TOWARDS A METAMODEL FOR EXTENDED QUEUING NETWORKS

Bourouis, A.* & Belattar, B.**

* Department of Mathematics and Computer Science, Faculty of Sciences,
University of Oum El Bouaghi, 4000 Oum El Bouaghi, Algeria

** Department of Computer Science, Faculty of Sciences,
University of Batna, 5000 Batna, Algeria

E-Mail: a.bourouis@univ-ueb.dz; brahim.belattar@univ-batna.dz

Abstract

This paper focuses on the development of EQNM²L which is an extended queuing modelling and markup language. We discuss the DSML metamodel and its XML-based exchange format which represent the cornerstone of the development process. EQNM²L enhances interoperability between a wide range of analytical solvers and simulation tools dealing with systems performance evaluation and based on the extended queuing theory. Furthermore, the Model Driven Engineering approach allows automatic generation of modelling environments and simulation/analytical codes which improve productivity and quality. Our aim is to induce discussion on and contributions for elaborating the whole metamodel and providing a starting point for the development of a standard inter-change format.

(Received in February 2011, accepted in April 2012. This paper was with the authors 2 months for 2 revisions.)

Key Words: Extended Queuing Systems, Metamodelling, Domain Specific Modelling Language, Discrete Event Simulation, Interoperability

1. INTRODUCTION

Several Domain Specific Modelling Languages (DSMLs) have been developed in the last decade. Their use gained a lot of success and a growing popularity. However, developing a DSML is still a challenging and time-consuming task. Domain specific modelling (DSM) often also includes automating the code generation directly from the DSM models. Automatic creation and maintenance of source code increases significantly developer's productivity. The reliability of automatic generation compared to manual coding will also reduce notably the number of defects in the resulting code thus improving quality.

Queuing network models have been used extensively as a modelling paradigm for deriving analytical as well as simulation based performance measures. They are commonly used to model a wide range of discrete event systems. Kendall's notation is a mean for describing queuing networks especially in case of simple systems. For complex ones, a graphical notation with textual annotations is used instead. To analyse a model either by simulation or by mathematical analytic tools, the model is commonly coded and saved directly in a proprietary simulation/analysis tool file format. Although it is the same formalism, there are always some ambiguities and disagreements on certain concepts as well as on the exchange format. Therefore, tools are not interoperable and models can't be reused. In addition to reconstructing models from scratch every time the tool is changed, the development of modelling environments and model transformations, including code generation, are time-consuming, very expensive and hard to validate and maintain. Model Driven Engineering (MDE) based on the concept of metamodelling seems to be the best solution to these problems.

The next section addresses key motivations and objectives of this work. A clear development methodology of DSML is presented in section 3. Section 4 discusses the basic domain concepts to capture relevant knowledge for constructing the metamodel presented in section 5. Concrete syntax and exchange format are described in section 6. Finally, conclusions and improvements are given in section 7.

2. MOTIVATIONS AND OBJECTIVES

The development and use of a common metamodel and file exchange format is motivated by:

1. Lack of interoperability: performance evaluation tools of discrete event systems based on queuing networks use different file formats for describing their models. There is no standard to ensure interoperability and reuse of conceptual models. Importing and exporting models is still hard to achieve due to the multitude of file formats. The problem is beyond simple technical solutions. XML-based standard exchange format seems to be the best solution.
2. Lack of expressiveness: unfortunately the expressive power of modelling languages depends on the tool purposes and capabilities. Tools don't model what they can't handle. In our opinion, a modelling language must be tool independent and focuses particularly on offering all necessary constructs for describing the largest possible range of systems.
3. Difficulty of the design and development of modelling environments: building modelling environments from scratch is hard, complex and time consuming. It is also difficult to maintain as the modelling language evolves. Model Driven Engineering seems offering the solution by automating the development based on metamodels.
4. Difficulty of code generation: the translation of the conceptual model into simulation model is complicated and needs careful verification in addition to programming skills. Besides the manual translation, the classical automatic translation alternative is barely coded in simulation environments using programming languages. So translation rules used in this case are not clear and not easy to modify as the used formalism evolve.

The first initiative for developing such a metamodel and an XML-based exchange format named QNML was undertaken by the Centre for Computer integrated Manufacturing enterprises [1]. It focuses only on analytical resolution of queuing networks and simulation was not considered. In addition, the expressiveness power of the formalism is limited and notably identical to that of the analysis tool RAQS developed at the same laboratory [2], so only concepts supported by this tool are considered.

Our aims behind this work are to explore basic concepts of Queuing theory, extract common terminology and clear semantics for designing a metamodel and an XML-based exchange format. An XML-schema will serve as a mean to define the markup language for describing queuing networks. Designing of an UML metamodel based on considered concepts that will serve as a basis for applying MDE techniques, especially, creating an integrated modelling environment and generating low level code from conceptual models. Taking into account the language extensibility, since this work constitutes only the starting version, other new concepts may be included. Modelling environment and automatic simulation code generation would evolve easily also. The standardization of this formalism facilitates greatly the task of experts and researchers in the domain of systems performance evaluation.

3. DSML DEVELOPMENT METHODOLOGIES

Several methodologies for developing DSLs have been proposed in [3-6]. A study conducted in [7], identified clearly and in detail the development process of DSLs independently of the

used tools and languages.

We can summarize the different stages of development:

1. Decision: the adoption of an existing DSL is the best solution and the decision to create a new one must be justified. The developing of a new DSL will have an economic impact to be considered (tools used, time spent, effort to develop, learning, etc.).
2. Analysis: the domain is clearly identified and basic concepts are collected. This requires the cooperation of domain experts and computer scientists. The domain analysis is often done informally, but clear methodologies have been proposed and can effectively serve as DARE [8], DSSA [9] and FODA [10].
3. Design: two important elements in this phase are considered, the relationship of the new DSL with an existing language and the degree of formality of the design description. The second element is the design specification. Once the image of the DSL is clear, it can be done informally using natural language for example, easy to produce but more difficult to handle, or formally, using known techniques such as regular expressions and formal grammars to specify syntax and rewrite systems, finite state machines and attribute grammars to specify the semantics.
4. Implementation: the implementation of the design is subject to various choices made about the nature of execution of the DSL. A non-exhaustive list of alternatives includes interpretation, compilation, pre-processor, embedding and extension (as a library of modifications made to the compiler/interpreter for the host language, etc...).
5. Deployment: the effort devoted to the deployment of the DSL and its acceptance is reduced by the success of the previous phases. In addition, the ease of use, adaptation to the domain, efficiency, expressiveness, access to the DSL, involvement of a large community for testing and the impact on productivity are key points to consider.

The first step towards the development of the Extended Queuing Networks Modelling and Markup Language (EQNM²L) has been presented in the previous section where motivations and objectives are discussed. The analysis of the domain knowledge is the critical phase that guides the design and implementation. For the deployment stage, an online and open source project has been created to involve a large community and easing access to this DSML.

4. DOMAIN KNOWLEDGE

Most of basic concepts discussed in this paper are based on [11-13] where a queuing network is considered as a set of service stations that are visited by jobs. Terms as transaction, customer, client and job refers to the same concept which characterizes any entity that moves through the network, from one station to another to acquire services.

4.1 Job classes and priorities

In queuing networks, jobs could belong to different classes. Job classes may differ in their service and arrival distributions, number of resource units needed to accomplish services and in their routing schemes leading to different life cycles.

A class independent priority mechanism may be considered to organize the whole population of jobs. Hence, jobs may be scheduled according to their priorities. Within the same priority category, another scheduling discipline must be indicated to distinguish between multiple jobs with equal priorities. It is possible also to consider High Value First (HVF) or Low Value First (LVF). A priority may be static or dynamic. While a dynamic priority changes its value over time, a static one assigned to a job remains unchanged. A dynamic priority may be time-dependent in the form of $Pr(t)$ or be a function of some system

parameters. It is also possible that a job changes its class or priority when it moves inside the system.

4.2 Service stations

Service centres, nodes, stations refer to the same concept. Each station is distinguished by a unique identifier and characterized by a number of parallel servers, usually identical, a waiting queue with limited or unlimited capacity, class independent scheduling disciplines, and a service probability distribution for each job class. Service for a particular job class is identified by a probability distribution and a rate. Rates are generally static, but it is also possible to apply dynamic rates varying over time, especially in terms of some system indicators.

Several service disciplines are used to serve jobs including FCFS (First Come First Served), LCFS (Last Come First Served), Service in Random Order (SIRO), Processor Sharing (PS), Shortest Job First (SJF), Round Robin, and Shortest Remaining Time First (SRTF). If Round Robin is used, the time slice must be indicated. It is important to mention that a variant of this discipline called Weighted Round Robin, consider job classes with weighted slices. Other specific disciplines may be considered.

The study of a system is influenced by its initial state which is described by the number of in each station. In contrast of the case where the system is initially empty, the realistic cases take into account the exact number of jobs in the system. This is important especially for closed systems where jobs are permanent components.

4.3 Asymmetric stations

If servers are identical, the station is qualified to be simple, ordinary or symmetric. Otherwise, it is said to be asymmetric where parallel servers present different characteristics. We consider here those having different service distributions or various vacation/failure schemes. In real world, it models for example, machines with different ages or manufacturers. This category of stations requires more information about job affectation in case of multiple idle servers. For example:

- Random: the task is assigned to a free server at random.
- Fastest Service: the task is assigned to the fastest free server (the greatest service rate or the smallest average time of service calculated from the log).
- Longest idle time: the task is assigned to the server which remained idle for the longest period of time from the free servers set.

It is possible to define other allocation strategies. In addition, sometimes the allocation process can't distinguish a single server and the use of another criterion is required.

4.4 Fork/join station

A special kind of stations able to fork jobs on arrival (of a given class) into several tasks (not necessarily of the same class) or recombines several tasks into one job. A Fork/join station may or not have a maximum of jobs to process simultaneously. Synchronization is necessary to accomplish this operation. The join station must wait for necessary tasks to run. In parallel processing, fabrication/assembly systems, supply chains and communication networks, the same split tasks of a specific job will be recombined in a join station. The join process begins as soon as all the different subtasks are waiting. In manufacturing, the situation may be more complicated since tasks from different classes are joined.

A fork station may operate independently from join stations or starts the next operation only upon accomplishment of a join operation. The special kind where the fork process of a

job can't start unless the join process for the previous job has finished is called split/merge station. In this case, response time becomes longer.

Generally, those primitives are instantaneous. Nevertheless it is possible to consider that such operations may take some amount of time specified by a probability distribution.

This idea could be generalized to consider split jobs as belonging to a new class, since their service process may differ from the original job.

4.5 Passive station

If we consider a computer system, then a job may use two or more resources at the same time, memory space and CPU for example. We consider here passive resources which are simple (passive) stations with a number of tokens (identical units) and allocate queues. It has no service to offer except allocating its tokens, so the service duration is generally assumed to be nil. These passive nodes are of two types. The first is devoted to allocation of tokens and the other for release.

When arriving at an allocate station, a job requests a number of tokens. If it gets them, it can continue visiting other nodes of the network; otherwise it must wait at this node. When arriving at the corresponding release node, it releases all its tokens. These tokens are then available for other jobs [11].

4.6 Decision station

A task, during its life cycle, may have to make decisions based on the state of the system. A discrete event system is described by a set of variables known as state variables. Thus, a choice structure may be assimilated to a service station in which the task takes a decision. The decision is based on a Boolean expression containing one or more state variables of the system among those predefined or user-defined.

The evaluation of the Boolean expression can have only two possible values. Hence, a decision structure has only two directions or issues. The task is then directed upon the result of the evaluation. Generally, this process is not time consuming.

4.7 Service types

1. Batch processing: Class dependent batch processing is possible by indicating the minimum and the maximum batch sizes. In this case, the whole batch is treated as one job. It is clear that a batch's jobs are homogeneous and belong to the same class.
2. Pre-emption: Some scheduling disciplines may or not cause pre-emption of the processed job when a new job arrives. The pre-empted job returns to the queue and may resume later starting at the point where it was stopped (pre-emptive resume) or restarting its service from the beginning (pre-emption without resume or pre-emptive repeat). Generally, pre-empted job returns to the front of the queue, but other strategies may be considered.
3. Blocking: it may arise in a station when its queue has a limited capacity. Several blocking models could be distinguished such as:
 - a. Rejection: the blocked job is forced to leave the system.
 - b. Blocking after service (BAS): the blocked job will be forced to wait in its origin station until the next station is able to accept it. Hence, the origin station will be blocked (still busy) and stops servicing other jobs.
 - c. Repetitive service (RS): the blocked job is forced to repeat service by joining an Orbit until the next station is able to accept it.
 - d. Waiting queue (WQ): a job that has just been served in a station is routed to a temporary waiting queue until next station can accept it. The origin station is released.

4. Load dependant service: In some situations [14], service may depend on the station load, expressed by the number of jobs inside the station. In this case, for each range of the station load, a service distribution is indicated.
5. Server failure and vacation: In practical queuing systems, service stations are not always reliable. They may become unavailable for a period of time for a variety of reasons (vacation, maintenance, failure, etc.). Servers failures are assumed to be independent identically distributed random variables specified by a distribution function, but some models requires different types of failure and follow different distributions. Failure may be synchronous (all servers fail at the same time) or asynchronous (servers fail independently). Two probability distributions are useful for modelling the mean time between failures (MTBF) and mean time to repair or recover (MTTR).

Vacation is slightly different from failure. While the latter is accidental, the former is scheduled and conscientiously undertaken [15]. It is a process governed by a policy that explicitly specifies:

- a. Starting rules: exhaustive when beginning vacation only if queue (system) is empty, otherwise it is non-exhaustive and can begin at any time.
- b. Termination rule: determines when resuming. Multiple vacation policy if keeping vacation until having a job to serve. Single vacation policy if resuming immediately and begin servicing if customers are waiting or stay idle otherwise.
- c. Vacation duration: symmetric (independent and identically distributed) or asymmetric (different distributions).

For multi-server systems, starting and termination policies are more complicated (synchronous or asynchronous). Other policies are also possible like guarantee of minimum service availability.

4.8 Source

A service station may or not receive jobs from outside of the system (exogenous arrivals), but must have at least an output job flow towards another station or outside the system.

An input from outside denotes a source of jobs and has a unique identifier. It is associated to an exogenous job flow characterized by a probability distribution, a job class and a priority to describe the corresponding arrival process. It is also possible to specify batch arrivals by indicating the distribution of the batch size. Arrival rate for a specific job class is generally static, but we may consider dynamic rates. Some sources have a limited population and it is important to take this into consideration.

4.9 Job behaviour

In a queuing model, the behaviour of jobs (or customers) is an important feature and three major situations may arise:

1. Balking: Some customers decide not to join the queue because of the queue length or insufficient waiting space. This behaviour results in the discouragement of customers for not joining an improper or inconvenient queue. It could be implemented using decision stations.
2. Reneging: Once in a queue, a jobs may choose to leave it if it have waited too long ("timeout reneging"). It requests a resource and wants to leave before acquiring it, for example because it could get quicker service by another resource with a shorter queue ("conditional reneging"). Reneging pertains to impatient customers. After being in queue for some time, few customers become impatient and may leave the queue.
3. Jockeying: Jockeying is a phenomenon, in which customers move from one queue to

another queue with a hope that they will receive quicker service in the new position. Job's behaviour is station dependant. It is more suited to be indicated for each station.

4.10 Orbit

In many situations, jobs leaving a service station join a region called "Orbit". After a delay they retry their queries. So, an orbit is a source of jobs fed by the system itself and allows its jobs to retry queries after a random time. In queuing theory, this is a key feature for retrial queues [16].

4.11 Output

Each station may have one or more outputs. A routing strategy must be defined for each job class. An output is identified by a reference to the destination station and routing information according to the routing strategy.

It is possible that a job changes its class after a service. This situation is common in manufacturing systems where a given machine transforms received products to new ones. An output normally needs to indicate associated resulting job class for every received one.

4.12 Routing strategy

After service completion, a job is routed according to a routing strategy which may be for example:

- Probabilistic: job moves to the next station with a given probability.
- Round robin: all possible destinations for a job class are chosen in a cyclic manner.
- Shortest queue length: job moves to the next station having the shortest queue.
- Smallest response time: job moves to the next station having the smallest response time calculated from the log.
- Smallest utilization: job moves to the next station having the smallest average utilization calculated from the log.
- Fastest service: job moves to the next station having the smallest average service time calculated from the log.

4.13 Regions

The concept of regions is very powerful and denotes a set of stations in the system. It allows expressing global constraints on a group of stations like finite capacity for specific job classes. It is also useful in global statistics computation. Hence, a Finite Capacity Region is a region of the model or a set of stations, where the number of jobs is controlled [17]. It is possible to define an FCR capacity by setting upper bounds for the number of customers in the region for specific classes or globally regardless of the job classes.

5. METAMODEL

Models are created using a modelling language called a metamodel. The modelling language is itself described in another language called meta-metamodel. The philosophy of DSM favours the creation of a new language for a specific task, and hence there are naturally new languages designed as meta-metamodels. DSM environments can considerably reduce the cost of obtaining tool support for a DSML, since a well-designed DSM environment will automate the creation of application modules costly to build from scratch, such as domain-

specific editors, compilers and models transformers. The domain experts only need to specify the domain specific constructs and rules, and the DSM environment provides a modelling tool tailored for the target domain.

It has been established that a language is characterized by its syntax and semantics. The syntax describes the different language constructs and their arrangement rules while semantics refers to the relationship between a signifier (program or model) and a signified (mathematical object) to give meaning to each of the constructs of the language. So the relationship between semantics and syntax is the same as between the content and the form.

The metamodel represents the abstract syntax and is the heart of the language, since it captures the whole domain concepts and their relationships. It acts as a pivot between the concrete syntax description and the semantics description. Concrete syntax contains information on how the concepts in the metamodel are to be represented to the language user. This is sometimes amended by a mapping in which each abstract concept is related to a textual notation or a graphical symbol.

The metamodel of EQNM²L illustrated by the Fig. 1 is expressed in UML class diagram. It represents the abstract syntax and a part of the static semantics. The rest of the static semantics could be expressed using OCL.

An extended queuing network (EQNET) is an aggregate of nodes, decision stations, sources, orbits, outputs, outsides, sink, finite capacity regions and a description. An EQNET may have a description which consists of a set of meta-information about the model as name, used tool, version and author. A node is an abstract concept that models a variety of specific stations as simple, passive and asymmetric stations. In contrast of an open network, a closed network does not need a sink.

It is important to note that some domain concepts discussed in previous sections are not considered in this minimal metamodel. Fork and Join station are still ambiguous and need agreement on their exact semantics. The metamodel in its actual state lacks of many concepts but is extensible to include new concepts in order to gain more maturity.

The static semantics is part of the metamodel. It consists of a set of rules to ensure that it is well formed (well-formedness rules). Rules allow expressing constraints and thus reducing the overall valid models. We will present these constraints in natural language while trying to be concise and clear. Subsequently, these rules will be expressed in OCL during the implementation phase in respect to the chosen implementation tool. In addition to the cardinality constraints already present in the class diagram, the other rules are:

1. All identifiers must be strictly positive integers which include nodes, sources, orbits and finite capacity regions ($id > 0$).
2. The identifier of a node is unique in the model.
3. A source of jobs must direct its job flow only to one node.
4. A job flow of the same class and priority can't be directed more than once from one node to another, to the outside or to an Orbit. It means that identical job flows from the same node could not be directed to the same target (Node, Orbit or Outside).
5. For a job class, the network of queues can be either open or closed.
6. Sources and orbits of the same station must have distinct identifiers.
7. A probability is always a real number comprised between 0 and 1 ($0 \leq pr \leq 1$).
8. If the routing strategy in a given station for a specific job class is probabilistic, then the sum of probabilities is equal to one ($\sum pr_i = 1$).
9. Job classes are positive integers (for Routing, Designation Output and Outside), but for a source or an orbit, it must be strictly positive (the class number 0 is reserved to represent any class not explicitly mentioned).
10. Only one Sink is authorized in a model (for open networks) for which at least one source

must exist (no sink without sources).

11. An Orbit manipulates only one job class, so it must receive only this class from the corresponding node to which, as a source, it must exclusively direct jobs.
12. Finite Capacity Regions must have distinct identifiers (identifiers are unique).
13. A decision station must not have output connections, but only IfTrue and ifFalse.
14. A node may belong to different finite capacity regions and a finite capacity region may contain different nodes.
15. The population attribute of a source of jobs is a positive integer (population $\neq 0$). Zero means unlimited.

Figure 1: EQNM²L Metamodel.

6. CONCRETE SYNTAX AND EXCHANGE FORMAT

Some DSL developers consider that in modern visual (or graphical) language environments there is no need to be very specific about concrete syntax. At the low level, information can be exchanged between different tools in a freeway concrete-syntax using XML, and at the high level, humans can input linguistic utterances into a tool using forms. In consequence, the only thing that remains is the need for renderings of models which are meaningful to humans.

However, because a language is a means of communication that must preserve interoperability, all tools developers need to agree on the XML schema for interchange as well as on the symbols to be used in rendering. In addition, both the XML schema and the set of symbols for rendering are considered concrete syntaxes of the language. If no agreement on concrete syntaxes is done, a lack of interoperability and confusion would arise. It is clear that a mapping from abstract syntax to a concrete syntax is as important as the inverse mapping.

It is possible to define several concrete syntaxes for the same abstract syntax. We adopted the derivation illustrated in Table I, matching each element of the abstract syntax with a visual (graphic) appearance. It is important to provide visual elements as distinct (or similar) as their abstractions. The visual syntax must be as close as possible to the domain graphical notation. This will facilitate the task of domain experts, ensure large deployment and encourage acceptance of the DSML.

Table I: Mapping between abstract and concrete syntax.

Abstract syntax	Concrete syntax	Abstract syntax	Concrete syntax
Simple Station		Sink
Asymmetric Station		Decision
Allocation		Finite capacity Region
Release		State
Source		Server
Orbit		Routing

For simple languages and in most of the current DSM tools, concrete syntax representation is directly mapped to the abstract syntax. Logical entities are always visualised as nodes, and logical relationships as edges. Nevertheless the experience has shown that using more complex metamodels, especially those conceived for automated model transformations, not only results in visual models being too complicated to overview, but it can also drain system resources heavily. Consequently, modern approaches make use of a separate visualisation metamodels, which describe the structural appearance of diagrams. The last technique allows hiding superfluous details; however it is still limited in the sense that classes can only be mapped to nodes and references to edges.

The problem with EQNM²L formalism is the exchange and reuse of developed models between various simulation and analysis tools. It is necessary to refer to an exchange standard format which must be open and promotes maximum interoperability.

Various textual exchange formats exist and most of them are based on XML. For example, QNAT software [18] uses the Mathematica format and RAQS software [2] uses a specific ASCII format. For JMT [17], it uses XML as a mean to describe models. The syntax is specified by the JMT model schema which serves as the unique format used by the simulation and the analytical engines. The main problem here is not only the multiple used formats, but even if XML is used, the solution suffers from the lack of expressiveness and agreement on a standard format. The exchange format for EQNM²L [19] is a proposed draft and may serve as a basis for developing a more adequate, stable and mature templates. It represents a metamodel described as an XML-schema. Each element of the metamodel is projected as an XML element. For the inheritance concept, XML provides restrictions and extensions that are similar useful constructs. This concept is illustrated in Fig. 2 where a Simple Station inherits from a Node. In addition, some constraints could be expressed easily, especially those of uniqueness.

```

<complexType name="Node">
  <sequence>
    <element name="state" type="eqnm2l:State" minOccurs="0" maxOccurs="unbounded"/>
    <element name="routing" type="eqnm2l:Routing" maxOccurs="unbounded"/>
    <element name="source" type="eqnm2l:Source" minOccurs="0" maxOccurs="unbounded"/>
    <element name="output" type="eqnm2l:Output" minOccurs="0" maxOccurs="unbounded"/>
    <element name="orbit" type="eqnm2l:Source" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="identifier" type="positiveInteger" use="required"/>
  <attribute name="capacity" type="nonNegativeInteger" use="optional" default="0"/>
  <attribute name="blocking" type="eqnm2l:blockType" use="optional" default="Rejection"/>
  <attribute name="disciplines" type="eqnm2l:disc" use="optional" default="FCFS"/>
  <attribute name="FCRs" type="eqnm2l:FCR-Type" use="optional"/>
  <attribute name="slice" type="eqnm2l:positiveDouble"/>
  <attribute name="preemption" type="eqnm2l:Preemption" use="optional"/>
</complexType>

<complexType name="SimpleStation">
  <complexContent>
    <extension base="eqnm2l:Node">
      <sequence minOccurs="1" maxOccurs="1">
        <element name="service" type="eqnm2l:Service" maxOccurs="unbounded"/>
        <element name="failure" type="eqnm2l:Failure-Vacation" minOccurs="0"/>
      </sequence>
      <attribute name="nbServers" type="positiveInteger" default="1"/>
    </extension>
  </complexContent>
</complexType>

```

Figure 2: Fragment of the EQNM²L XML-based exchange format.

7. CONCLUSION

We presented EQNM²L which is a domain specific modelling language based on queuing theory. Extended queuing networks are simply queuing networks with an extension for new concepts that help modelling more accurately. Obtained EQNM²L models may have analytical solution, approximations, or simulation techniques but expressiveness is the most interesting feature. The formalism is really tool-independent and fairly expressive if we consider that it constitutes only a starting version and extensible to include new concepts.

Main advantages of EQNM²L include high level interoperability enhancement throw the proposed metamodel. In the context of MDE, the metamodel would guide the creation of modelling environments and models transformation, in particular automatic code generation. In addition, automation allows reducing development time and make validation easier. It is possible de define a common visual (concrete) syntax, but even if several rendering schemes are adopted by different tools, the metamodel remains the same.

The presented XML-based exchange format enhances low level interoperability by encouraging models exchange and reuse between different tools. Moreover, automatic simulation code generation for various tools in different programming or simulation languages could be achieved at this level using Text to Text (T2T) transformations.

The metamodel presented in this paper is a minimal version and several concepts are not yet considered. It could be extended easily, whenever a new concept has a clear semantics. We focused only on the analysis and design phases, but the development process must be achieved using existing DSL tools to offer various modelling environments. Domain experts

must be involved in order to improve different aspects of this DSML. This includes more complete metamodel, more concise semantics and more adequate visual or concrete syntax. In addition the proposed exchange format must also reflect those efforts. In our opinion, it is much easier to conclude an agreement on a DSML than a specific tool.

REFERENCES

- [1] Chalavadi, U. M. (2004). Automatic configuration of queuing network models from business process descriptions, Master Thesis, Oklahoma State University
- [2] Kamath, M. (1994). Recent developments in modeling and performance analysis tools for manufacturing systems, *Computer Control of Flexible Manufacturing Systems*, 231-263
- [3] Hudak, P. (1998). Modular domain specific languages and tools, *Proceedings of the Fifth International Conference on Software Reuse (JCSR98)*, 134-142
- [4] Spinellis, D. (2001). Notable design patterns for domain-specific languages, *Journal of Systems and Software*, Vol. 56, No. 1, 91-99, [doi:10.1016/S0164-1212\(00\)00089-3](https://doi.org/10.1016/S0164-1212(00)00089-3)
- [5] Kelly, S.; Tolvanen, J. P. (2008). *Domain-Specific Modeling: Enabling Full Code Generation*, John Wiley and Sons, Hoboken, New Jersey
- [6] Cook, S.; Jones, G.; Kent, S.; Wills, A. C. (2007). *Domain-Specific Development with Visual Studio DSL Tools*, Addison-Wesley, Upper Saddle River
- [7] Mernik, M.; Heering, J.; Sloane, A. M. (2003). *When and how to develop domain-specific languages*, Report Software Engineering SEN-E0309, Stichting Centrumvoor Wiskundeen Informatica (CWI), Amsterdam
- [8] Frakes, W.; Prieto-Diaz, R.; Fox, C. (1998). DARE: Domain analysis and reuse environment, *Annals of Software Engineering*, Vol. 5, No. 1, 125-141, [doi:10.1023/A:1018972323770](https://doi.org/10.1023/A:1018972323770)
- [9] Taylor, R. N.; Tracz, W.; Coglianese, L. (1995). Software development using domain-specific software architectures, *ACM SIGSOFT Software Engineering Notes*, Vol. 20, No. 5, 27-37, [doi:10.1145/217030.217034](https://doi.org/10.1145/217030.217034)
- [10] Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E.; Peterson, A. S. (1990). *Feature-oriented domain analysis (FODA) feasibility study*, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University
- [11] Bolsh, G.; Greiner, S.; de Meer, H.; Trivedi, K. S. (2006). *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*, Second Edition, John Wiley & Sons, Hoboken
- [12] Gross, D.; Shortle, J. F.; Thompson, J. M.; Harris, C. M. (2009). *Fundamentals of Queueing Theory*, 4th Edition, John Wiley & Sons, Hoboken
- [13] Baccelli, F.; Bremaud, P. (2003). *Elements of Queueing Theory*, Springer Verlag, Berlin
- [14] Al-Hawari, T.; Aqlan, F.; Al-Araidah, O. (2010). Performance Analysis of an Automated Production System with Queue Length Dependent Service Rates, *Int. Journal of Simulation Modelling*, Vol. 9, No. 4, 184-194, [doi:10.2507/IJSIMM09\(4\)2.168](https://doi.org/10.2507/IJSIMM09(4)2.168)
- [15] Tian, N.; Zhang, Z. G. (2006). Vacation Queueing Models: Theory and Applications, *International Series in Operations Research & Management Science*, Hillier, F. S. (Series Editor), Stanford University, Springer Science + Business Media, LLC
- [16] Gomez-Corral, A.; Artalejo, J. R. (2008). *Retrial Queueing Systems: A Computational Approach*. Springer-Verlag, Berlin
- [17] Bertoli, M.; Casale, G.; Serazzi, G. (2009). JMT: performance engineering tools for system modelling, *ACM SIGMETRICS Performance Evaluation Review*, Vol. 36, No. 4, 10-15, [doi:10.1145/1530873.1530877](https://doi.org/10.1145/1530873.1530877)
- [18] Manjunath, D.; Bhaskar, D. M.; Tahilramani, H.; Bose, S. K.; Umesh, M. N. (1998). QNAT: A Graphical Tool for the Analysis of Queueing Networks, *IEEE TENCON'98, International Conference*, New Delhi, Vol. 2, 320-323
- [19] Sourceforge. EQNM2L, from <http://sourceforge.net/projects/eqnm2l/>, accessed on 01-02-2011