A Knowledge Base to Support the Automatic Derivation of Performance Models of Operational Systems

Mahmoud Awad
Dept. of Computer Science, MS 4A5
Volgenau School of Engineering
George Mason University
Fairfax, VA 22030, USA
mawad1@gmu.edu

Daniel A. Menascé
Dept. of Computer Science, MS 4A5
Volgenau School of Engineering
George Mason University
Fairfax, VA 22030, USA
menasce@gmu.edu

ABSTRACT

An essential function of autonomic self-managing systems is to dynamically adjust their configuration and system resources to respond to changing workload demands. This adaptation can be achieved by searching for an optimal or near-optimal configuration with the help of combinatorial search techniques that use analytic performance models to assess the utility of each configuration evaluated by the search method. One challenge of this approach is that manual generation of performance models in autonomic systems may be impractical or impossible without a priori knowledge of the system's functions and detailed architecture. Additionally, manually building performance models requires skills not always available. This paper discusses an overall framework for automatic performance model derivation. Parts of that framework have been addressed in prior publications by the authors. The focus of this paper is on a *knowledge base* that stores a variety of known performance models and their solutions. The models in the knowledge base range from single queue to queuing network models. The paper presents an XML-based language to describe models and solutions in the knowledge base as well as a method to search for candidate models that match measurements obtained from operational systems.

Keywords

Performance models; Automatic derivation of performance models; XML; PMIF; M/M/1; M/G/1; M/M/c; M/G/c.

1. INTRODUCTION

Modern computer systems are becoming increasingly complex and dynamic and, consequently, more difficult to model and manage. Analytical performance modeling aims to abstract much of the system complexity by focusing on the important system characteristics. However, there are still a number of impediments to the widespread use of analytical performance models. First, modeling requires highly specialized skills by performance engineers who develop models designed specifically for a particular system. Individuals with such specialized skills are hard to find, and large organizations that use multiple computer systems would need to develop and maintain a separate model for each system, which is costly.

Second, the dynamic nature of modern computer systems means that analytical performance models need to constantly adjust to changing workloads, resource demands and changes in system parameters, design and architecture. Therefore, the modeling process needs to be repeatable, dynamic and, if possible, automated in order to respond to system changes.

Finally, even though models are abstractions of computer systems, detailed knowledge of the system architecture and user behavior are considered essential to the modeling process. The decision to abstract one aspect of the system while modeling another is one of the specialized skills of performance engineers. When modeling existing operational systems, such detailed knowledge of the computer system internals may not be available or easy to gather, possibly because of the system complexity or the inability to conduct detailed analysis of the system architecture and user behavior.

To address the difficulty to manage complex and dynamic systems, system vendors and hosting providers are incorporating *autonomic* capabilities that allow systems to self-configure, self protect, self-heal, and self-optimize. Such capabilities are collectively referred to as self-management or self-* [13]. In autonomic systems, self-management features are implemented using an autonomic controller whose job is to monitor, analyze, plan and adapt systems to respond to their changing environment, such as variable workload intensity and resource utilization. Autonomic systems and cloud data centers greatly benefit from incorporating analytical performance models into their controllers. Analytical models offer performance prediction capabilities and, therefore, more timely and more efficient resource allocation compared to static and reactive adaptation techniques.

The thesis we posit in our research is that impediments to developing analytical performance models in autonomic systems can be overcome by automating the repeatable steps taken by performance engineers. More specifically, we discuss an automation framework for model derivation of operational systems, which is a bigger challenge than modeling systems that are still going through the software development life cycle. We do realize, though, that full automation of system modeling is not feasible given the large space of possible system configurations, system architectures, application designs and user behavioral patterns. However, there is a large number of clues that can be gleaned from an operational system by observing its behavior through its resource consumption and operational logs. For example, architectural system composition can be inferred using monitoring tools and operating system commands readily available in autonomic systems, such as cloud computing environments.

When modeling an operational system, where few internal details are available, one might resort to source code recov-

ery techniques, where source code is recovered and analyzed to determine the application behavior and its impact on the hardware resources. Another approach is to analyze system logs and audit traces in order to understand the application's architecture and behavior. Reverse engineering the source code is intrusive and assumes having proper access to the application executables and permission to reverse engineer them. On the other hand, system log and audit trace analysis—although less accurate—is less intrusive while providing more insight into the system internals than black-box modeling.

In [3], we introduced a framework for the dynamic derivation of analytical performance models in autonomic systems, which is also illustrated in Fig. 1. The framework can be incorporated in autonomic systems in order to dynamically adjust their configurations and system resources to respond to changing workload demands. The framework uses non-intrusive techniques to infer a workload model and a system model (queues and interconnections), which are combined to produce an analytical performance model without much user intervention.

In our framework, the system model is identified using reverse engineering techniques to determine the architectural pattern that best matches the system. The workload model is identified by collecting and analyzing the system and application logs. These logs represent the system's observable behavior, such as arrival rates, system's concurrency level, response times and CPU utilization. Depending on how much can be harvested from the system and application logs, our framework determines which overall analytical model to select (black-box, white-box or the right level of gray in between). Black-box modeling represents the highest level of system abstraction, and it is the least expensive but least accurate, while white box modeling is the most expensive but also the most accurate. Performance modeling is the art of finding the most accurate model representation of the system at an acceptable cost.

This paper focuses on the design, implementation, and validation of the Performance Model Analyzer component. This component finds the best-fit analytical performance model given the results produced by the Workload Model Analyzer. The Performance Model Analyzer includes a comprehensive repository of performance models and their characteristics. This Analyzer compares the observed features of the application system with the available performance models in the repository. It then associates the best-fit analytical performance model with the corresponding application model, and passes on that information to the Adaptation Planner.

The repository of performance models is stored in the autonomic controller's knowledge base using an XML-based representation inspired by the XML-based Performance Model Interchange Format (PMIF) [24]. We chose XML to represent performance models because XML is human and machine readable, used by many applications to describe system configurations, used to isolate application layout from its behavior and is generally accepted as a standard data interchange format. The storage of the repository in a native XML database speeds up the process of comparing the system's observed behavior and workload parameters with the stored performance model characteristics.

PMIF is a queuing network model interchange format used to represent system performance model data for the purpose of moving model definitions between modeling tools [24]. Modeling tools either support the PMIF specification natively or provide an interface to transform their native model definition format to PMIF and vice versa. PMIF was designed as a standard portable format aimed at representing performance models whose characteristics and workload parameters are known. Contrary to the representation in our knwoledge base, the PMIF representation does not include solutions to the models it represents. In this context, we differentiate between techniques used for model specification and those used for model identification. For the purpose of our automatic derivation framework, we need to be able to compare the observable behavior and architectural characteristics of an operational system to a set of standard model specifications in order to determine the best-fit performance model. For example, given an inter-arrival distribution and service time distribution, which of the analytical models in the knowledge base best approximates the operational system's workload model and system model?

The XML representation used in our knowledge base uses a subset of the elements and attributes of PMIF 2.0 and its extensions and adds new elements to it so that it can be used for model identification.

Based on the results of input and output log analysis, our framework selects the appropriate workload model and system model from the model library and combines them to arrive at a Computed Model (CM) for the application system. The CM is then solved and compared to the actual system in order to determine the model accuracy, and whether or not a more detailed model for the same architectural pattern should be used [1].

We conducted a number of experiments to evaluate the framework's automation capabilities. In this paper, we focus on the ability of the framework to select the analytical model that most accurately describes the actual system. More detailed experiments showing the predictive power of the CM can be found in [2].

The main contributions of this paper are: (1) A PMIF-inspired XML representation of analytical performance models used in the automated model generation process. (2) An approach for model fitting that takes into account the availability of operational system logs and uses reverse engineering techniques.

The rest of the paper is organized as follows. Section 2 describes our framework for the dynamic derivation of analytical performance models. Section 3 describes our proposed XML-based representation of analytical performance models that is inspired by PMIF. Section 4 describes our model identification and validation process and algorithm based on the proposed XML-based model representation. Section 5 discusses some related work. Finally, section 6 presents discussions and concluding remarks. The appendix shows an excerpt of the XML schema used in our framework to define models in the knowledge base.

2. FRAMEWORK

Figure 1 illustrates the framework for the automatic derivation of analytical performance models within the context of the MAPE-K loop of autonomic systems [13]. The MAPE-K loop, or the autonomic manager control loop, stands for **M**onitor, **A**nalyze, **P**lan, and **E**xecute based on **K**nowledge. Our framework consists of the following components:

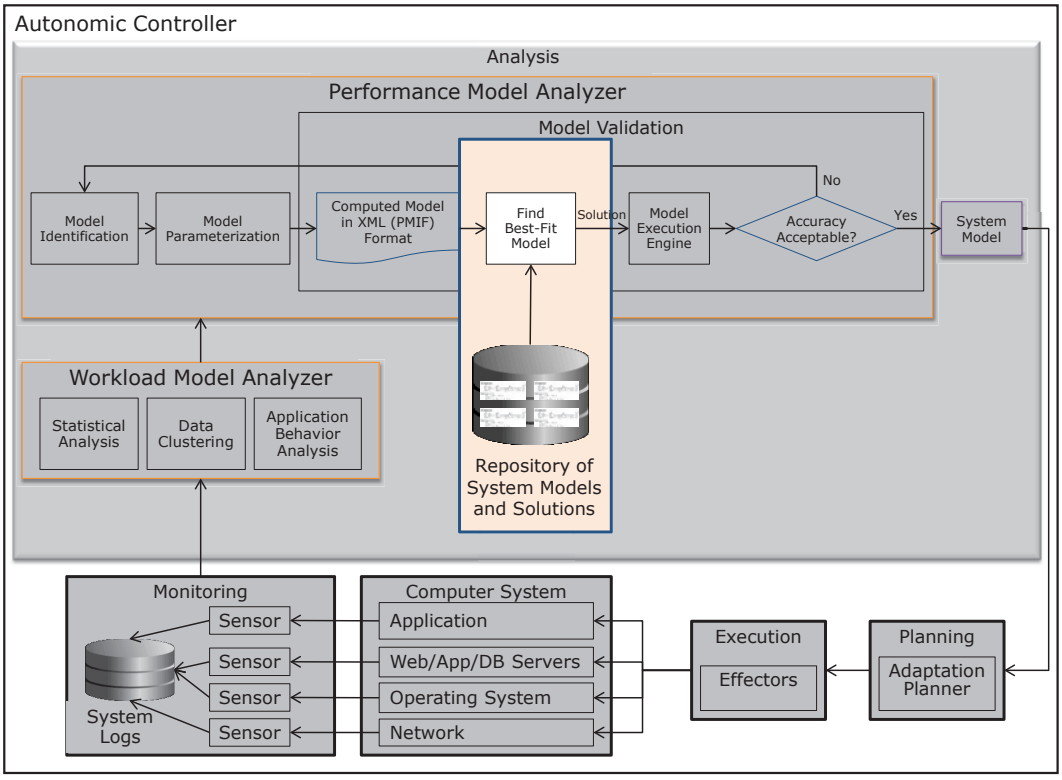

Figure 1: Automatic Performance Model Derivation

1. **Monitoring**: In this step, performance-related data is collected from the various network, operating system, server, and application logs.

2. **Analysis**: In this step, the Workload Model Analyzer extracts possible performance model parameters by parsing server logs and hardware resource utilization logs. Some of the analyzer's functions include statistical analysis to determine the best-fit data distribution, data clustering to determine the transaction classes and system behavioral analysis to determine the various patterns of interaction between users and the system using Customer Behavior Model Graphs (CBMG) [19] and, internally, between system components using Client-Server Interaction Diagrams (CSID) [18].

   The Performance Model Analyzer, which is the focus of this paper, uses a repository of performance models to find the best-fit analytical performance model given the results produced by the Workload Model Analyzer. The Performance Model Analyzer includes three steps: model definition, model parameterization, and model validation.

3. **Planning**: The Adaptation Planner produces adaptation plans based on the selected performance model in order to predict future resource needs or respond to immediate system adaption needs.

4. **Execution**: The Effector's job is to implement the adaptation plans by adjusting hardware or software resources as needed.

2.1 Workload Model Characterization

The Workload Model Analyzer's behavioral analysis component was addressed in [4], where we presented an automation technique for deriving CBMGs, which model the repeated sequence of interactions between the user and the application system. User sessions are identified and analyzed, and all the repeated sequences are modeled as CBMGs, which are then clustered to produce the average number of visits to each system function, and consequently, the service demands per system function. We ran a number of experiments that showed that our approach for deriving CBMGs is accurate and can be used to estimate the various system workloads.

In this paper, we assume that the system's logs have been analyzed and that statistical analysis, data clustering and system behavioral analysis have been performed as explained in [4].

2.2 System Model Characterization

System Decomposition

The Performance Model Analyzer begins by decomposing the system into individual components, which are modeled as queues, and determining the interconnections between the queues. Some system components may or may not be relevant to the final model, and the automated process must determine whether or not to disregard components depending on their level of impact on the system performance. It is also important to determine the system boundaries in order to separate internal system components from external inter-

faces and determining whether or not the external interface is an integral part of the system. It is conceivable, though, that the decomposition may reveal a system component with no clear interconnections with other components.

Components are divided into three categories when decomposing the operational system.

1. Hardware servers, i.e., the physical computers or virtual machines on which software applications run. Depending on the modeling level, further decomposition may be needed to determine the devices that make up the hardware server, such as disks, memory, and CPU.

2. Software servers, such as web servers. application servers, and database servers. Multiple software servers may reside on the same hardware server or virtual machine. At this point, we are only concerned with decomposing the system without determining the structure or relationship with other software servers.

3. Application modules, such as web services, batch jobs, and database procedures.

Although our framework limits the system model decomposition process to these components, fine grained decomposition is also possible, for example, by further identifying low-level methods or functions.

The system decomposition process involves parsing two types of files—log files and configuration files—in order to identify the software servers and application modules. Configuration files identify the components while log files determine how often they are accessed or how they are related. Comparing log files and configuration files help differentiate between internal components versus external interfaces. Any component referenced in the log files but not in the configuration files are considered external interfaces.

The hardware server specifications are collected by running basic non-intrusive operating system commands, such as the Linux built-in utility `lshw`, against the set of servers discovered by parsing the log and configuration files. In addition, the decomposition process determines how often a component is referenced, accessed or invoked, which helps determine the relevance of a system component in the modeling process in addition to its impact on the system performance.

System Components Interconnections

The interconnections between system components are identified using three methods:

1. Hardware Server Decomposition is obtained by parsing the results of Linux commands, such as `lshw`.

2. Software Server relationships produce Client-Server Interaction Diagrams (CSID) and are obtained by parsing configuration files and access logs.

3. Software Module relationships are identified by parsing access logs and invocation logs and produce a Customer Behavior Model Graph (CBMG).

Software Server Deployment on Hardware Servers

It is also important to determine which software server is installed on which hardware server or virtual machine. A software server is hosted on one and only one hardware server while many software servers can be hosted on the same hardware server.

Architectural Patterns

In its simplest form, the architectural pattern of any system can be either a single-queue pattern or a queuing network (QN) pattern. Our framework models every system initially as a single queue, and determines whether or not further decomposition of the system is needed in order to provide a more accurate model.

As described later in the section 3, an architectural pattern is a set of nodes (system components) and their interconnections. If the model is representing the interconnections between hardware components, the discovery of multiple CPUs or multiple CPU cores indicates multiprocessing, which can be modeled as a waiting line and multiple servers. At the software server level, the same pattern of one waiting line and multiple servers indicates load balanced servers. The same pattern at the software application level indicates a Map-Reduce or split-join function. Software architectural patterns such as client-server architecture and service-oriented architectures can also be modeled as a QN.

As mentioned above, the Performance Model Analyzer, includes a Model Identification engine whose jobs is to analyze system logs and application and operating system configuration files to find the analytical performance model that best describes the operational system. The following are examples of how the Model Identification engine can distinguish between architectural patterns that are inherently represented by an open QN model or a closed QN model:

1. Multitier architecture: Look in the server logs and configuration files for references to multiple software servers or hardware servers. Look for mappings between software servers and hardware servers.

2. Service-Oriented Architecture (SOA): look for fully encapsulated services with service composition, service discovery, SOAP or REST protocols with payloads in XML or JSON formats.

3. Online Analytical Processing (OLAP): Look for reporting frameworks with business intelligence, ETL, data warehousing, data mining, data marts and dimensional modeling.

4. Batch Processing: Look in the system logs for batch ETL processes, repeated executions of the same process at the same time of the day.

3. MODEL REPRESENTATION AND IDENTIFICATION

As indicated above, our XML model representation is inspired by PMIF. The following highlights the major similarities and differences between our XML schema and the various PMIF extensions:

1. Inter-arrival and service time distributions: We introduced the InterarrivalDistribution element to facilitate the identification of single queue models. We also included ServiceTimeDistribution in the SystemComponent element for the same purpose. The SystemComponent element is the equivalent to the Node element in PMIF.

2. Workload characterization: In PMIF, the various workload types and attributes are already identified and the actual workload intensity values are specified in the OpenWorkload and ClosedWorkload elements. For our purpose, we need to automatically identify whether the operational system includes a single transaction class or multiple classes, an open, closed, mixed or interactive workload. When the workload model is characterized, the XML schema determines which XML elements and attributes to collect for the corresponding workload type. For example, ClosedInteractiveMultiClass workload type shares the TransactionClassID, NumberOfJobs and TimeUnits attributes with the ClosedMultiClass workload type but has an additional attribute of ThinkTime. Recognizing that the operational system includes an interactive user population means that the framework needs to find the average think time from the operational system's access logs.

3. System model identification: We used the PMIF's optional Arc element, which is considered redundant in PMIF because of the Transit element. The Arc (with its From and To attributes) is a simple and intuitive mechanism for representing customer-system interaction using Customer Behavior Model Graphs (CBMG) [19] and system components interaction using Client-Server Interaction Diagrams (CSID) [18].

4. Model solution formulas and algorithms: For each model in our framework's repository, we specify the exact or approximate formulas and algorithms used to solve the model. PMIF cannot, and does not need to, be used to represent model solutions because it relies on the modeling tools' built-in solvers for that. Remember that PMIF is aimed at representing a model specification so that it can be exchanged between modeling tools.

   We provide a mechanism for models to be solved automatically (preferably programmatically) without user intervention even when interactive modeling tools are unavailable. Our XML schema provides a mechanism for specifying multiple solution formulas and algorithms and the name of the exact class or method called (and in which programming language) in order to execute the formulas or algorithms.

5. Simple types: We used the same simple types used in PMIF in anticipation of integrating some PMIF extensions into our XML knowledge base. We currently use the same TimeUnitsType and SchedulingType elements as PMIF.

Figure 2 shows our proposed XML schema that is inspired by PMIF, and listing 2 in the appendix is an excerpt of the XML-schema. The full XML schema and sample model definitions from the knowledge base can be found at cs dot gmu dot edu/~menasce/KB_XML_Schema/.

The following is a brief description of the XML classes and attributes of our XML schema:

- **AnalyticalPerformanceModelType**: Consists of a **WorkloadModel** and a **SystemModel**
- **WorkloadModel**: has an element of InterarrivalDistribution (Exponential or General) and a workload type: OpenSingleClass, OpenMultiClass, ClosedSingleClass, ClosedMultiClass, ClosedInteractiveSingleClass, ClosedInteractiveMultiClass, MixedSingleClass or MixedMultiClass. All Open workloads include transaction classes with the following attributes:
  - TransactionClassID: Uniquely identifies transaction classes and is used to associate a class ID with a queue ID in order to specify the server's service demand per transaction class.
  - ArrivalRate, ArrivalUnits, TimeUnits: These attributes tell the automation engine which workload attributes are needed from the operational system in order to solve the system model. The equivalent attributes for Closed workloads are NumberOfJobs, ThinkTime and TimeUnits. Mixed workloads may include multiple transaction classes with the corresponding attributes for the respective transaction class.
- **SystemModel**: Consists of an **ArchitecturalPattern** and a **ModelSolution**.
- **ArchitecturalPattern**: Similar to PMIF, and architectural pattern consists of one or more nodes (SystemComponents) with optional directional arcs. An Arc connects FromSystemComponent to ToSystemComponent.
- **SystemComponent**: Has a unique identifier and a name and is described using ServiceTimeDistribution, NumberOfServers, ServiceDiscipline (which are self explanatory) and ServiceDemandPerClass.
- **ServiceDemandPerClass**: For each of the workload's TransactionClassIDs, ServiceDemandPerClass specifies the ServiceDemandValue and ServiceDemandTimeUnits.
- **ModelSolution**: Consists of one or more **Result**s. Each **Result** consists of one or more **Formula**s. For example, listing 1 is one of the results and formulas for <AnalyticalPerformanceModel Name="Black Box M-G-1 Single Queue Model">:
- **Formula**: Consists of:
  - **Parameter**: one or more **InputParameter**s and **Metric**s grouped in a **ParameterList**. Parameters in the **ParameterList** can have **Constraint**s that are used when solving the model.
  - **ImplementationCode**: Has attributes ProgrammingLanugage, CodeOrFileReference and the actual Code, which can be the actual solution code, a reference to a solution subroutine or a reference to a solver.
  - Choice of **MathematicalComputation** or **AlgorithmicComputation**
  - **Type**: Whether the formula is an Exact, product-form solution or Approximate
- **MathematicalComputation**: This is the text of the solution's mathematical formula while the actual implementation is referenced in **ImplementationCode**

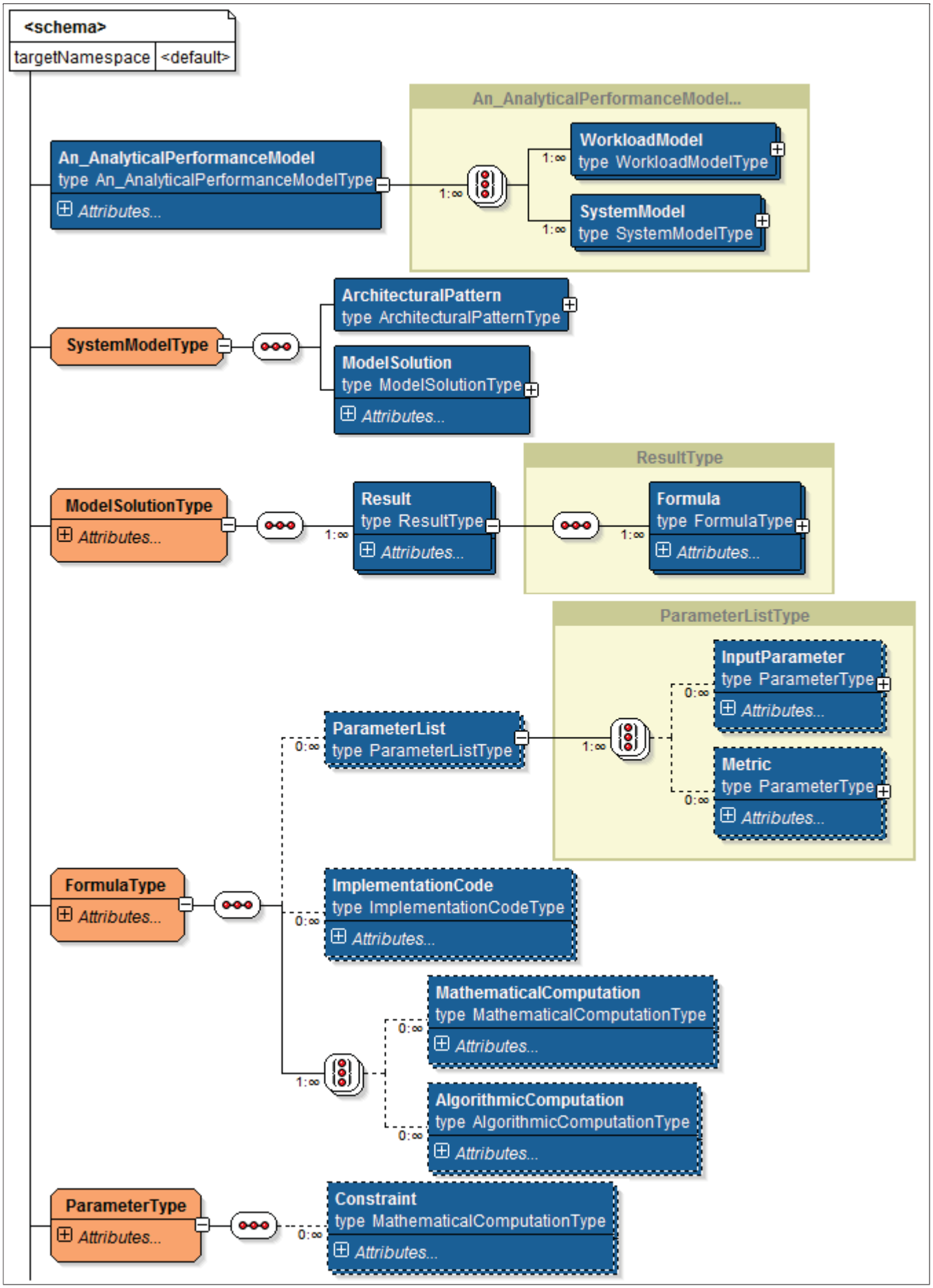

Figure 2: XML Schema for Model Identification (Partial)

Listing 1: Excerpt of M/G/1 Model XML Document

```
<ModelSolution Name="Black Box M/G/1 Single Queue Model Solution">
 <Result ID="1" Name="T: Average Response Time">
 <Formula ID="1" Name="T Formula 1" Type_="Exact">
  <ParameterList>
  <Metric ID="1" Name="average response time"
    DataType_="FLOAT" MathematicalSymbol="T" Type_="Output"/>
  <InputParameter ID="2" Name="average service time of requests"
    DataType_="FLOAT" MathematicalSymbol="x_bar" Type_="Input">
   <Constraint Name="average service time constraint" FormulaText="x_bar>0"/>
  </InputParameter>
  <InputParameter ID="3" Name="average arrival rate of requests"
    DataType_="FLOAT" MathematicalSymbol="lambda" Type_="Input">
   <Constraint Name="utilization constraint" FormulaText="lambda*x_bar&lt;1"/>
  </InputParameter>
  <InputParameter ID="4" Name="coefficient of variation of the service time distribution"
   DataType_="FLOAT" MathematicalSymbol="C_s" Type_="Input">
   <Constraint Name="coefficient of variation constraint" FormulaText="C_s>=0"/>
  </InputParameter>
  </ParameterList>
  <ImplementationCode Name="T_Implementation_Code"
   ProgrammingLanugage="Math" CodeOrFileReference="Code"
   Code="T=findMG1ResponseTime(x_bar,lambda,C_s)"/>
  <MathematicalComputation Name="T_Mathematical_Computation"
    FormulaText="T=x_bar+(((lambda*x_bar)*x_bar*(1+(C_s^2))/(2*(1-(lambda*x_bar))))"/>
 </Formula>
 </Result>
</ModelSolution>
```

- **AlgorithmicComputation**: This is the text of the solution's algorithm while the actual implementation is referenced in **ImplementationCode**

4. MODEL IDENTIFICATION AND VALIDATION

The Performance Model Analyzer implements a search process in which the system's observed characteristics are used to search the repository of well-known performance models in the KB to find the best fit model. Including the model solutions in the repository allows us to automate the process of solving the model and calibrating it as needed in order to find the most accurate model at an acceptable cost.

We illustrate now the technique we use to fit an analytic model by using the KB. Assume for the sake of this example that we are interested in single queue models with $c \geq 1$ servers such as in M/M/1, M/G/1, M/M/c, G/G/1, or G/G/c [17]. For these models, the average response time T can be expressed as

$$T = f(\lambda, C_a, \bar{x}, C_s, c) \tag{1}$$

where $f()$ is a function of λ (the average arrival rate of requests), C_a (the coefficient of variation, i.e., standard deviation divided by the mean, of the interarrival time of requests), $\bar{x}$ (the average service time of requests), and c (the number of parallel servers). Using this notation, the function $f()$ for an M/G/1 queue is

$$T_{\mathrm{M/G/1}} = \bar{x} + \frac{\rho\ \bar{x}\ (1 + C_s^2)}{2\ (1 - \rho)} \tag{2}$$

where $\rho = \lambda\ \bar{x}$. An approximate expression for M/G/c is given by

$$T_{\mathrm{M/G/c}} \approx \bar{x} + \frac{C_s^2 + 1}{2} . \frac{C(\rho, c)}{c\ (1 - \rho)/\bar{x}} \tag{3}$$

where $\rho = \lambda\ \bar{x}/c$ and

$$C(\rho, c) = \frac{(c\rho)^c/c!}{(1 - \rho) \sum_{n=0}^{c-1} (c\rho)^n/n! + (c\rho)^c/c!}. \tag{4}$$

Our method for fitting a performance model to a model in the KB is presented below.

- Input: $\mathcal{S} = \{(t_i, T_i)\}$, for $i = 1, \cdots, n$, which is an ordered set of transaction arrival instants, t_i, and their response times T_i.

- Step 1: Process the set $\mathcal{S}$ and compute the average arrival rate λ as the inverse of the average interarrival times:

$$\lambda = \left[\frac{\sum_{i=1}^{n-1} (t_{i+1} - t_i)}{n - 1} \right]^{-1} \tag{5}$$

- Step 2: Compute $C_a = \sigma_a/(1/\lambda)$ where σ_a, the unbiased sample standard deviation of the interarrival times, is computed from $\mathcal{S}$ as

$$\sigma_a = \sqrt{\frac{\sum_{i=1}^{n-1} [(t_{i+1} - t_i) - 1/\lambda]^2}{n - 2}} \tag{6}$$

- Step 3: Determine the type of arrival process (InterarrivalDistribution). If $C_a \approx 1$ then InterarrivalDistribution = Exponential (i.e., the arrival process is Poisson because the interarrival times are exponentially distributed. This is the case because the exponential distribution is the only continuous distribution with a coefficient of variation equal to 1. Otherwise, InterarrivalDistribution = General.

- Step 4: Find all models in the KB that have AP as an arrival process. This can be done using XQuery as in the example below that retrieves all models in the KB that have exponentially distributed interarrival times.

```
xquery version "3.0";
for $x in
 collection('/db/PhD_XML_DB/All_Models?select=
                *.xml')
          /An_AnalyticalPerformanceModel
where $x/WorkloadModel/InterarrivalDistribution =
               "Exponential"
return <Model>
           {$x/@Name}
       </Model>
```

  This query would return for example the following single queue models.

```
<Model Name="Black Box M/M/1 Single Queue Model"/>
<Model Name="Black Box M/M/c Single Queue Model"/>
<Model Name="Black Box M/G/1 Single Queue Model"/>
<Model Name="Black Box M/G/c Single Queue Model"/>
```

- Step 5: Repeat the following sub-steps for each model m retrieved in the previous step:

  - Step 5.1: Determine the set of parameters for model m from the XML document retrieved for model m (see e.g., Listing 1 for M/G/1). For example, for M/M/1 we need λ and $\bar{x}$. For M/G/1, the needed parameters are λ, $\bar{x}$, and C_s.
  - Step 5.2: Divide the input sequence $\mathcal{S}$ into n subsequences $\mathcal{S}_i, i = 1, \cdots, n$ and compute the values of the average interarrival rates, λ_i, and average response time, $\bar{T}_i$, for subsequence i.
  - Step 5.3: Use an optimization model to estimate the values of the missing parameters similarly to what was done by the authors in [16, 2]. We provide two examples here. Consider first the case of an M/M/1 performance model; the computed average response time, T_i, for sub-sequence i is

$$T_i = \frac{\bar{x}}{1 - \lambda_i.\bar{x}}. \tag{7}$$

    Note that $\bar{x}$, the average service time, does not change with the arrival rate λ_i in sub-sequence i. The measured average response time in subsequence i is $\bar{T}_i$. Then, the error ξ_i for subsequence i is $T_i - \bar{T}_i$.

    In order to find the value of $\bar{x}$, we solve the optimization problem below, which finds the value of the average service time $\bar{x}$ that minimizes the sum of squared errors subject to the constraint that the sum of the errors is zero along with the additional constraints such as the average service time $\bar{x}$ has to be greater than zero and the server utilization $\lambda\ \bar{x}$ has to be less than 1.

$$\begin{aligned} \text{Minimize} \quad & \sum_{i=1}^{n} \xi_i^2 \\ \text{s.t.} \quad & \sum_{i=1}^{n} \xi_i = 0 \\ \bar{x} \quad & > \quad 0 \\ \lambda\bar{x} \quad & < \quad 1 \end{aligned} \tag{8}$$

    We show two examples: one for M/M/1 and another for M/G/1. We used as input a sequence $\mathcal{S}$ of 200 values of arrival instants of requests to a server and the response times for these 200 requests. For M/M/1, the expression for ξ_i is

$$\xi_i = \frac{\bar{x}}{1 - \lambda_i\bar{x}} - \bar{T}_i. \tag{9}$$

    The result of solving the optimization problem using a GRG Nonlinear solver yields a value of $\bar{x} = 1.29$ and a sum of square errors equal to$\sum_{i=1}^{n} \xi_i^2 = 142.7$.

    For M/G/1, the expression for ξ_i is

$$\xi_i = \left[\bar{x} + \frac{\lambda_i\bar{x}\bar{x}(1 + C_s^2)}{2(1 - \lambda_i\bar{x})}\right] - \bar{T}_i. \tag{10}$$

    To solve the minimization problem, we need to add the constraint that $C_s \geq 0$. The solution to this optimization problem yields $\bar{x} = 1.09$, $C_s = 2$, and a sum of square errors equal to $\sum_{i=1}^{n} \xi_i^2 = 33.98$.

- Step 6: Select the model that fits best the data as the one with the smallest value of $\sum_{i=1}^{n} \xi_i^2$, which in the case of the example above is the M/G/1 model with parameters $\bar{x} = 1.09$ and $C_s = 2$.

The example above assumes that there is a closed form expression for Eq. (1). This is true for single queue models and for open queuing networks. Awad and Menascé showed how to deal with this problem in the case of open queuing networks [2]. However, this is not the case for closed queuing networks, which instead of a closed form expression for obtaining their solution, have to be solved by computational recursive algorithms such as in Mean Value Analysis (MVA) or by iterative algorithms such as in Approximate Mean Value Analysis (AMVA) [17]. In this case, the function f in Eq. (1) is computed by executing an algorithm and the objective function in the optimization problem given in Eq. (8) does not have a closed form.

However, this does not pose a problem. There are nonlinear solvers that take objective functions whose value is computed by executing an algorithm. An example of that is MathWork's `fmincon` function
(see www dot mathworks dot com/help/optim/ug/fmincon.html).

5. RELATED WORK

In [24], the authors introduced PMIF 2.0, which is defined and implemented using an XML while its predecessor,

PMIF 1.0, was defined using the Electronic Industries Association/CASE Data Interchange Format (EID/CDIF) and implemented using LISP [23]. The authors described the PMIF 2.0 XML schema and the QN meta model on which it was based, and highlighted its improvements over its predecessor model exchange format. The viability of PMIF 2.0 was demonstrated by exchanging a bank ATM performance model (with two different workloads) between the modeling tools SPE.ED and Qnap. A few extensions to PMIF 2.0 were suggested, which included adding additional service time distributions, server queuing disciplines and priorities, among others.

Subsequent publications by the authors of PMIF 2.0 introduced a number of schema extensions including the Experiment, Output and Result extensions, among others. In [25], the PMIF Experiment schema extension allows modelers to define multiple model runs using different workloads while identifying the individual output metrics desired from those runs. The extension allows for the specification of input variables with multiple values and output variables that represent the desired metrics, such as CPU utilization. It also allows modelers to specify iterations, variable range values, solution types and final experiment output specifications. The authors focused on transforming experiment results into table and chart outputs in .xls and LaTeX formats. The authors implemented a previously published open model experiment using Java and Apache POI to transform results and to create MS Excel output and XSLT to transform PMIG to Qnap's internal modeling format.

In [21], the authors implemented the same workload characterization algorithm described in [19, 18] for the purpose of generating similar workloads in stress testing tools (i.e., load testing) to answer what-if questions. The authors developed a graphical user interface to aid in producing the CBMGs and generating stress testing workloads. Although the authors did not highlight the technology stack used to implement their tool, we feel that the use of the ELK stack and Logstash in particular makes our approach more generic and more efficient. For example, in our experiments we implemented automated log parsing of Apache common log format as well as combined log format plus Tomcat access logs with a slight modification of the Grok filter. The use of Elasticsearch for storing log entries allows for fast search and processing, which is essential when parsing and analyzing multiple large log files. Finally, the purpose of our approach for parsing server logs is to reverse engineer the systems into their corresponding QN performance models. This requires that we parse system access logs as well as other system activity logs. The use of the ELK stack allows us to use a consistent tool set to parse and analyze system logs, and derive the corresponding performance model.

In [10], the authors presented an approach for automatically generating component-based performance models for running Java EE applications. In [12], the authors presented an automated process for performance prediction in autonomic systems using QNs. They decompose applications into black boxes, identify queuing models for each black box, assemble models into a QN model, and produce performance predictions. The authors focused on the black box identification.

Some of the prior work that tackled the parameterization of analytical models includes [6, 7, 8], where the problem of estimating known model parameters is treated as an optimization problem that is solved using derivative-free optimization. The objective function to be optimized is based on the distance between the observed measurements and the corresponding points derived from the model. The authors point out that the main problem is determining how to couple these two sets of points in order to arrive at an objective function to be minimized. The proposed approach is applied to a small set of single queue models.

Menascé tackled the issue of model parameterization when some input parameters of an open queuing network are already known [16]. The author proposed a closed-form solution for the case when a single service demand value is unknown, and a constrained non-linear optimization solution when a feasible set of service demands are unknown. However, he did not propose a solution when none of the service demands are known a priori. The approach in that paper assumes knowledge of the number of stations in the queuing network.

In [14], the authors presented a survey of performance modeling approaches focusing mainly on business information systems. The authors described the general activities involved in workload characterization, especially estimating service demands, and the various methods and approaches used to estimate it. Some of these methods include general optimization techniques, linear regression, and Kalman filters.

In [9], the authors presented a method for extracting architecture level performance models in distributed component-based systems using tracing information and instrumentation to infer system components, their connections and the (probabilistic) dependency of their parameters. In contrast, our approach does not require such knowledge of system components or their relationships.

In [20], the authors presented an iterative methodology for building performance models in virtualized environments with a focus on the I/O function of storage systems. The method implemented in that paper focused on the storage component of a particular IBM mainframe system. Our approach addresses a higher level of system abstraction where the internal structure of individual servers, such as the storage system, is unknown.

In [11], the authors presented Modellus; a system for automated web-based application modeling that uses workload characterization, data mining and machine learning techniques. The work in [5] used Kalman Filters to estimate resource service demands for the purpose of system performance testing. The authors attempted to find the workload mix that would eventually saturate a certain system resource in a test environment in order to determine the system's bottlenecks.

In [26], the authors proposed a method to more accurately match the transaction arrival rates with intervals of steady resource utilizations. By comparing the service demand measurements at different intervals of steady resource utilizations, one can continue to improve the service demand estimates. The Workload Modeler (WLM) tool parses application access logs for the sole purpose of finding arrival rates in order to compare them with server utilization logs. Our approach performs both syntactic analysis and semantic analysis by extracting complete workflows from the access logs. We focus on modeling application systems by analyzing the user interaction with the system and determining workloads accordingly as opposed to analyzing requests in-

dividually.

In [15], the authors propose an approach based on queuing networks to represent system configurations. They use symbolic analysis and satisfiability modulo theory (SMT) to find a model that fits continuous changes in run-time conditions.

6. CONCLUSIONS

We proposed a framework for performance model fitting in which operational systems are compared to a list of known performance models to find the best-fit model. In this paper, we focus on the internal structure of the knowledge base used to store well-known single-queue and queuing networks models. The knowledge base is used for model definition and identification (i.e., model fitting). We adopt an XML-based representation of analytical performance models similar to the Performance Model Interchange Format (PMIF), but extended it to support model solutions, in order to support our automatic model identification framework.

7. REFERENCES

[1] Awad, M. and Menascé, D. A., "Performance Model Derivation of Operational Systems Through Log Analysis," IEEE Intl. Symp. Modeling, Analysis and Simulation of Computer Systems and Telecommunication Systems (MASCOTS 2016), Imperial College, London, UK, September 19-21, 2016.

[2] Awad, M. and Menascé, D. A., "On the Predictive Properties of Performance Models Derived Through Input-Output Relationships", Proc. European Performance Engineering Workshop (EPEW 2014), 2014.

[3] Awad, M. and Menascé, D. A., "Dynamic Derivation of Analytical Performance Models in Autonomic Computing Environments", In: Computer Measurement Group Conference on Performance and Capacity, 2014.

[4] Awad, M. and Menascé, D. A., "Automatic Workload Characterization Using System Log Analysis", In: Computer Measurement Group Conference on Performance and Capacity, 2015.

[5] Barna, C., Litoiu, M., Ghanbari, H., "Autonomic Load-Testing Framework." In: Proceedings of the 8th ACM International Conference on Autonomic Computing, pp. 91–100, 2011.

[6] Begin, T., Brandwajn, A., Baynat, B., Wolfinger, B.E., Fdida, S., Towards an Automatic Modeling Tool for Observed System Behavior. In: Formal Methods and Stochastic Models for Performance Evaluation, pp. 200–212, 2007.

[7] Begin, T., Brandwajn, A., Baynat, B., Wolfinger, B.E., Fdida, S., "High-Level Approach to Modeling of Observed System Behavior. ''In: ACM SIGMETRICS Performance Evaluation Review, 35(3), pp. 34–36, 2007.

[8] Begin, T., Baynat, B., Sourd, F., Brandwajn, A., "A DFO Technique to Calibrate Queuing Models." In: Computers & Operations Research 37, no. 2, pp. 273–281, 2010.

[9] Brosig, F., Huber, N., Kounev, S.: "Automated extraction of architecture-level performance models of distributed component-based systems." In: 26th IEEE/ACM Intl. Conf. Automated Software Engineering (ASE), pp.183–192 (2011).

[10] Brunnert, Andreas, Christian Vogele, and Helmut Krcmar. "Automatic performance model generation for java enterprise edition (ee) applications." In Proc. 10th European Workshop, EPEW 2013, Venice, Italy, September 16-17, 2013.

[11] Desnoyers, P., Wood, T., Shenoy, P., Singh, R., Patil, S., and Vin, H.: "Modellus: Automated modeling of complex internet data center applications." In: ACM Tr. on the Web (TWEB) 6, no. 2 (2012).

[12] Harbaoui, Ahmed, Nabila Salmi, Bruno Dillenseger, and J-M. Vincent. "Introducing queuing network-based performance awareness in autonomic systems." In Autonomic and Autonomous Systems (ICAS), 2010 Sixth International Conference on, pp. 7-12. IEEE, 2010.

[13] J.O. Kephart and D.M. Chess, "The Vision of Autonomic Computing", IEEE Computer, 36(1), January 2003, pp. 41–50.

[14] Kounev, S., Huber, N., Spinner, S., Brosig, S., "Model-Based Techniques for Performance Engineering of Business Information Systems." In: Business Modeling and Software Design, Lecture Notes in Business Information Processing (LNBIP), Vol. 0109, pp. 19–37, 2012.

[15] Incerto, E., M. Tribastone, and C. Trubiani, "Symbolic Performance Adaptations" Proc. 11th Intl. Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2016), May 16-17, 2016, Autsin, TX.

[16] Menascé, D., "Computing Missing Service Demand Parameters for Performance Models." In: Proceedings of the Thirty-fourth Intl. Computer Measurement Group Conference, pp. 7–12, 2008.

[17] Menascé, D.A., Almeida, V.A.F., and Dowdy, L., "Performance by Design: Computer Capacity Planning By Example." Prentice Hall, 2004.

[18] Menascé, D.A. and Almeida, V.A.F., "Scaling for E-Business: technologies, models, performance, and capacity planning," Prentice Hall, 2000.

[19] Menascé, D.A., V.A.F. Almeida, R. Fonseca, and M. A. Mendes, "A Methodology for Workload Characterization of E-commerce Sites," In: 1999 ACM Conference on Electronic Commerce, Denver, CO, November, 1999.

[20] Noorshams, Q., Rostami, K., Kounev, S., Tuma, P., Reussner, R.: I/O Performance Modeling of Virtualized Storage Systems. In: IEEE 21st Intl. Symp. Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), pp.121–130 (2013)

[21] Ruffo, G. , R. Schifanella, M. Sereno, and R. Politi, WALTy: A User Behavior Tailored Tool for Evaluating Web Application Performance. In: Proceedings of the third IEEE International Symposium on Network Computing and Applications (NCA04), 2004.

[22] Singh, Rahul, Upendra Sharma, Emmanuel Cecchet, and Prashant Shenoy. "Autonomic mix-aware provisioning for non-stationary data center workloads." In: Proceedings of the 7th international conference on Autonomic computing, pp. 21-30. ACM, 2010.

[23] Smith, Connie U., and Lloyd G. Williams. "A performance model interchange format." Journal of Systems and Software 49.1 (1999): 63-80.

[24] Smith, Connie U., Catalina M. Lladó, "Performance Model Interchange Format (PMIF 2.0): XML Definition and Implementation Technical Report." Performance Engineering Services, Santa Fe, New Mexico (2004).

[25] Smith, Connie U., Catalina M. Lladó, and Ramón Puig-

janer. “How to Automatically Execute Performance Models and Transform Output Into Useful Results”. Computer Measurement Group (2009).

[26] Sudheendra, Suhas, Mitesh Patel, and Pratik Kumar. “Approach to build performance model for a web-based system from its application server logs.” In: Proc. 32nd International Computer Measurement Group Conference, 2006.

Appendix

This appendix (see next page) shows an excerpt of the XML schema for model identification and concentrates on the `ModelSolutionType` part of the schema. The complete schema can be found at
cs dot gmu dot edu/~menasce/KB_XML_Schema/.

Listing 2: Excerpt of XML Schema for Model Identification

```
<xsd:complexType name="ModelSolutionType">
  <xsd:sequence>
    <xsd:element name="Result" type="ResultType" minOccurs="1" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="Name" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="ResultType">
  <xsd:sequence>
    <xsd:element name="Formula" type="FormulaType" minOccurs="1" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="ID" type="xsd:string" use="required"/>
  <xsd:attribute name="Name" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="FormulaType">
  <xsd:sequence>
    <xsd:element name="ParameterList" type="ParameterListType" minOccurs="0" maxOccurs="
        unbounded"/>
    <xsd:element name="ImplementationCode" type="ImplementationCodeType" minOccurs="0"
        maxOccurs="unbounded"/>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element name="MathematicalComputation" type="MathematicalComputationType"
          minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="AlgorithmicComputation" type="AlgorithmicComputationType"
          minOccurs="0" maxOccurs="unbounded"/>
    </xsd:choice>
  </xsd:sequence>
  <xsd:attribute name="ID" type="xsd:string" use="required"/>
  <xsd:attribute name="Name" type="xsd:string" use="required"/>
  <xsd:attribute name="Type_" type="SolType_" use="required"/>
</xsd:complexType>
<xsd:complexType name="ParameterListType">
  <xsd:choice maxOccurs="unbounded">
    <xsd:element name="InputParameter" type="ParameterType" minOccurs="0" maxOccurs="
        unbounded"/>
    <xsd:element name="Metric" type="ParameterType" minOccurs="0" maxOccurs="unbounded
        "/>
  </xsd:choice>
</xsd:complexType>
<xsd:complexType name="ParameterType">
  <xsd:sequence>
    <xsd:element name="Constraint" type="MathematicalComputationType" minOccurs="0"
        maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="ID" type="xsd:string" use="required"/>
  <xsd:attribute name="Name" type="xsd:string" use="required"/>
  <xsd:attribute name="DataType_" type="xsd:string" use="required"/>
  <xsd:attribute name="MathematicalSymbol" type="xsd:string" use="required"/>
  <xsd:attribute name="Type_" type="ParType_" use="required"/>
</xsd:complexType>
<xsd:complexType name="ImplementationCodeType">
  <xsd:attribute name="Name" type="xsd:string" use="required"/>
  <xsd:attribute name="ProgrammingLanugage" type="xsd:string" use="required"/>
  <xsd:attribute name="CodeOrFileReference" type="ImplType_" use="required"/>
  <xsd:attribute name="Code" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="MathematicalComputationType">
  <xsd:attribute name="Name" type="xsd:string" use="required"/>
  <xsd:attribute name="FormulaText" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="AlgorithmicComputationType">
  <xsd:attribute name="Name" type="xsd:string" use="required"/>
  <xsd:attribute name="AlgorithmText" type="xsd:string" use="required"/>
</xsd:complexType>
```