored in places surrounding an executed transition. Transitions may create, delete and read data. Similarly to CPN nets, updating a value of an element stored in a place can be modeled as replacing it by a new element. (Update operations are not implemented in Query 1.0 as well.)

For a given transition $t \in T$, the arcs from the set $\{ (p, t) : (p, t) \in A\}$ are called input arcs and arcs from $\{ (t, p) : (t, p) \in A\}$ output arcs. Correspondingly, we call input and output places as places connected with input and output arcs. A transition t may remove elements from its input places or read them. For an input arc $a \in P \times T \cap A$, the mode of operation is determined by the I(a) arc attribute. The graphical representation of the deleting arc, for which $I(a)$=*delete*, is an arrow, for reading arc, where $I(a)$=*read* is a line.

Arcs are assigned with additional attributes: a query expression and a multiplicity that is applicable for input arcs only.

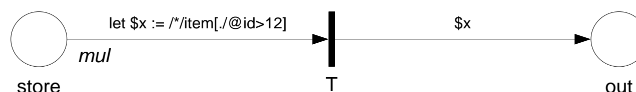
Expressions

Expressions assigned to input arcs (p, t) should return sequences of nodes in the source document $M(p)$ stored in the input place p. In particular, the expression can take the form of XPath expression or FLWOR statement. In the later case it is required that the statement do not create new nodes in the output sequence. The result of expression evaluation, to be useful, must be assigned to a variable that can be referenced in other expressions, e.g. in output arcs' expressions. The typical form of input arc expression is, thus: `let var:= expression`. A variable appearing on the left hand of the assignment will be called the arc's input variable. Expressions appearing at input arcs of a transition may contain references to input variables defined at other arcs. In this case a query containing a reference to a variable `var` is dependent on the query defining it, i.e. the query where the variable *var* is bound by an expression. It is assumed, that in a correctly specified net, queries assigned to input arcs of a transition can be ordered according to the dependency relation.

Expressions assigned to output arcs of a transition specify sequences of nodes that will be added to output places during its execution. They may contain references to input variables or define own variables that are bound by expressions, e.g. iteration variables of the `for` statement.

Example 1

Fig. 1 presents an example of a simple transition moving nodes from its input place *store* to the output place *out*.



Rys. 1 Tranzycja przesuwająca zbiory znaczników określone przez wyrażenia

Fig. 1 Transition moving set of nodes specified by arc expressions

Let us assume, that for the current marking the place `store` contains the data:

```
<store>
     <item id='7'/>
     <item id='12'/>
     <item id='13'/>
     <item id='21'/>
     <item id='27'/>
</store>
```

As the transition T is fired, the input arc expression `let $x := /item[./@id>12]` will be evaluated an the variable `$x` will be assigned with the node sequence:
`(<item id='13'/> <item id='21'/> <item id='27'/>)`
Then the nodes in `$x` will be removed from the place `store` and added to the place `out` as specified in the output arc expression: `$x`.

Constants in expressions
A typical construct in colored Petri net are constants appearing as arc expressions. [Constants at input arcs are used for removing indicated tokens or testing their presence in the connected input place; constants at output arcs define tokens to be added.]

In the case of XQPN nets, constants correspond to sequences of XML tags conforming the syntax of XML documents. They may appear only as output arc expressions. A constant define then a node or several nodes that will be inserted to the document stored in the connected output place as the transition is executed.

For input arcs (p, t) linking a place p with a transition t an expression defining a constant would specify a node that is external to the data model of the document $M(p)$. Such node can not be matched to any node in $M(p)$ on the identity basis; however, the equality of constants and nodes in $M(p)$ can still be checked. Instead of using constant, it is proposed to use a query calling the function `deep-equal()` that compares the content of two nodes and their descendants.

As example, for the constant `<item id='7'/>` the query may take form of:

```
for $i in //
return if (deep-equal($i, <item id='7'/>))
then $i else ()
```

The XQPN editor may be equipped with a handy shortcut, that writes such query for the specified constant.

Multiplicity

Multiplicity in XQPN nets corresponds to weights of arcs in PT-nets. The parameter is used to control the exact number of nodes that a transition will read or remove from its input places, therefore it can be applied to input arcs only. Multiplicity is defined as a pair of two numbers (symbols) [*min*, *max*] specifying bounds. A transition is enabled if input arc expression can be evaluated to a sequence containing at least *min* tokens (nodes). If more then *min* tokens are available, then the length of the resulting sequence can not exceed the *max* parameter (the sequence can be arbitrary restricted if the actual length is greater then *max*). If *min* and *max* are equal, they can be marked on the diagram as a single number. Apart from numbers, specification of bounds may contain a symbol of ‘*’ corresponding to *all tokens* (infinity). In most cases values of ‘1’ or ‘*’ are used as multiplicity parameters.
Returning to the example presented in Fig. 1:

- Setting multiplicity parameter `mul` to 4 (or [4,4]) implies that for a given marking the transition is not enabled
- For `mul` defined as *, the transition is enabled and will remove all matching elements from the input place
- For `mul` defined as 1 ([1 1]) exactly one element will be arbitrary selected from two nodes satisfying the predicate `[./id>12]`. It will be then removed from the place `store` and added to the place `out`. Other element will remain in the input place.

The multiplicity introduces nondeterminism into the net bahavior. On the stage of preparing of the execution of a transition an arbitrary decision is made which elements in its input places will be actually accessed and then processed. The mechanism of selection according to the multiplicity can be compared with nondeterministic binding of free variables in CPN nets.

In PT-nets it is possible to assign weights to output arcs. Similar solution (e.g., assigning multiplicity for output arcs) was not introduced to XQPN nets, as output arc expressions enable the very precise control of nodes that are added to output places (both their values and number).

Execution of XQPN nets

Execution of XQPN nets consists in subsequent executing (firing) of transitions and updating the net state. Execution of a transition has two stages: preparatory, when a binding for input variables is established and the stage of actual execution consisting in removing nodes (XML tags) from input places (connected with the executed transition with deleting arcs) and adding nodes to output places. From those two stages, the first is far much more complex, as regards algorithmic issues.

Bindings

In CPN nets a *binding* is an assignment of values to free variables that appear in arc expressions or a guard of a transition. A transition is enabled and can be fired if there exists a binding for which values assigned to free variables match the tokens present in corresponding input places (and the guard of the transition evaluates to the true value). Potentially, for a given net marking there may exist a number of appropriate bindings for a transition, thus it can be executed for different values of variables and different combinations of tokens removed form input places and inserted to output places.
The consequence of selecting XQuery language for specifying expressions is that the semantics of binding in XQPN differs with respect to CPN nets. In XQPN variables appearing in arc expressions are either bound in the enclosing expression, or are references to the variables defined and bound in expressions assigned to other arcs. This in general corresponds to the interpretation of binding in CPN nets. However, variables appearing in queries may also change value, as they are executed.

A small modification to the net from Example 1 is presented in the Fig. 2. The output arc is assigned with a *FLWOR* expression containing the variable `$y` bound by expression `in $x`. During the query execution the variable iterates over the sequence of nodes `$x`, thus, it is not possible to assign to it a value remaining constant during the transition execution.

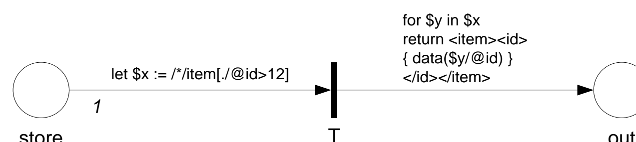


Fig. 2 FLWOR expression assigned to the output arc

In XQPN nets only the bindings for input variables, i.e the variables appearing as l-values in input arcs expressions, are required. However, the algorithm of their calculation has an intermediary step, during which bindings are first established for hidden variables related to input arcs, further referenced as `$inputnodes-i`. The values of them are then used instead of the documents stored in input places to calculate actual bindings of input variables.

Let us assume, that we want to calculate a binding for an input variable `$xi` appearing in the expression `let $xi := exp-i` of *i*-th input arc of a transition.

Algorithm of binding calculation

1. Execute the query `exp-i` and store the result in a temporary value:

```
let $tmp :=  exp-i,
```

2. Calculate the sequence of nodes being the children of the document's root node, for which the query `exp-i` would return the sequence `$tmp`:

```
$inputseq-i = local:upcast($tmp)
```

3. If the number of elements in the sequence: `count($inputseq-i)` is less then min_i value of the multiplicity [min_i, max_i] assigned to the *i*-th arc, then STOP, the binding is not enabled.

4. If max_i > `count($inputseq-i)`, then limit the number of elements in the sequence by selecting exactly max_i elements; in the other case, leave it unmodified. In the algorithm implementation, those conditions are wrapped by a function returning the sequence restricted according to multiplicity parameters. The result is then stored in the variable `$inputnodes-i`.

```
let $inputnodes-i := local: arbitray_select(inputset-i)
```

5. Finally, calculate the binding of `$xi` as:

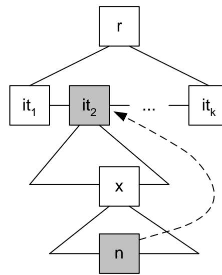
```
let $xi := $inputnodes-i exp-i
```

If an expression attributed to the input arc is not defined as an assignment, the last step is omitted. However, for deleting arcs the binding of the hidden variables `$inputnodes-i` must be calculated, as they specify the tokens to be deleted from the input places. For expressions without assignment attributed to reading arcs the algorithm finishes at the step 3 (arcs are used only to establish the enabledness of a transition and neither for deleting tokens, nor for binding variables)

The `local:upcast()` function is defined as follows:

```
declare function local:upcast ($n as node()*) as node()* {
for $i in /*/*
return if ($i intersect $n or $i//* intersect $n) then $i
else ()
};
```

Its effect is depicted in the Fig. 3. The node *n* contained in the sequence passed as the argument is a descendent of the top-level node it_2 (being a child of the root node *r* of a document assigned to a place). In the step (2) of the algorithm, the node it_2 will be inserted in the `$inputseq-i` and will become a potential element of the sequence `$inputnodes-i`.



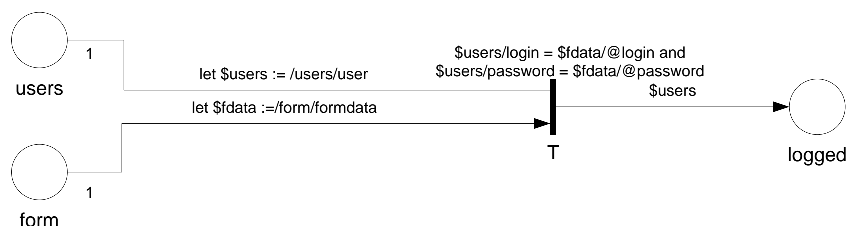
Rys. 2 Działanie funkcji `upcast()`

Fig. 3 The upcast() function returns top-level nodes whose descendants are contained in its argument

This somehow complicated solution can be justified as follows. XQPN nets are intended to model systems processing XML data and places should be treated first of all as collections of objects. e.g.: database records. In opposition to CPN nets, they are loosely typed: top level nodes in a document assigned to a place can have different structure and represent different types. However, when reading or deleting nodes from input places we should refer to objects as the whole, not their attributes, as it makes no sense to delete an attribute of a single database record. On the other hand, in XQPN nets there is no specification and control of types (colors) and a great attention should be made while specifying output arcs expressions to avoid insertion of not intended types of objects to output places of a transition.

Guards

Guards are logical expressions assigned to transitions. A guard of a transition may refer only to its input variables. A transition is enabled for a given binding if the guard evaluated for the binding returns true. Guards may be omitted in the specification; in this case they are treated as `(true)` constants. Fig. 4 shows an example of application of guards to describe required data constraints. The transition *T* models a process of logging in. The input place *users* contains records (*user*) specifying user identifiers and passwords, the place *form* contains, possibly multiple, data that can be entered to the logging form (*formdata*). As the transition fires, user data matching those entered in the form will be inserted to the place logged.



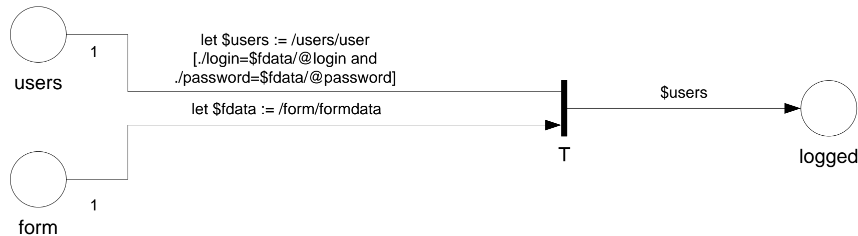
Rys. 3 Tranzycja modelująca logowanie użytkownika

Fig. 4 Transition modeling the process of logging in

Execution of the transition requires that first a binding for variables `$users` and `$fdata` should be selected. According to multiplicity specification, they both must be assigned with values being sequences containing exactly one node (of type `user` for $users and correspondingly `formdata` for `$fdata`). In the current prototype implementation of XQPN execution module an ineffective algorithm for guards checking is used: they are tested after a candidate binding is calculated. However, in many cases the guards can be eliminated. Fig. 5 shows a modified net from Fig. 4. In this example analogous behavior was modeled without guards. Elimination of guards was performed by introduction of a dependency between input

arc expressions causing that a binding for the variable `$fdata` should be selected before calculating a binding for the variable `$users`.

Process of binding calculation is actually a complex recursive procedure in which query expressions assigned to input arcs are first ordered according to the dependency relation and then bindings for subsequent variables are fixed. Steps (4) and (5) are in fact placed in a loop, in which calls to the `arbitray_select()` function return next subsequences meeting constraints imposed by multiplicity parameters of input arcs.



Rys. 4 Tranzycja, w której dozór zastąpiono zależnością

Fig. 5 Rewritten transition: the guard was replaced by dependent expressions

In this point a question arises whether it is computationally feasible to find a binding or enumerate all bindings for given multiplicity values? In a general case, the answer is no. Let us analyze a simple example: for n top level nodes in the document assigned to an input place linked by an arc with multiplicity [k, k], where k<n. subsequent calls to `arbitray_select()` function should enumerate all n!/(n-k)! subsequences. However, for a typical situation, where the multiplicity is set to one, or *, the k variable in the formula would have the values 1 or 0, what gives computationally feasible numbers of possible bindings (n for the multiplicity 1 and 1 for the multiplicity *).

Firing

An enabled transition can be fired (executed). Firing of a transition consists in updating data in its input and output places. Documents in input places connected with a transition by deleting arcs are udated with XQuery queries using the `except` operator:

```
return <pl-i>{$/pl-i/* except $inputnodes-i}</pl-i>,
```

where `<pl-i>` is a root tag of a document assigned to *i*-th input place.

For output places p_j an output sequence of nodes `$outputnodes-j` is calculated by executing the query assigned to the output arc (t, p_j). Then the place content is replaced by the sequence returned by the query:

```
return <pl-j>{$/pl-j/* union $outputnodes-j}</pl-j>,
```

where `<pl-j>` is a root tag of a document assigned to *j*-th output place.

It should be emphasized, that assumed in XQPN operations on XML data models are based on *identity* but not *equality* of nodes (two nodes with the same content are treated as distinguishable elements). It is admissible that a place contains several nodes with an identical content. A transition connected with such place with a deleting arc having the multiplicity 1 will delete the only one element appearing in the sequence `$inputnodes-i`, but not all elements that can be found equal to it. From this perspective XQPN nets preserve the very specific property of CPN nets: documents assigned to places can be interpreted as multisets of token values, where each token value instance corresponds to a node in an XML document.

Implementation

Two prototype modules for execution of XQPN nets were implemented and tested for several dozens of test cases in form of small nets containing a few transitions and places. The modules were based on free software packages offering the XQuery support: Altova XML[9] and Saxon [8]. Chronologically, the first execution module was implemented in C# language using Altova XML package. It was applied in experiments with systematic testing of web applications by automatic generation of the cases based on XQPN specification [cee-set].

However, some drawback of the solution caused that we decided to compare the utility of Altova XML and Saxon libraries for XQPN implementation, and finally chosen the Java platform and Saxon libraries as its base. This decision also influenced the specification of XQPN nets. The form presented in the paper is a refined version that defines the semantic of the net in reference to XML data model that is not accessible in Altova XML library, whereas is fully accessible in Saxon.

The Altova XML package is available as COM component for Windows. It can be used from any .NET compliant languages, it provides also a wrapper for Java. Saxon-B is distributed as a collection of Java packages. It provides three software interfaces: *s9api*, *XQJ* and native interface using internal data structures. The library was also recompiled and released for .NET platform.

We have made a test whose goal was assessment of the performance of the compared libraries on a very simple net consisting of one transition that calculates two sequences of nodes from input places, then makes their intersection and adds the result to an output place. The input places were feed with randomly generated XML data that were loaded into the memory (what is not reflected in the measured time). The result was converted to a textual form, what was obligatory, as Altova XML component offers only textual input and output.

Test results (processor Pentium M 1.7Ghz, 1 MB RAM) are presented in Table 1.

Table 1. Execution time in seconds

Package and interface	Number of top level nodes in the document			
	100	1000	10000	100000
Altova XML	0.02	0.18	1.82	19.49
Saxon s9api DOM	0.25	8.49 sec	85.2	928
Saxon XQJ	0.54	0.71	1.31	7.01
Saxon XQJ precompiled queries	0.09	0.36	1.37	12.33

The tests have shown a greater efficiency of Altova XML component, especially for small data sets. On the other hand, Saxon libraries have a constant overhead visible for small data sets that vanishes for larger documents. The only exception is the execution of queries against the DOM data model, what can be justified by the discussion on the internal solutions applied in the Saxon package [10]. The drawback of the Altova XML component is a very narrow COM interface, that precludes more advanced manipulations, e.g.: execution queries for various bindings of variables declared in XQuery expressions as external variables. Actually, the Altova XML component gives no access to the XML data model, what can be considered essential for proper implementation of XQPN semantics.

Finally, we have decided to develop the implementation on the Java platform with use of the offered by Saxon *XQJ* interface without precompiled queries. The same object of *XQJ API* representing a query (`XQExpression`) with attached XML document encompassing the content of input places is reused several times. During the process of binding calculation the

same queries can be executed many times. Additional tests have shown that while a single call of to complex procedure of binding calculation consisting in execution of a few XQuery queries has taken 0.54 sec, calling it 100 times increased the execution time to 0.87 sec, and not to 54 sec, as it can be supposed.

Implementation of the algorithm for binding calculation is quite an interesting issue, because its steps can be realized either in XQuery environment, or outside, in the platform language. A possible solution is the generations of the monolithic XQuery queries returning aggregated information about variables bindings and sequences of nodes in input places that were a base of their calculation. The other approach may consist in multiple executions of small queries that are controlled by an algorithm implemented outside the XQuery environment. In the prototype implementation the second solution was selected.

Another interesting issue is rewriting of queries. The point where rewriting might occur very useful is the automatic elimination of guards. The currently implemented algorithm of binding calculation is inefficient for the net shown in Fig. 4, however, quite operational for the net in Fig. 5. If the number of top-level nodes in the place *form* is m and in *users* is n, then in the first case the number of calculated and tested bindings is $n \cdot m$, whereas in the second case is up to m. We can assume that in the case of the real system model, n can range about 10000, while m will be a relatively small number.

Conlusions

The paper presents XQPN nets,a class of colored Petri nets designed for the manipulation of XML data stored in places. The characteristic property of XQPN nets is the use of XQuery language to define expressions assigned to arcs and to perform operation on the data.

XQPN nets are proposed as the tool for specification, simulation and testing of the systems using XML as internal data exchange format. The domain of application is quite large, it may include internet applications, web services, and SOA solutions based on execution of BPEL processes.

To summarize key differences between XQPN nets and classical CPN nets:

- Marking in XQPN is an assignment of XML documents (XML data models) to places. XQPN nets assume no constraints on the data structure. It may conform to a schema definition; however, a schema definition is not required in the specification. In the consequence XQPN nets do not use the notion of type assigned to a place, they can be considered as loosely typed.
- In some cases marking in XQPN can be considered as a multiset over a type; this regards the situation, where all top-level nodes have an identical structure and possibly equal contents. However, XML data model is more flexible then a multiset of tuples appering in the CPN specification.
- XQPN nets provide atomic operations to access all tokens from a certain places (enough to set an input arc multiplicity to *). We found this issue crucial for modeling data base operations, e.g accessing all records from a table matching certain criteria. This can be expressed in CPN nets, but with a loose of atomicity.

At present the prototype implementations of XQPN nets are relatively slow, average timing of firing a transition is about a second or a fraction of second. This however, was not considered as a drawback during the experiments with testing real web applications, where XQPN model was used as the specification of test cases [cee-set].

References

[1] Reisig W.: Petri Nets – An Introduction, EATCS Monographs on Theoretical Computer Science, Volume 4. Springer.

1985, tłum. na jęz. polski: *Sieci Petriego – Wprowadzenie*, WNT 1988.

[2] Jensen K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use, Vol. I-III, Springer Verlag, 1995/96.

[3] W3C: Extensible Markup Language (XML), http://www.w3.org/XML/

[4] W3C: XQuery 1.0: An XML Query Language, http://www.w3.org/TR/xquery/

[5] W3C: XML Path Language (XPath) 2.0, http://www.w3.org/TR/xpath20/

[6] W3C: XQuery Update Facility 1.0, http://www.w3.org/TR/xquery-update-10/

[7] XQuery 1.0 and XPath 2.0 Data Model (XDM), http://www.w3.org/TR/xpath-datamodel/

[8] http://www.saxonica.com/

[9] http://www.altova.com/

[10] Kay M.: Ten ReasonsWhy Saxon XQuery is Fast, Data Engineering, December 2008 Vol. 31 No. 4

[11] Szwed P., Wadowski D., Paździora K. A framework for testing Web services based on XQPN Petri nets, in Huzar Z., Nawrocki J., Szpyrka M. (eds) IFIP 2009: Software Engineering Techniques in Progress. AGH University of Science and Technology Press, Kraków 2009, 53–66.