A Lexical and Syntactical Analyzer for the Exporting of QNAP[*]to the Performance Model Interchange Format (PMIF).

Jerònia Rosselló, Catalina M. Lladó, Ramon Puigjaner
Universitat de les Illes Balears
Departament de Matemàtiques i Informàtica
07071, Palma de Mallorca, Spain.
jeronia.rossello@vodafone.es, cllado@uib.es, putxi@uib.es.

Connie U. Smith
Performance Engineering Services
PO Box 2640 Santa Fe
New Mexico, 87504-2640 USA
www.perfeng.com

Abstract

A Performance Model Interchange Format (PMIF) provides a mechanism whereby system model information may be transferred among performance modeling tools. The PMIF allows diverse tools to exchange information and requires only that the importing and exporting tools either support the PMIF or provide an interface that reads/writes model specifications from/to a file. This paper presents the development of the Qnap exporting mechanism to the PMIF. Since access to Qnap internal code is not possible, the only way of exporting Qnap is by translating Qnap input files, for which a lexical and syntax analyzer needs to be developed.

Keywords: Software Performance Engineering, Tool Interoperability, Performance Model, XML, Queueing Network Model, Interchange Format, Compiler Techniques.

Resumen

Un Formato de Intercambio de Modelos de Rendimiento (PMIF, *Performance Model Interchange Format*) proporciona un mecanismo para el intercambio de información de modelos de rendimiento entre diferentes herramientas de modelado. El PMIF permite que varias herramientas intercambien información solo requiriendo que las herramientas que exportan e importan modelos soporten el PMIF o dispongan de una interfaz que lee/escribe las especificaciones en PMIF desde/a un fichero. Este artículo presenta el desarrollo del mecanismo de exportación de Qnap a PMIF. Dado que el acceso al código interno de la herramienta Qnap no resulta viable, la única manera de exportar Qnap es traduciendo los ficheros de entrada de Qnap, por lo cual se necesita desarrollar un analizador léxico y sintáctico.

Palabras claves: Ingeniería de Rendimiento de Software, Interoperabilidad de Herramientas, Modelos de Rendimiento, XML, Redes de Colas, Formatos de Intercambio, Técnicas de Compilación.

1 INTRODUCTION

(PMIF) [10, 7] is a common representation for Queuing Network Model (QNM) data that can be used to move models among modeling tools. A user of several tools that support the format can create a model in one tool, and later move the model to other tools for further work without the need to laboriously translate from one tool's model representation to the other, and the need to validate the resulting specification. Tools that

[*]Qnap is a commercial tool developed by Simulog [5] for queueing networks modelling.

support the PMIF format only need to implement the export and import mechanisms to the PMIF rather than implement customized exports and imports for every other tool that they want to share information with[1].

PMIF users could, for example, compare solutions from multiple tools; create input specifications in PMIF or in a familiar tool rather than learn the interface to multiple tools; migrate a model to temporarily use another tool to study more detailed models; and create software performance models to study architecture and design trade-offs, then use another tool to study details of the computer system.

Depending on the modeling tool and the availability of its source code, its exporting and importing mechanisms to/from PMIF can vary from a simple implementation to a huge engineering project. In [7] a prototype is described in which the exporting tool is *SPE·ED* and the importing tool is Qnap [2, 6] . In this case, both importing and exporting mechanisms were quite straightforward. On the contrary, when thinking about the exporting mechanism from Qnap to PMIF, one finds out that unfortunately it is not so simple.

Due to the fact that we do not have access to Qnap's internal code, the Qnap to PMIF export mechanism can only be achieved by using the Qnap input file that defines the QNM in Qnap's format. Such a file needs to be carefully analyzed in order to be translated. As a consequence, the translation consists of a much more laborious process (compared to for instance the use of the Document Object Model (DOM) in the export mechanism from *SPE·ED* [8]) that needs to be carefully studied and designed. This process includes the use of compiler techniques such as lexical and syntactical analysis (see the Appendix for an overview of these techniques). This paper presents the analysis, design and implementation of such a process.

The rest of the paper is organized as follows. Section 2 gives an overview of the PMIF format, describing its main components and how they relate. Following, Section 3 considers the most importat issues that had to be addressed in the design of the Qnap to PMIF exporting mechanism while Section 4 describes its implementation. Section 5 shows an example of the application of the translation process step by step. Finally, Section 6 reports some future work and conclusions. The Appendix provides an overview of compiler techniques and tools for lexical analysis and syntactic analysis.

2 PMIF OVERVIEW

PMIF was first defined using an EIA/CDIF (Electronic Industries Association/CASE Data Interchange Format) paradigm that calls for defining the information requirements for a Queueing Network Model (QNM) with a meta-model [9, 10], that is, it is a model of the information that goes into constructing a QNM. A transfer format was then created from the meta-model and used to exchange information. The PMIF allows diverse tools to exchange information and requires only that the importing and exporting tools either support the PMIF or provide an interface that reads/writes model specifications from/to a file.

A new version of the meta-model has been recently defined together with a new PMIF specification (PMIF 2.0) [7, 8], which is implemented as an XML (Extensible Markup Language) schema [12].

The diagram of the XML Schema for PMIF 2.0 is in Figure 1. The diagram shows that a *QueueingNetworkModel* is composed of one or more *Nodes*, and one or more *Workloads*. A *Server* provides service for one or more *Workloads*. A *Workload* represents a collection of transactions or jobs that make similar *ServiceRequests* from *Servers*. There are two types of *Workloads*: *OpenWorkload* and *ClosedWorkload*.

A *ServiceRequest* specifies the average *TimeService*, *DemandService* or *WorkUnitService* for each *Workload* that visits the *Server*. A *TimeServiceRequest* specifies the average service time and number of visits. A *DemandServiceRequest* specifies the average service demand (service time x number of visits). A *WorkUnitServiceRequest* specifies the average number of visits requested by each *Workload* that visits a *WorkUnitServer*. Upon completion of the *ServiceRequest*, the *Workload Transits* to other *Nodes* with a specified probability.

More detailed information and the PMIF Schema definition can be found in [7, 8].

In addition, a prototype was also developed, in which the exporting tool is *SPE·ED* and the importing tool is Qnap. *SPE·ED* uses the Document Object Model (DOM) [12] to export the pmif.xml. On the other hand, since the access to Qnap source code was not provided, an XSLT specification that transforms a pmif.xml file into a file that is read and executed by Qnap was implemented.

Moreover, in [3] the design and implementation of a PMIF Web service for the modeling tool Qnap is presented as a further step of the transformation from PMIF to Qnap.

[1]It is possible to use the PMIF without an explicitly coded import and export function as long as the tool provides a file input/output capability.

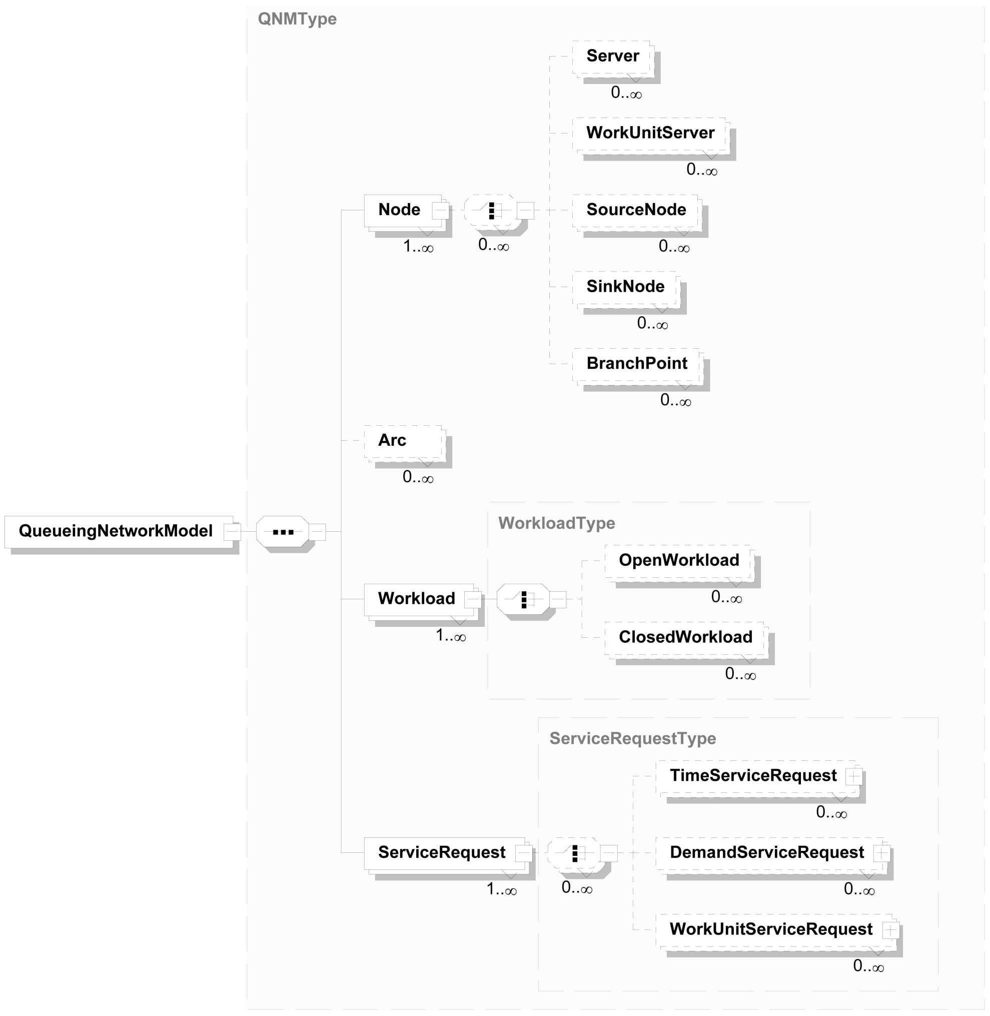

Figure 1: XML Schema for PMIF 2.0

3 QNAP TO PMIF EXPORTATION MECHANISM

As previously stated, we need to develop a mechanism to export a Qnap model to the PMIF format. Due to the fact that we do not have access to Qnap internal code we can only do it through the design and implementation of a mechanism to transform Qnap input files into files in PMIF format. This transformation can be achieved by means of a lexical analyzer and a syntax analyzer.

A lexical analyzer needs to know the regular expressions that form the language and a syntax analyzer needs to know the grammar that the language uses (see the Appendix for more detailed information). In our case, the regular expressions and the grammar are a subset of those that form the Qnap language [6]. The description and justification of them is detailed in subsection 3.1.

The syntax and lexical analysis of the Qnap input file gives the information needed to generate the PMIF file. However, it is convenient to do the analysis of the Qnap file and the generation of the PMIF file in two separate phases. Subsection 3.2 justifies the need for two phases and explains their design and interaction.

3.1 Qnap Modified Regular Expressions and Grammar

The Qnap language has its own reserved words, types and grammar [6]. The reserved words and types can be detected by using regular expressions, so a Qnap compiler uses a specific set of regular expressions. However, Qnap is a modelling tool that allows the user to solve more cases and in more ways, than the PMIF covers. Therefore, the Qnap grammar and its regular expressions have been reduced so that only what is admissible by the PMIF specification is admitted.

The regular expressions admitted are:

- The ones with numeric format, i.e. with float or exponential notation, which follow:
  NUMBER = $[1-9][0-9]*$
  FLOAT =$[0-9]*\backslash\cdot[0-9]+$
  IDENT = $([a-z])([0-9]|[a-z])*$
  EXPONEG = $[0-9]*\backslash\cdot[0-9]+\backslash E\backslash-[0-9]*$
  EXPOPOS = $[0-9]*\backslash\cdot[0-9]+\backslash E\backslash+[0-9]*$
  EXPONEGCURTA = $[0-9]*\backslash E\backslash-[0-9]*$
  EXPOPOSCURTA = $[0-9]*\backslash E\backslash+[0-9]*$
- Specific Tokens used in the Qnap language: /STATION/, /DECLARE/, /END/, NAME, =, TYPE, SERVICE, EXP, (,), ;, , , SERVER, SOURCE, SINGLE, MULTIPLE, INFINITE, INTEGER, REAL, QUEUE, CLASS, STEP, UNTIL, FIFO, PS, SCHED, INIT, TRANSIT, OUT.

The following are some restrictions that the modified grammar imposes:

- Everything that in Qnap is specific to simulation programs (for instance, the *customer* object type or the *flag* object) does not need to be included in our grammar since the PMIF currently does not include those.
- If a Qnap model has only one class of clients, the class declaration is optional. In the new grammar, at least one class needs to be defined. Moreover, all the classes that are going to be used in the model need to be declared at the beginning and before any node (station) is declared.
- Qnap also allows the change of class for a client that goes from one node to another which is not allowed in the PMIF or the new grammar.
- Qnap permits the definition of two nodes as initial nodes for the same class. This is not covered either in our modified grammar.

The resulting modified grammar can be seen as a sub-grammar of the Qnap grammar and it is shown below:

program ⟶ GeneralBlocList /END/
GeneralBlocList ⟶ BlocList
BlocList ⟶ ComandDeclare BlocListStation
BlocListStation ⟶ BlocListStation ComandStation | ComandStation
ComandDeclare ⟶ /DECLARE/ VariableDeclareList
VariableDeclareList ⟶ VariableDeclareList PartVariableDeclare
| PartVariableDeclare
PartVariableDeclare ⟶ CLASS IdentifierClassList ;
| QUEUE IdentifierQueueList ;
| INTEGER IdentifierValueList ;
| REAL IdentifierValueList ;
IdentifierQueueList ⟶ IdentifierQueueList , QueueSize
| QueueSize
QueueSize ⟶ identifier | identifier (number)
IdentifierClassList ⟶ IdentifierClassList , identifier
| identifier
IdentifierValueList ⟶ IdentifierValueList , identifier ValueAssignment
| identifier ValueAssignment
ValueAssignment ⟶ = number | = double | void
ComandStation ⟶ /STATION/ ParameterName ParametersStationList
ParameterName ⟶ NAME = llistaIdentificadorsStation , identifier

```
| IdentifiersStationList , identifier ( number )
| IdentifiersStationList , identifier ( number STEP number UNTIL number )
| identifier
| identifier ( number )
| identifier ( number STEP number UNTIL number )
ParametersStationList ⟶ ParametersStationList ParameterStation
| ParameterStation
ParameterStation ⟶ ParameterType
| ParameterService
| ParemeterSched
| ParameterInit
| ParameterTransit
ParameterType ⟶ TYPE = StationType ;
StationType ⟶ SERVER | SERVER , multiple | SOURCE | multiple
multiple ⟶ SINGLE | MULTIPLE ( number ) | INFINITE
ParameterService ⟶ SERVICE ClassList = Distribution ;
Distribution ⟶ EXP ( number ) | EXP ( double )
ClassList ⟶ ( IdentifierList ) | void
IdentifierList ⟶ IdentifierList , identifier | identifier
ParameterSched ⟶ SCHED = SchedType ;
SchedType ⟶ FIFO | PS
ParameterInit ⟶ INIT ClassList = number ;
ParameterTransit ⟶ TRANSIT ClassList = routing ;
routing ⟶ queue | PairList final
PairList ⟶ PairList , pair | pair
pair ⟶ queue , number | queue , double
final ⟶ , queue | void
queue ⟶ identifier | identifier ( number ) | OUT
```

3.2 Modular Design

The lexical analyzer can carry out some actions depending on the tokens found (see the Appendix for a detailed explanation). These actions could create a DOM object and afterwards generate the PMIF output file from it. However, the structure of the QNAP language combined with the characteristics of the DOM would make the code included in the analyzer long and complicated (some of these are detailed below). This is the reason why we developed two modules that carry out the transformation in two phases. The first one applies the syntax and lexical analysis to the Qnap input file and generates memory structures containing the information necessary for a second phase which is in charge of generating the XML file. The process is shown in Figure 2, in which the syntax analyzer (SA) asks the lexical analyzer (LA) for a token (*tk?*) and the lexical analyzer responds with a token. $Obj1 \cdots ObjN$ are the memory structures or objects. When the first phase finishes, it has obtained the characteristics of the model and saved them into the memory structures. The second phase works on these structures to generate the PMIF XML output.

The main Qnap characteristics that make the generation of memory structures convenient while carrying out the analysis are:

- Qnap has default values for most of the model parameters. The memory structures are initialized with those default values.
- Qnap allows the update of those values at any point in the model specification and as many times as wanted (and it is very usual to find this sort of programming in Qnap programs). The last one found (the one that appears latest in the input file) is the value used. So, for instance, a node (STATION in Qnap) can be redefined as many times as wanted and the last definition (seeing the input file sequentially) is the valid one. Using the intermediate memory structures, these are directly modified every time a modification of the model definition is found.

Qnap models often have redefinitions of nodes and attributes in the input file, thus the overhead is higher for updating a DOM document tree than it is for the intermediate memory structures.

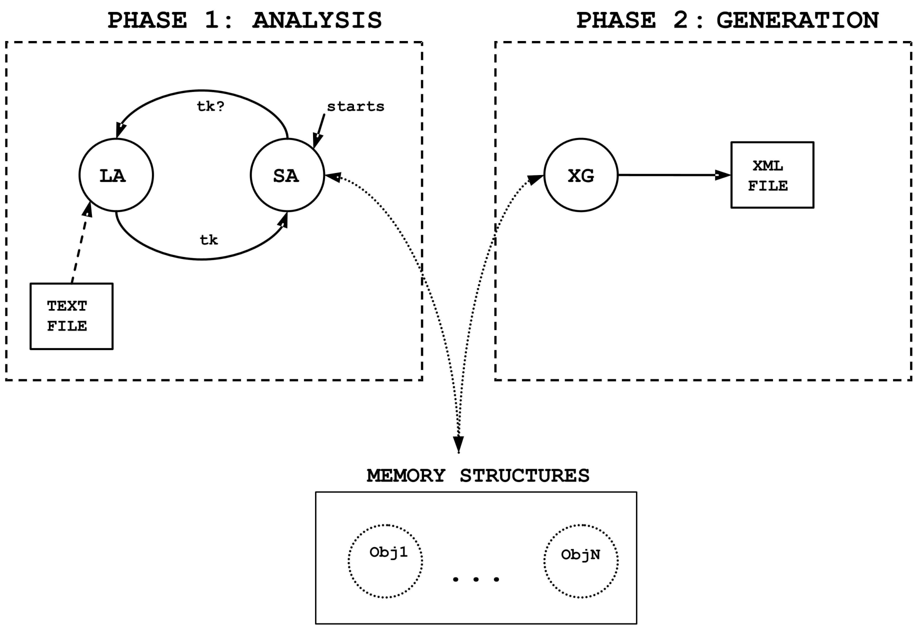

Figure 2: 2-Phases Design. LA: lexical analyzer, SA: syntax analyzer, XG: XML-Generator, tk: token

4 IMPLEMENTATION OF THE MECHANISM

The implementation of the two phases described in the previous section starts with the use of JLex [11] and JavaCup [4] tools (described in the Appendix) to create the lexical analyzer and syntax analyzer respectively. JLex uses the file with the Qnap regular expressions as the input file. JavaCup uses the grammar described in subsection 3.1 as input plus the required actions needed to create the memory structures that contain the information of the model.

The lexical analyzer generated by JLex carries out the subsequent actions in this specific order:

- Groups the characters of the Qnap input file into tokens and passes them to the syntax analyzer. These tokens are manipulated with the Symbol class, so the objects sent to the syntax analyzer are Symbols (java specification of tokens). Each token or Symbol is composed of the type of the token (or name) and the value of the token.

- Detection of the regular expressions subset corresponding to the ones with numeric format (detailed in section 3) and detection of identifiers. The value assigned to the numeric tokens (value of Symbol) is the value of the number, and the name of the token is *Double* or *Integer.* In the case of identifiers, the value of Symbol is the name of the string and the type is *identifier.*

- Detection of the rest of the tokens. In this case, the name (type) of the Symbol is similar to the name of the token detected and the value is the corresponding value found in the Qnap input file.

The syntax analyzer generated is able to check whether the input adheres to the grammar described in subsection 3.1. Its tasks can be briefly described as the following:

- Queries the lexical analyzer for tokens, and detects whether the sequence of received tokens is valid against the grammar.

- While the grammar productions are being reduced, the memory structures (described below) are filled with the corresponding values.

- Once all the grammar is reduced, the filling of the memory structures is finished and the phase two (XML Generation) can start.

Subsequently, the memory structures are described in detail. Three tables form these structures: the Nodes Table Information (NTI), the Routing Table Information (RTI) and the Workload Table Information (WTI).

The NTI saves the characteristics of the nodes in the system (STATIONs in Qnap). This table has as many rows as the system nodes and it has $3 + N$ columns, where N is the number of workloads (CLASSes in Qnap) defined in the model. The first column contains the type of the node, the second column the scheduling policy and the third column the quantity of servers. Finally, the following N columns contain the service time for each specific workload. This table is implemented as a 2D-vector as shown in Table 1.

	Type	Sched	Quantity	Workload1	Workload2	...	WorkloadM
Node1							
Node2							
...							
NodeN							

Table 1: Nodes Table Information

The RTIs save the routing probabilities (TRANSITs in Qnap) of all the nodes of the model. There is a RTI for each workload. Given a workload, the RTI values represent the routing probabilities between all the pair of nodes of the system. The set of RTIs is implemented as a 3D-vector. The 2-D part corresponding to 1 workload is shown in Table 2.

Workload1	Node1	Node2	...	NodeN
Node1				
Node2				
...				
NodeN				

Table 2: Routing Table Information

The WTI saves the workload characteristics for open and closed workloads. For closed workloads only the first part of the table will be filled. On the contrary, for open workloads only the second part of the table will be used. Table 3 shows these parts, implemented as a 2D-vector.

	ClosedWorkload			OpenWorkload		
	NumberOfJobs	ThinkDevice	ThinkTime	ArrivalRate	ArrivalsAt	DepartsAt
Workload1						
Workload2						
...						
WorkloadN						

Table 3: Workload Table Information

The second phase (see Fig. 2) or XML generation is in charge of generating a DOM structure by querying the three previously described tables, and posteriorly, creating the PMIF XML document. This functionality involves 3 well defined steps:

- Initialization of the DOM structure with the basis: A node called *QueueingNetworkModel*, including two mandatory child nodes: *Node* and *Workload* nodes [7, 8]. So that, it includes:
  $< QueueingNetworkModel >$
  $< Node/ >$
  $< Workload/ >$
  $< /QueueingNetworkModel >$

- Implementation of the four methods which are in charge of creating nodes, workloads, requests and arcs (described in Table 4). All of them get data from the table structures described above and add the nodes with the required attributes and children to the DOM structure.

- Generation of the PMIF XML file from the DOM structure.

NodesGeneration	Adds as many children to $< Node/ >$ as nodes are in the NTI each with its attributes. Children can be *SourceNode*, *SinkNode*, *WorkUnitServer* or *Server*
WorkloadsCreation	Adds as many children to $< Workload/ >$ as workloads are in the WTI. A child can be *ClosedWorkload* or *OpenWorkload*, each one of them with their own attributes
ArcsCreation	Creates as many *Arc* nodes as TRANSITs are in the RTI's
RequestCreation	Creates *ServiceRequest* which will contain *TimeServiceReq* nodes by querying the NTI and the RTI

Table 4: Description of the Methods used to generate the DOM structure

5 USE CASE

This section describes the Qnap to PMIF exporting mechanism step by step. The example used is based on the ATM model from [10]. The Qnap code (input file to be transformed) corresponding to this model is shown below:

```
/DECLARE/    QUEUE cpu;
             QUEUE atm, disks;
             QUEUE sourcen1, sourcen2 ;
             CLASS withdraw, getbala;
             REAL twithdra, tgetbal;

/STATION/    NAME= cpu;
             SCHED = PS;

/STATION/    NAME = atm;
             SERVICE = EXP(1);
             TYPE = INFINITE;

/STATION/    NAME = disks;
             SERVICE = EXP(0. 05);
             SCHED = FIFO;

/STATION/    NAME = sourcen1;
             TYPE= SOURCE;
             SERVICE= EXP(1);
             TRANSIT(withdraw)= cpu, 1;

/STATION/    NAME = sourcen2;
             TYPE= SOURCE;
             SERVICE= EXP(1);
             TRANSIT(getbala) = cpu, 1;

/STATION/    NAME = atm;
             TRANSIT(withdraw) = cpu, 1 ;

/STATION/    NAME = disks;
             TRANSIT(withdraw) = cpu, 1 ;

/STATION/    NAME = atm;
             TRANSIT(getbala) = cpu, 1 ;

/STATION/    NAME = disks;
             TRANSIT(getbala) = cpu, 1 ;

/STATION/    NAME = cpu;
             SERVICE(withdraw) = EXP(0.000315);
             TRANSIT(withdraw) = atm, 0.55, disks, 0.4, OUT , 0.05 ;
```

```
/STATION/     NAME = cpu;
              SERVICE(getbala) = EXP(0.00025);
              TRANSIT(getbala) = atm, 0.6, disks, 0.3, OUT , 0.1 ;

/CONTROL/
/END/
```

The lexical analyzer detects the tokens and the parser proves that the token sequence complies with the grammar and at the same time generates the tables (see section 4) and fills them with the values found. For this example the tables are filled as shown below:

	Type	Sched	Quantity	Withdraw	Getbala
Sourcen1	Source	PS	1	1.0	0
Sourcen2	Source	PS	1	0	1.0
Disks	Single	FCFS	1	0.05	0.05
Atm	Infinite	IS	1	1.0	1.0
Cpu	Single	PS	1	3.15E-4	2.5E-4
OUT	Single	PS	1		

Table 5: Nodes Table Information for the ATM example

Withdraw	Sourcen1	Sourcen2	Disks	Atm	Cpu	OUT
Sourcen1	0.0	0.0	0.0	0.0	1.0	0.0
Sourcen2	0.0	0.0	0.0	0.0	0.0	0.0
Disks	0.0	0.0	0.0	0.0	1.0	0.0
Atm	0.0	0.0	0.0	0.0	1.0	0.0
Cpu	0.0	0.0	0.4	0.55	0.0	0.05
OUT	0.0	0.0	0.0	0.0	0.0	0.0

Table 6: Routing Table Information for the ATM example and the Withdraw workload

Getbala	Sourcen1	Sourcen2	Disks	Atm	Cpu	OUT
Sourcen1	0.0	0.0	0.0	0.0	0.0	0.0
Sourcen2	0.0	0.0	0.0	0.0	1.0	0.0
Disks	0.0	0.0	0.0	0.0	1.0	0.0
Atm	0.0	0.0	0.0	0.0	1.0	0.0
Cpu	0.0	0.0	0.3	0.6	0.0	0.1
OUT	0.0	0.0	0.0	0.0	0.0	0.0

Table 7: Routing Table Information for the ATM example and the Getbala workload

Both Tables 6 and 7 show one row completely empty, *Source1* and *Source2* respectively. This is normal considering the Qnap restriction of only allowing 1 class (or workload) for source node.

When the parser has finished and the structures are filled the second phase starts with the generation of the DOM structure by querying the three previously shown tables. The XML file in PMIF format is afterwards created resulting with the following:

```
<?xmlversion = ”1.0”encoding = ”UTF − 8”? >
< QueueingNetworkResultsxmlns : xsi = ”http : //www.w3.org/2001/XMLSchema − isntance”
xsi : noNamespaceSchemaLocation = ”http : //www.perfeng.com/pmif/pmifschema.xsd” Name = ”ATM” >
< Node >
< SourceNode Name = ”sourcen2”/ >
< SourceNode Name = ”sourcen1”/ >
< WorkUnitServer Name = ”disks” Quantity = ”1” SchedulingPolicy = ”FCFS” ServiceTime = ”0.05”/ >
< WorkUnitServer Name = ”atm” Quantity = ”1.0” SchedulingPolicy = ”IS” ServiceTime = ”1.0”/ >
< WorkUnitServer Name = ”cpu” Quantity = ”1” SchedulingPolicy = ”PS” ServiceTime = ”3.15E − 4”/ >
< SinkNode Name = ”OUT”/ >
< /Node >
< Workload >
< OpenWorkload WorkloadName = ”withdraw” ArrivalRate = ”1.0” ArrivesAt = ”sourcen1” DepartsAt = ”OUT” >
< Transit To = ”cpu” Probability = ”1.0”/ >
< /OpenWorkload >
< OpenWorkload WorkloadName = ”getbala” ArrivalRate = ”1.0” ArrivesAt = ”sourcen2” DepartsAt = ”OUT” >
< Transit To = ”cpu” Probability = ”1.0”/ >
```

	ClosedWorkload			OpenWorkload		
	NumberOfJobs	ThinkDevice	ThinkTime	ArrivalRate	ArrivalsAt	DepartsAt
Withdraw				1.0	Sourcen1	OUT
Getbala				1.0	Sourcen2	OUT

Table 8: Workload Table Informationle for the ATM example

```
< /OpenWorkload >
< /Workload >
< Arc FromNode = "sourcen2" ToNode = "cpu"/ >
< Arc FromNode = "sourcen1" ToNode = "cpu"/ >
< Arc FromNode = "disks" ToNode = "cpu"/ >
< Arc FromNode = "atm" ToNode = "cpu"/ >
< Arc FromNode = "cpu" ToNode = "disks"/ >
< Arc FromNode = "cpu" ToNode = "atm"/ >
< Arc FromNode = "cpu" ToNode = "OUT"/ >
< ServiceRequest >
< TimeServiceRequest WorkloadName = "getbala" ServerID = "sourcen2" ServiceTime = "1.0" >
< Transit To = "cpu" Probability = "1.0"/ >
< /TimeServiceRequest >
< TimeServiceRequest WorkloadName = "withdraw" ServerID = "sourcen1" ServiceTime = "1.0" >
< Transit To = "cpu" Probability = "1.0"/ >
< /TimeServiceRequest >
< TimeServiceRequest WorkloadName = "withdraw" ServerID = "disks" ServiceTime = "0.05" >
< Transit To = "cpu" Probability = "1.0"/ >
< /TimeServiceRequest >
< TimeServiceRequest WorkloadName = "getbala" ServerID = "disks" ServiceTime = "0.05" >
< Transit To = "cpu" Probability = "1.0"/ >
< /TimeServiceRequest >
< TimeServiceRequest WorkloadName = "withdraw" ServerID = "atm" ServiceTime = "1.0" >
< Transit To = "cpu" Probability = "1.0"/ >
< /TimeServiceRequest >
< TimeServiceRequest WorkloadName = "getbala" ServerID = "atm" ServiceTime = "1.0" >
< Transit To = "cpu" Probability = "1.0"/ >
< /TimeServiceRequest >
< TimeServiceRequest WorkloadName = "withdraw" ServerID = "cpu" ServiceTime = "3.15E − 4" >
< Transit To = "disks" Probability = "0.4"/ >
< Transit To = "atm" Probability = "0.55"/ >
< Transit To = "OUT" Probability = "0.05"/ >
< /TimeServiceRequest >
< TimeServiceRequest WorkloadName = "getbala" ServerID = "cpu" ServiceTime = "2.5E − 4" >
< Transit To = "disks" Probability = "0.3"/ >
< Transit To = "atm" Probability = "0.6"/ >
< Transit To = "OUT" Probability = "0.1"/ >
< /TimeServiceRequest >
< /ServiceRequest >
< /QueueingNetworkResults >
```

This file can be easily validated against the PMIF 2.0 XML Schema (the complete schema definition may be seen at http://www.perfeng.com/pmif/pmifschema.xsd)

6 CONCLUSIONS AND FUTURE WORK

A Performance Model Interchange Format (PMIF) provides a mechanism whereby system model information can be interchanged between different performance modeling tools. The exporting and importing tools can either support the PMIF or provide an interface to read/write model specifications from/to a PMIF file. This paper describes the non-trivial process carried out in order to implement the Qnap exporting mechanism to PMIF. This process requires the lexical and syntactical analysis of the Qnap input file in order to make possible the translation to the PMIF format.

A separate research project is also developing the complete meta-model for performance results and the XML schema for it. Future work is then the development of the Qnap exporting mechanism for the results. Moreover, the consequences of other possible PMIF modifications (as for instance adding solving instructions) on the the exporting mechanism should also be addressed in the near future.

So far, we have developed PMIF export mechanism prototypes for Qnap and for *SPE·ED* and the importing mechanism for Qnap (details of the last two can be found in [8]). We would like to use this experience and the prototypes to develop some common routines (i.e. an API) that would make it easier for a tool developer to implement an internal interface to PMIF.

Appendix: Compiler Techniques and Tools

A programming language can be defined by describing the aspect of its programs (the *syntax* of the language) and the meaning of its programs (the *semantics* of the language) [1]. In order to represent the syntax of the language, a well known notation is used called context-free grammars or BNF (for Backus-Naur Form). Besides specifying the syntax of a language, a context-free grammar can be used to help guide the translation of programs, since a grammar naturally describes the hierarchic structure of most constructions of programming languages.

In the lexical analysis (or scanning) the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning. Each token is treated as a unique entity. The blanks separating the the characters of these tokens would normally be eliminated during lexical analysis.

In the syntax analysis (or parsing) the characters or tokens are grouped hierarchically into nested collections with collective meaning. In other words, syntax analysis involves grouping the tokens of the source program into grammatical phrases that are used to synthesize output. Usually, the grammatical phrases of a source program are represented by a parse tree. A parse tree describes the syntactic structure of the input.

For the transformation of a Qnap file into a file in PMIF format we need a lexical analyzer and a syntax analyzer. There exist software tools that make it possible to easily create such analyzers. Some examples are *JLex* [11] and *JavaCup* [4] which use Java technologies.

JLex makes possible the generation of lexical analyzers. The user only needs to write a specification file (.lex) where the regular expressions that the analyzer needs to find are defined. Using this file, *JLex* quickly builds a Java application (a lexical analyzer) that can analyze files composed of chains of characters.

The resulting lexical analyzer searches the file to find strings that fit the regular expressions contained in the specification file (.lex). The regular expressions are defined in the .lex file together with the action to carry out when these expressions are found by the lexical analyzer. A regular expression represents a set of chains and the action is a set of orders that will be executed when the lexical analyzer finds a chain that fits the regular expression.

JavaCUP is an utility for generating syntax analyzers. The user needs to provide *JavaCUP* with an input file (.cup) where the grammar is specified, and then *JavaCUP* generates a java syntax analyzer. This analyzer can check whether an input text file adheres to the specified grammar.

The grammar specification file (.cup) includes the grammar specified in terms of terminal symbols, non terminal symbols, and a set of production rules. It can also include java code for executing specific actions during the analysis.

The syntax analyzer and the lexical analyzer work together. The syntax analyzer asks the lexical analyzer for the tokens (terminal symbols) it encounters.

Fig. 3 summarizes this process.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles Techniques and Tools.* Addison-Wesley, 1986.

[2] M. Potier, D.; Veran. Qnap2: A portable environment for queueing systems modelling. In D. Potier, editor, *I International Conference on Modeling Techniques and Tools for Performance Analysis (Paris, May 1984)*, pages 25–63. North Holland, May 1985.

[3] J. Rossello, C.M. Llado, R. Puigjaner, and C.U. Smith. A web service for solving queueing networks models using pmif. Tecnical report. To be plublished in Proceedings of the Fith International Workshop of Software and Performance, July 2005.

[4] C. Scott Ananian Scott Hudson, Frank Flannery. Cup parser generator for java. www.cs.princeton.edu/ appel/modern/java/CUP/.

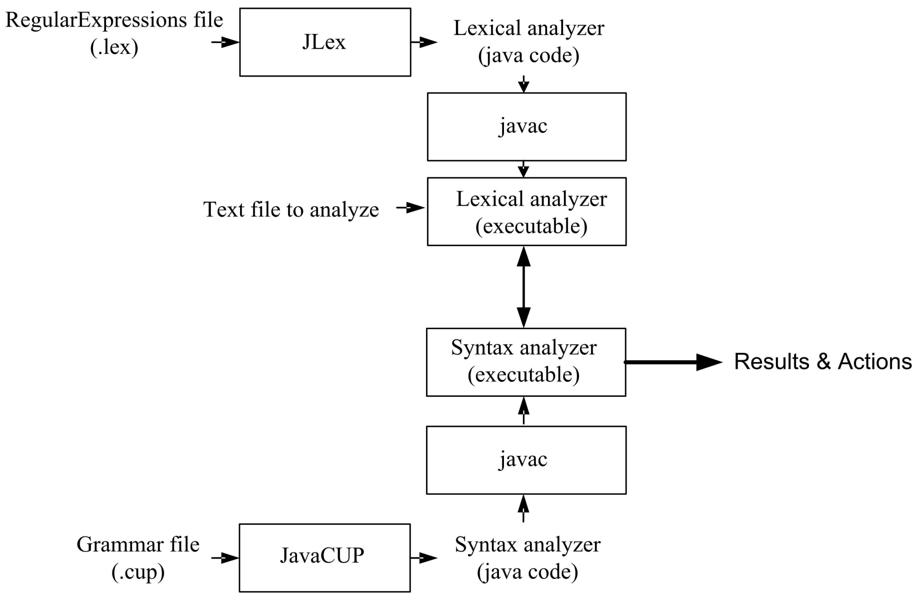
RegularExpressions file (.lex)
JLex
Lexical analyzer (java code)
javac
Text file to analyze
Lexical analyzer (executable)
Syntax analyzer (executable)
Results & Actions
javac
Grammar file (.cup)
JavaCUP
Syntax analyzer (java code)

Figure 3: Lexical and Syntax Analysis Process

[5] Simulog. Qnap modelling tool. www.simulog.fr.

[6] Simulog. Modline 2.0 qnap2 9.3: Reference manual, 1996.

[7] C.U. Smith and C.M. Llado. Performance model interchange format (pmif 2.0): Xml definition and implementation. In *Proc. of the First International Conference on the Quantitative Evaluation of Systems*, pages 38–47, September 2004.

[8] C.U. Smith and C.M. Llado. Performance model interchange format (pmif 2.0): Xml definition and implementation. tecnical report. www.perfeng.com/paperndx.htm, April 2004.

[9] C.U. Smith and L.G. Williams. Panel presentation: A performance model interchange format. In *Proc. of the International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, 1995.

[10] C.U. Smith and L.G. Williams. A performance model inter-change format. *Journal of Systems and Software*, 49(1), 1999.

[11] Elliot Berk.Princeton University. Jlex: A lexical analyzer generator for java(tm). www.cs.princeton.edu/ appel/modern/java/JLex/.

[12] w3c. World wide web consortium. www.w3c.org, 2001.