A Framework For Testing Web Services Based On XQPN Petri Nets

Piotr Szwed[1], Dariusz Wadowski[1], Krzysztof Paździora[1]

[1] Institute of Automatics, AGH University of Science and Technology, al. Mickiewicza 30, 30-060 Kraków, Poland
pszwed@ia.agh.edu.pl, slashwadowice@tlen.pl, gryzli83@gmail.com

Abstract. A framework for testing web applications basing on web services is presented. The framework uses a formal specification of the tested system in form of XQPN net, a colored Petri net allowing manipulation of XML data. Tests of real application are driven by transitions in XQPN net. After performing transitions in the XQPN model and corresponding calls to the tested system, the equality between the states of model and tested system is checked. The paper describes XQPN nets and its application in modeling of web applications and presents architecture and main components of the framework.

Keywords: XQuery, Petri net, testing, web service

1 Introduction

The need of testing of web services and web service based applications emerged with the rapid development of this technology. At present there exist several tools for carrying out unit testing and performance analysis of web services, e.g. *soapUI* [1] or tools for ASP.NET platform included in Visual Studio. They offer a capability of defining individual test cases specifying called operation of a web service, its parameters and assertions concerning expected results. They allow also running a manually prepared sequence (scenario) of test cases. One of the recent initiatives in this field is Web Service Test Forum [2] a community grouping leading companies that decided to develop and publish testing scenarios that can be checked against various implementations of web services.

In the presented framework we have taken a different approach to defining scenarios. Instead of manually preparing a sequence of test cases, we deliver a formal specification of the tested system in form of XQPN Petri network, a dedicated type of colored Petri network being capable of manipulating XML data. Randomly generated sequence of transitions in XQPN network defines a scenario that is used to invoke calls to an application build on a web service. We believe that such solution is superior to testing predefined scenarios, as automatically generated sequences can be notably longer and due to their random nature provide better coverage.

The paper is organized as follows. Chapter 2 discusses the application of relative correctness methods in testing and verification. In Chapter 3 XQPN networks are introduced. Chapter 4 presents a partial XQPN model of a web service based

application. In Chapter 5 the infrastructure of the tested system is specified; finally Chapter 6 describes the architecture of the framework and its main components.

2 Background

The approach presented in this paper is closely related to previous research on relative correctness of concurrent and real time systems conducted in Institute of Automatics at AGH University of Science and Technology. The statement of relative correctness problem comprises three objects: a verified system, a system specification serving as a criterion and a mapping between those systems. To present this concept without theoretic details we may assume that both the verified system and the criterion are amenable to transition systems $TS = (S, \rightarrow)$, where S is a set of states and $\rightarrow \subseteq S \times S$ is a transition relation. The notion of state is quite general. It can be interpreted as a vector of states of sequential processes and shared variables in a concurrent system, as suggested in [3,4], a marking in a Petri net [6,7] or a state of a data base and system views in case of web application. The mapping may take the form of relation between states of the verified and criterion systems [3,4] or a relation between transitions taking the form of observation function [5,6,7].

This concept is depicted in **Fig. 1**. The transition system at left *Imp* represents the verified system or *implementation*, the system *Spec* is the criterion system (*specification*). Usually, the implementation is more complex than the specification, it may contain transient states not mapped directly to states of specification (eg. 3) and nonobservable actions (τ).

The definitions of relative correctness assume a certain form of synchronous execution of both systems: if the current state of *Imp* is (1) and the corresponding state of *Spec* is (1') then in the next step both systems should synchronize on actions *a* and *a'* and reach states (2) and (2') associated by *sm* relation: In the next step *Imp* should reach the state (4) or (5) via transient state (3) executing nonobservable transition τ and b_1 or b_2, whereas *Spec* synchronously should get to 3' making the transition b'.

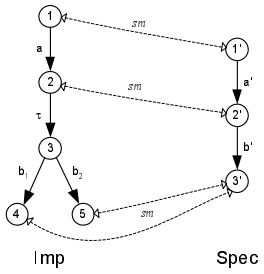

Fig. 1 Illustration of relative correctness problem; two transition systems Imp and Spec linked by *sm* relation between states.

The verification process in model checking approach is driven by transitions in *Imp* that models the behavior of the verified system under the stimuli originating from the environment (**Fig. 2**). After performing an observable transition or reaching an observable state in *Imp* it is checked whether corresponding transition is allowed or corresponding state is a successor of previous state in *Spec*.

The relative correctness definition identifies also special conditions that may occur in *Imp*: a *deadlock* (impossibility to perform any transition at a certain state) and a *divergence* (possible infinite loop of nonobservable transitions). Both conditions have similar effects: execution of *Imp* does not manifest the behavior expected by the analysis of *Spec*.

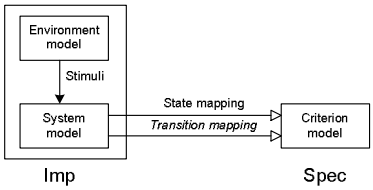

Fig. 2 Control flow in model checking approach

The flow of control in verification process may be reverted (**Fig. 3**). In this case *Spec* is treated as a source of formally defined test cases for *Imp*. Execution of a transition (or sometimes a set of transitions) in *Spec* is mapped onto stimuli from the environment of verified system. After executing corresponding transition in *Imp*, assertions about reached state of *Imp* can be checked. This may imply some rearrangements as regards nonobservable states and transitions.

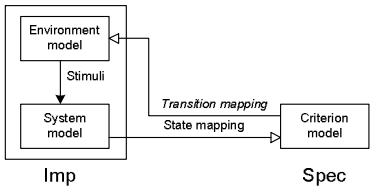

Fig. 3 Control flow in testing

In our framework the subject of verification (*Imp*) is a real web application implemented as a client of a web service. Transitions in verified system correspond to web service operations as defined in WSDL language [8]. We assume that the verified application has standard three tier architecture comprising model, view and controller layers. The state of verified system is an aggregation of data displayed in views, data managed internally by the controller, eg. session state and the data base state. As the common platform to represent system state XML representation is used.

As a specification tool XQPN nets, a dedicated brand of colored Petri nets are used. XQPN nets manipulate XML data stored in places using as expressions constructs of XQuery language [9].

Transitions in XQPN net serving as formal specification of test cases invoke calls to operations of the web service. It is required that if call to the web service succeeds and corresponding transition in XQPN is enabled, they both lead to states that are linked by a state mapping relation (*sm* in **Fig. 1**). In particular, *sm* may be defined as the equality of selected state components for both systems taking the form of XML data.

Using a real web application instead of a model implies some limitations. For a real application it is difficult or even impossible to construct a representation of its whole state space. This might require restoring previous data base state, as well as undoing changes to internally managed data, eg. a session state. Instead, we attempt to cover partially the state space of the system leaving at the moment as an open question the criteria of a good coverage.

3 XQPN Petri nets

XQPN (XQuery Petri Networks) are colored Petri networks allowing manipulation of XML documents stored in places. Marking in XQPN can be treated as a set of XML documents contained in places or a large XML document encompassing them all. Similarly to Colored Petri Nets (CPN) [10], places store sets of distinguishable objects corresponding to various nodes in a document hierarchy. Another common property with CPN networks is the presence of expressions assigned to arcs or guards assigned to transitions. In XQPN they take the form of XQuery [9] language expressions.

3.1 XQuery language

XQuery is the language for querying XML data. We selected XQuery as a 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 to process XML documents in textual form (located in memory or in a file) [11,12] as well as stored in a data base, e.g. IBM DB2 [13] or Oracle [14].

Queries of XQuery return sequences of XML tags (nodes) or atomic values. There is no syntactical difference between a sequence containing one element and its single element. Structure of nodes returned by queries may differ from this appearing in the source document due to the capability of transforming and adding nodes. The other capability offered by XQuery is joining data originating from different sources.

XQuery is a programming language. It allows defining variables and assigning values to them, defining and calling functions. XQuery syntax includes conditional expressions and loops in form of *FLWOR* expression. (*FLWOR* is the abbreviation of first letters of keywords `for`, `let`, `where`, `order by`, `return`). An example of FLWOR expression is:

```
for $x in /bookstore/book
where $x/price>30
```

```
    order by $x/title
    return $x/title
```

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

Both conditional and *FLWOR* expressions could not be treated as structured instructions controlling a program flow, they rather resemble operators, as they return sequences of nodes.

3.1 Description of XQPN nets

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

- P – is a finie set of places,
- T – is a finie set of transitions,
- $A \subseteq P \times T \cup T \times P$ is a set of arcs,
- Q: $A \rightarrow \mathbf{Q}$, is a function, assigning XQuery queries to arcs, we will denote a set of all XQuery queries by **Q**,
- 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 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: $T \rightarrow \mathbf{Q}$ is a function assigning guards to transitions (guards are XQuery queries returning Boolean values).

We will introduce XQPN nets on a small preliminary example presented in **Fig. 4**. The transition *T* is linked with an input arc with a place *store*. The input arc expression `let $x := /*/item[./@id>12]`assigns to the variable `$x` all nodes <`item`> whose value of attribute `id` is greater then 12.

Assuming, that the place *store* contains 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 variable `$x` will be assigned with the node sequence:
`(<item id='13'/> <item id='21'/> <item id='27'/>)`
and then the nodes in `$x` will be removed from the place *store* and added to the place *out* as specified in the output arc expression: `$x`.

Using expressions of XQuery language as arc inscriptions allows a great flexibility in selecting nodes from input places arcs and creating nodes in output places. A

transition can change organization of data before transferring them to an output place. For example using the expression

```
for $y in $x return
<item><id>{data($y/@id)}</id></item>
```

containing *FLWOR* construct for output arc of transition T will extract `id` values from attributes and place them in `<id>` tags.

The exact number of nodes moved by a transition is controlled by a *multiplicity* parameter assigned to input arc, marked as *mul* in the drawing in **Fig. 4**. This parameter corresponds to the weight in PT-nets [15]. Multiplicity is defined as a pair of two numbers [*min*, *max*] specifying bounds. A transition is enabled if input arc expression can evaluate using at least *min* tokens. However, if more then *min* tokens are available, then a maximal number not greater then *max* will be taken. If *min* and *max* are equal, it can be marked on the diagram as a single number. Apart from numbers specification of bounds can contain a symbol of '*' corresponding to *all tokens* (infinity). In most cases values of '1' or '*' are used as multiplicity parameters.

For a net presented in **Fig. 4** setting multiplicity to '*' implies that input arc expression will evaluate to all tokens satisfying `/*/item[./@id>12]`, and `$x` will contain all three nodes. If this parameter is set to '1', then exactly one element will be arbitrary selected among those matching the expression and `$x` will contain only one node. This node will be then removed *from* the place store and added to the place *out*.

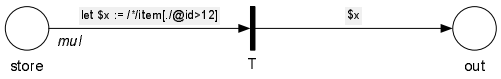

Fig. 4 Transition T moving nodeset specified by input arc expression from the place store to the place out

The net in **Fig. 5** shows other constructs that may appear in XQPN specification. The place *users* is connected with the transition *T* by a *read arc*. Lack of arrow symbolizes that the transition *T* only reads data from its input place, but does not remove them. The other element is a guard that enables the transition only for a those combination of values of `$users` and `$fdata`, where `login` and `password` subnodes are matched.

It should be emphasized that in XQPN all variables appearing in guards and output arc expressions should be bounded by `let` instruction in input arc expressions (e.g: `let $users := /users/user`), In case of additional variables appearing only in output arcs they serve mainly for iteration over nodesets and must be bounded by *FLWOR* expressions. XQPN specification allows a dependency between input variables (values of previously bound variables can be used in expressions defining subsequent variables).

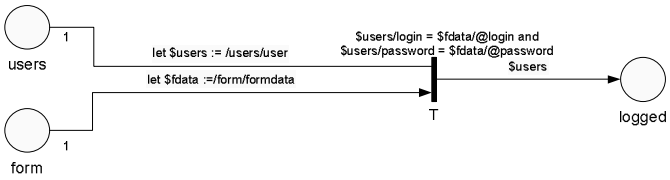

Fig. 5 Example of the transition with read arc and guard

4 Modeling Web Applications with XQPN

We assume that modeled web applications have classical three tier architecture consisting of a *View*, *Controller* and a *Model*. Corresponding layers can be found in XQPN specifications comprising places for *View*, transitions and auxiliary places for *Controller* and places representing data base tables in *Model*. Specifications are usually supplemented by the fourth layer *Test Data* containing places where data corresponding to values entered by users are stored.

We will present this approach on a partial model of an application representing internet shop covering two use cases: Display Goods (**Fig. 6**) and Add to Cart (**Fig. 7**).

In Display Goods use case a logged user with assigned session id selects a category and executes a script of the web application. The place *insession* in Test Data layer contains session numbers surrounded by `<id>` tags. similarly the place incategory contains identifiers of categories.

The place *goods* of the *Model* layer contains the following XML data:

```
<goods>
  <item id='7' category='1'>
    <name>Mouse</name>
    <price>10</price><stock>12</stock>
  </item>
  <item id='10' category='2'>
    <name>Notebook Basic</name>
    <price>299</price><stock>0</stock>
  </item>
  <item id='11' category='2'>
    <name>Notebook</name>
    <price>499</price><stock>2</stock>
  </item>
  <item id='17' category='3'>
    <name>LCD Screen</name>
    <price>399</price><stock>8</stock>
  </item>
</goods>
```

Those data may originate from a data base table and can be assigned to the place before running the test.

The transition *display* reads data from its input places and put them to the place *vgoods* executing the output arc query (partially visible on the diagram):

```
return <vgoods session='{$ses}'>{
for $i in $items
where $i/@category=$cat
return $i } </vgoods>
```

The place *vgoods* belongs to the *View* layer, potentially it can contain multiple `<vgoods>` nodes, each for different category and session combination. This models a situation where different logged users concurrently display goods offered by an internet store. As it can be noticed, a View model in the specification defines the data that are presented by the view, but not the way how they are presented, i.e before they are transformed to HTML representation.

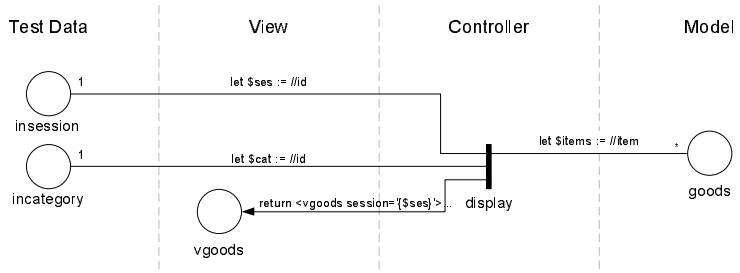

Fig. 6 Model of the use case Display Goods

The net in **Fig. 7** models the use case Add to Cart. It is assumed that the use case starts when a browser displays a list in *vgoods* and the users enters a number of items he wants to buy. The place *inquantity* contains numbers that will be used in test cases placed in `<num>` tags, e.g: 0,1,….10. The script is modeled by three transitions: *addToCart*, *displayCart* and *addToCartFail*.

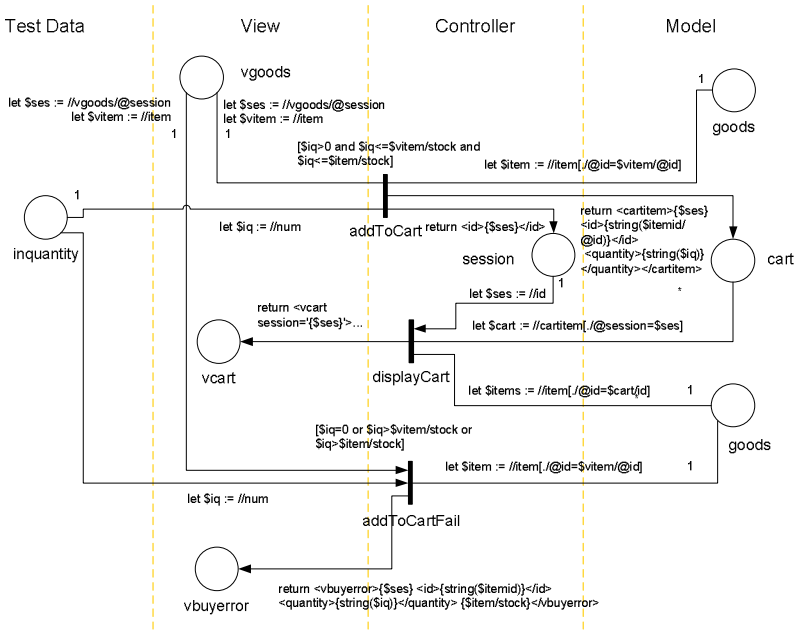

Fig. 7 Model of the use case Add to Cart

The transition *addToCart* is executed if its guard evaluates to true; as it can be seen the guard tests if the selected number of items `$iq` is greater then 0 and less or equal quantities in *vgoods* (list in a view) and *goods* (data base state). Execution of this transition adds an item to the cart (together with the session identifier and the requested quantity) and enables the transition *displayCart*. On the other hand, if the transition *addToCart* can not be executed due to guard evaluation returning false, the transition *addToCartFail* with complementary guard can be executed and produce `vbuyerror` notification.

The transition *displayCart* produces a view of the cart executing an inner join expression at its output arc (partially visible on the diagram):

```
return <vcart session='{$ses}'>
{
for $i in $items
  return <item>{$i/@id}
         {$i/name}
         {
         for $p in $cart
         where $i/@id=$p/id
         return $p/quantity
         }
  </item>
}
</vcart>
```

Input arc expressions for this transitions contain typical dependency between variables that influence the order in which they should be bounded: (`$cart` depends on `$ses` and `$items` depends on `$cart`), thus variables should be bounded by evaluating queries in the order:
`let $ses:=//id`, then
`let $cart := //cartitem[./@session=$ses]` and finally:
`let $items := //item[./@id=$cart/id]`.

5 Tested system

Designing the testing framework we assumed that the tested system is a web service or a web application build entirely on the functionality provided by a web service. A web service can be viewed as a collection of executable code hosted on a web server whose functions are exposed through standard XML protocols. The interface of a web service is defined as a WSDL document [8] specifying operations, their parameters and return values. A web service consumer accesses the web service operations through an object of Proxy class that for majority of implementation platforms can be automatically generated basing on WSDL description of the web service. The Proxy class takes care of communication with a web service and allows calling web service operations as if they were invoked on a local object.

To validate the framework design we have implemented a web service and a web application for an internet shop, whose partial specification was presented in Chapter 4. The tested system (**Fig. 8**) was implemented in C# language on the ASP.NET platform. Its functionality covers typical tasks: logging, searching for goods, adding them to the cart, finalizing the purchase, etc.

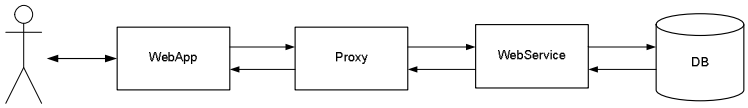

Fig. 8 Components of Web Application basing on a web service

Operations of the tested web service correspond to transitions in *Controller* layer of XQPN specification presented in Chapter 4. It is assumed that the access to the data base is fully wrapped by the web service and there is no direct communication between the *WebApp* and the data base. The *WebApp* client of the web service is responsible for calling web service operations and translating its output to HTML format acceptable by a browser. In some cases it implements parts of business logic, eg. after calling the operation *addToCart*() and receiving the success notification it calls *displayCart*() to show the list of goods selected for the purchase.

6 Architecture of the framework

The testing framework is build of a several components allowing carrying out a set of test cases by synchronously executing XQPN specification and web service operations and then comparing states reached in the specification and the tested system. The components and the data they exchange are shown in **Fig. 9**. Below we provide their short description.

XQPN Editor is a tool for preparing the network specification containing both the network structure as well as marking of selected places in form of XML data. Specification is saved in a file that can be read by *XQPN Executor*.

XQPN Executor is capable of executing transitions in XQPN network. It can fire a specified transition, fire any enabled transition or run a simulation consisting in consecutive execution of randomly selected transitions. At any moment a simulation can be halted and the current marking in XQPN can be examined. *XQPN Executor* provides an interface to be called programmatically; it provides also a console, where a user can issue commands. The console can be used for debugging and validating XQPN specification before applying it in tests as well as for manual execution of test cases.

DB Logger is a tool that examines a data base structure and content and returns its state in XML format. It can also log changes in the data base in more compact differential form. *DB Logger* is used to retrieve initial marking of *Model* places of XQPN network before running test cases and for examining their content during the tests execution.

Proxy (patched) is a standard proxy class generated from WSDL web service description. The generated source code (in C# language) is modified by inserting extra code that logs web service calls and responses. Those data are written to *WS Log*. Output data (*Output XML*) are also used for further comparison with the content of places representing views in XQPN model.

Log Viever is a utility for examining data written in *WS Log*. This can be helpful in analysis of errors detected by running test cases.

Transition2Web Mapper translates information on transition and variables binding to calls of web application build on the tested web service. It uses additional data specifying mapping of transitions in the XQPN model to scripts, mapping of parameter names and preferred call method (POST or GET). The module includes an embedded browser component that manages internal browser data, e.g. cookies, if applicable, and is capable of displaying output of web application after translation to HTML format.

Transition2Proxy Mapper plays similar role in our framework. The main difference is that it directly calls web service methods via the proxy class. It can be used instead of *Transition2Web Mapper* to test a web service rather then application using it.

Comparator is used for testing equality (inclusion in some cases) of marking retrieved from *XQPN Executor* and the data originating from the tested application (Output XML and DB content XML). In general we expect equality the data base state and XML data in places belonging to the *Model* layer in XQPN specification and, and inclusion between Output XML and content of the View data. It is planned to modify the framework at this point by implementing additional functionality that would accumulate and assign to virtual places responses from web service.

Test Driver is responsible for coordination of the automatic execution of test cases. (As it was mentioned earlier it is also possible to execute tests step by step in manual mode using the *Console* provided by *XQPN Executor*.). After initiating the test suite (loading XQPN specification to *XQPN Executor*, retrieving data base state and setting initial marking, loading mapping information) *Test Driver* repetitively execute test cases makes the following steps:

1. Sends a request to the *XQPN Executor* module to fire an enabled transition.
2. The transition with a binding selected in XQPN Executor is transformed to a call of *WebApp* by *Transition2Web Mapper*. The call to *WebApp* propagates through *Proxy* and results in execution corresponding *Web Service* method.
3. Its output corresponding to the *View* layer data is caught at the *Proxy*.
4. Those data are merged with data base state returned by *DB Logger* to form *Test Result*.
5. *TestDriver* invokes *Comparator* module that checks whether *Test Result* data are included in the *Reached Marking* retrieved from *XQPN Executor* module and formulates a verdict: *Test Case Result.*

The whole process stops when an error is detected, there is no enabled transition in XQPN network or a user halts it at a certain moment.

Assembling those components needs certain orchestration. In some cases it is required that several transitions in XQPN are executed in a sequence, e.g. *addToCart* and *displayCart* in **Fig. 7**. We manage with this issue by assigning internal priorities to transitions and defining breakpoints where *Test Result* data should be compared with the marking reached in *XQPN Executor*. That means that in step (1) of a test case execution *XQPN Executor* potentially may fire a few enabled transitions until reaching a breakpoint.

The other issue is the synchronization of transitions in the model and calls to *WebApp*. In general, XQPN specification is executed faster then a tested web application and a maximal time the *TestDriver* waits for *Test Result* must be specified.

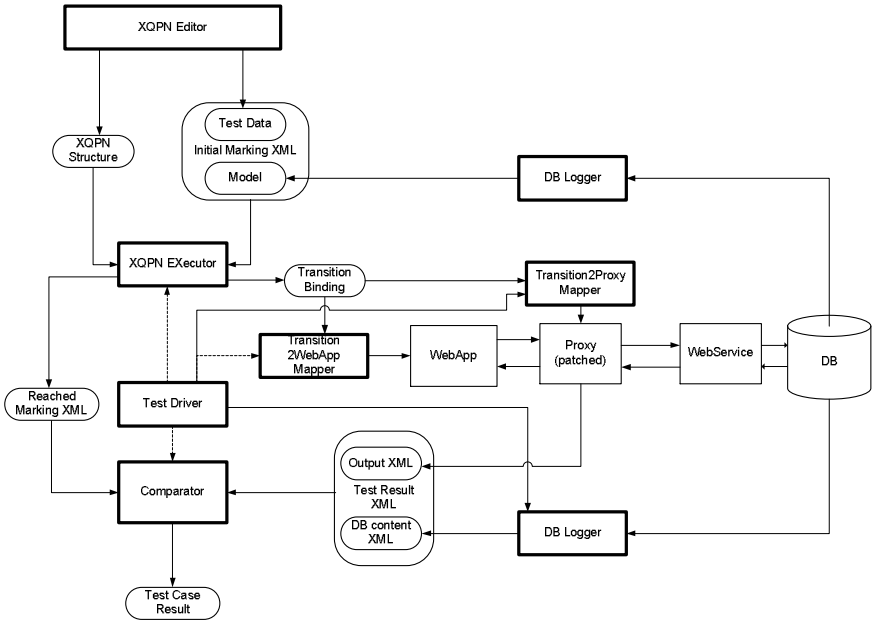

Fig. 9 Architecture of the framework

At present the framework is in prototype phase. Basic modules of the framework were implemented and validated by preparing and running a set of test cases. In general, it is an iterative process, in which a new functionality to the tested application is added, then a XQPN specification defining test cases is prepared and after running the test elements of framework design are fine-tuned.

The framework components were implemented in C# language on a .NET platform. We consider XQPN Executor as the crucial module in the framework. Current implementation of this module uses free COM component Altova XML [11] for XQuery processing. In parallel an alternative executor based on Java Saxon libraries [12] is developed. Tests proved that Altova component is superior while executing queries on small data sets, for larger XML documents containing thousands of nodes Saxon libraries are more efficient and they provide greater flexibility.

Our framework still lacks of tools providing GUI for configuration of tests. For example mapping between XQPN transitions and calls to a web application is specified in form of XML configuration files that should be prepared manually. It is planned to extend the XQPN Editor with a capability of defining such mapping after validating the framework on further examples and establishing detailed requirements.

7 Conclusion

The presented framework for testing web services and web applications based on web services uses the formal specification in form of XQPN Petri net as the source of test

sequences. Tests are driven by transitions in XQPN model. After executing a transition and invoking a corresponding call to the tested application the equivalence between reached model state and data containing the system response and data base state is checked.

This approach offers an advantage that was not mentioned earlier: it is possible to validate the specification before the system is created by simulating and observing its behavior in XQPN Executor. Moreover, if developer selects to implement the system in XQuery environment, queries used in XQPN specification can be ported to implemented application or even the application can be generated automatically from its specification. We plan to examine this solution in the future.

The other direction that is quite straightforward is the verification based on interaction of the tester with the real application. This may be done offline, by examining logs that are already registered or online by intercepting calls to web application and synchronously making transitions in XQPN Executor module.

References

1. eviware software ab, http://www.soapui.org/
2. Web Service Test Forum, www.wstf.org
3. Szmuc, T.: Correctness verification of concurrent systems. In Shriver, B.D.: (eds.): Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, vol. II, Software Track, IEEE Computer Society Press, 1989, 295–304
4. Szmuc, T.: Poprawność współbieżnych systemów oprogramowania, Zeszyty Naukowe AGH, Automatyka, vol. 46, 1989
5. Szwed, P.: Analiza poprawności oprogramowania współbieżnego z wykorzystaniem funkcji obserwacji, praca doktorska, Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki AGH, Kraków 1999
6. Szwed P. Verification of relative correctness of Petri nets, In: 5th Conference on Computer Methods and Systems, Kraków 14-16 november 2005, 295—300
7. Szwed P. Verification of the correctness of Real Time systems specified with timed Petri nets Computer Methods and Systems. In: 5th Conference on Computer Methods and Systems, Kraków 14-16 november 2005, 289—294
8. Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl
9. W3C: XQuery 1.0: An XML Query Language, http://www.w3.org/TR/xquery/
10. Jensen K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use, Vol. I-III, Springer Verlag, 1995/96
11. Altova XML, http://www.altova.com/altovaxml.html
12. Saxonica: XSLT and XQuery Processing, http://www.saxonica.com/
13. Chamberlin, D. Saracco, C.M.:Query DB2 XML data with XQuery, http://www.ibm.com/developerworks/db2/library/techarticle/dm-0604saracco/
14. Oracle XQuery, http://www.oracle.com/technology/tech/xml/xquery/index.html
15. Reisig W.: Petri Nets – An Introduction, EATCS Monographs on Theoretical Computer Science, Volume 4. Springer. 1985