Model Evolution by Run-Time Parameter Adaptation

Ilenia Epifani
Politecnico di Milano
Dipartimento di Matematica
Piazza L. da Vinci, 32
20133 Milano, Italy
ilenia.epifani@polimi.it

Carlo Ghezzi, Raffaela Mirandola,
and Giordano Tamburrelli
Politecnico di Milano
DeepSE Group at DEI
Piazza L. da Vinci, 32
20133 Milano, Italy
(ghezzi|mirandola|tamburrelli)@elet.polimi.it

Abstract

Models can help software engineers to reason about design-time decisions before implementing a system. This paper focuses on models that deal with non-functional properties, such as reliability and performance. To build such models, one must rely on numerical estimates of various parameters provided by domain experts or extracted by other similar systems. Unfortunately, estimates are seldom correct. In addition, in dynamic environments, the value of parameters may change over time. We discuss an approach that addresses these issues by keeping models alive at run time and feeding a Bayesian estimator with data collected from the running system, which produces updated parameters. The updated model provides an increasingly better representation of the system. By analyzing the updated model at run time, it is possible to detect or predict if a desired property is, or will be, violated by the running implementation. Requirement violations may trigger automatic reconfigurations or recovery actions aimed at guaranteeing the desired goals. We illustrate a working framework supporting our methodology and apply it to an example in which a Web service orchestrated composition is modeled through a Discrete Time Markov Chain. Numerical simulations show the effectiveness of the approach.

1. Introduction

Software engineers use models to reason about systems by abstracting from details. Models are especially useful in the design stage to drive architectural decisions that may affect the overall quality of the final systems. By reasoning on models, engineers may anticipate flaws that would otherwise percolate through the development process and lead to later costly corrective maintenance activities. Many modeling approaches have been proposed, and several of them are currently used in practice. They may differ —among others— in the kind of properties they help reason about and in the level of precision or formality of the results one may obtain through them.

In this paper, we focus on models that may be used to reason about non-functional properties of the software-to-be. Furthermore, we deal with models that may be used for automatic verification of certain desired properties. In particular, we focus on properties such as reliability and performance, and on modeling approaches based on Discrete Time Markov Chains (DTMCs) and on Queuing Networks (QNs), which can be verified by using a probabilistic model checker (e.g., PRISM, [22, 24]) or the suite JMT for QN modeling and workload analysis (e.g., [7, 8]). The key problem of models is *accuracy*. Intuitively, a model is accurate if the information the designer may extract by reasoning on it provides the right amount of detail and precision. In the case of non-functional requirements, models are heavily dependent on parameters that must be provided a-priori by domain experts or are extracted by other similar systems. For example, in modeling performance of a composite Web service that is built by orchestrating existing services, one needs to rely on estimates or on published data on quality of service (QoS) of the components, such as performance parameters. Unfortunately, estimates are seldom correct. In addition, many current large distributed systems change over time. For example, service-oriented architectures (SOAs) or pervasive systems are increasingly built as dynamically adaptable and evolvable aggregates of components that may change at run time. As a consequence, design-time assumptions, even if initially accurate, may later change after the system is deployed and even while it is running. We claim that, to deal with these issues, models must be kept alive at run time, and must be continuously refined to achieve increasingly better accuracy, by updating the relevant parameters. *A priori* parame-

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

ters may be updated by observing the real data at run time and through some strategy for estimate refinement which generates *a posteriori* values. We propose a Bayesian approach to address this problem.

This paper contributes to two related relevant problems. First, it lays the foundations for an iterative model-driven development, which aims at verifying that an implementation satisfies non-functional requirements. If the resulting running system behaves differently from the assumptions made at design time, the feedback to the model shows why it does not satisfy the requirements. This may lead to a further development iteration or, ideally, to self-repearing actions that may automatically generate an implementation, according to an autonomic computing [17] approach. Second, it provides a Bayesian technique to re-estimate probabilities, which can be applied to different formal models (such as DTMCs or QNs). In particular, a running example is used to illustrate the approach in the context of reliability modeling via DTMC [20]. The proposed run-time methodology and a prototype tool supporting it define the KAMI framework which is the main contribution of this paper. KAMI stands for *Keep Alive Models with Implementations*.

The remainder of the paper is organized as follows: Section 2 provides an extended description of the proposed approach. Section 3 describes an example we used to validate our approach and motivates its choice. Section 4 illustrates a model for our example based on DTMCs, which supports reasoning about reliability properties. Section 5 describes the KAMI approach for model evolution by run-time adaptation. Section 6 describes the KAMI support tool. Section 7 illustrates the simulations performed through KAMI on our example and reports the numerical results we obtained. Section 8 discusses related works. Section 9 concludes the paper describing the current limitations of our approach and future work.

2. The KAMI Approach

We observed that models are helpful in software design, because they allow system designers to integrate their previous experience, documentation, and measurements into models that can be analyzed to diagnose problems and explore competing alternatives. In particular, models support verification of compliance between different design choices and requirements. Different models are usually provided to reason about different quality attributes and, in particular, analyze functional and non-functional requirements. For example, models for non-functional properties can be used to predict and validate software performance or reliability, as shown in [5] and [2]. Our proposal focuses on this category of models, which includes queueing networks, Markov chains, Bayesian networks etc.

Models for non-functional properties are characterized by and depend on numerical parameters. Let us consider, for example, a component based system modeled through a QN [9] for performance analyses. Specifying a QN requires several parameters: (1) customer interarrival time distribution (CITD), (2) service time distribution (STD), and (3) routing probabilities (RP), which are usually unknown at design time. Consequently, software engineers rely on estimates of these parameters provided by domain experts or extracted by previous versions of the system under design. However, there is no guarantee that these values are correct or still hold in the environment in which the system will be deployed. As stated before, system designers validate a model against desired requirements and drive the implementation following the model structure. If the parameters do not correspond to reality, the software will not exhibit the predicted performance, leading to unsatisfactory behaviors or failures.

Run-time adaptation of non-functional properties comes into play solving this issue. Since predicted system behaviors could differ from the actual one, or they may change over time because of changes in the environment, we claim that models for non-functional requirements should coexist with the implementation at run time. It is thus possible to feed models with run-time data to update their internal parameters. Consequently, modified models provide increasingly more accurate descriptions and allow us to automatically check the desired requirements while the system is running. At this stage, our approach can only deal with model evolution by continuous estimation of its numerical parameters. The parameters we can estimate through our method represent the actual values of the relevant non-functional characteristics of the system under design (e.g., reliability of an external component) and of the usage profile (e.g., customer interarrival time distribution). Future work will address the important open problem of automating additional more complex modifications to the model, such as structural changes.

The advantages of this methodology (KAMI) are twofold. First, updated models better capture real system behaviors. Second, updated models evolve at run time following the changes in the environment. In both cases when a model shows that a given requirement is violated, it is possible to react by triggering reconfigurations. Moreover, the more data we collect from the running instances of the system, the more precise our models will be. Indeed, model parameters will eventually converge to real values characterizing the modeled system as we will show in the sequel. Conceptually, KAMI establishes a *feedback control loop* between models and implementation. At design time, models are developed to verify non-functional requirements and drive the implementation. At run time, the real system provides data exploited as *feedbacks* that may update the model, increasing its correspondence with reality and our

confidence in it. It is important to notice that, in KAMI, it is not strictly necessary to model the whole system, but only the sub-parts that are considered as critical.

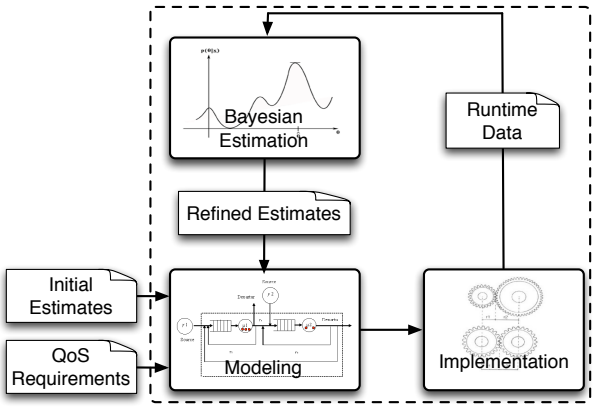


Figure 1. Methodology Scheme

A crucial factor of KAMI is the mechanism adopted to transform run-time data extracted by running instances of the implemented system into estimates of model parameters. KAMI performs this task by exploiting Bayesian Estimation Theory [6]. An informal explanation that justifies this approach is given in Section 5.1.

Summing up, let us consider again the example of a component based system modeled with a QN. When the system has been completely developed, tested, and deployed it is possible to collect data from its running instances. We can measure, for example, the customer interarrival time (CIT) and through the Bayesian estimation we can estimate its distribution (CITD). Consequently, the QN model is updated and checked at run time against the desired requirements.

3. A Running Example

This section illustrates a running example, which deals with Web-service compositions, used in this paper to illustrate the KAMI approach. Web-service compositions (and SOAs in general [28]) make an excellent case for the need of keeping models alive at run time. A Web-service composition is an orchestration of Web services aimed at building a new service by exploiting a set of existing ones. The orchestration is performed through a *workflow language*, such as BPEL [1, 10], a de-facto standard. BPEL instances coordinate services that are typically managed by external organizations, other than the owner of the service composition. This distributed ownership implies that the final functional and non-functional properties of the composed service rely on behaviors of third-party partners that influence the obtained results. At design time, a model can be used to guarantee that the QoS of a composite service satisfies the requirements, based on the hypothesized QoS of each composed external service. However, design-time verification does not suffice. The declared QoS of composed services may turn out not to be met in practice. In addition, because of the decentralized nature of services and of multiple ownership, external services may undergo independent and unanticipated changes, which may lead to violating the global QoS requirements.

The running example we use in the paper is based on a case study, illustrated in [3], which deals with a distributed system for medical assistance. The application, called Tele-Assistance (TA), consists in a BPEL process for remote assistance of patients. Figure 2 illustrates the application, in which a server runs the TA composite service. The description is provided graphically. A summary of BPEL constructs and the graphical notation we use to describe them are summarized in the Appendix.

The process starts as soon as a Patient (PA) enables the home device supplied by TA, which sends a message to the process' receive activity *startAssistance*. Then, it enters an infinite loop: every iteration is a pick activity that suspends the execution and waits for one of the following three messages: (1) *vitalParamsMsg*, (2) *pButtonMsg*, or (3) *stopMsg*. The first message contains the patient's vital parameters that are forwarded by the BPEL process to the Medical Laboratory service (LAB) by invoking the operation *analyzeData*. The LAB is in charge of analyzing the data and replies by sending a result value stored in a variable *analysisResult*. A field of the variable contains a value that can be: *changeDrug*, *changeDoses* or *sendAlarm*. The latter message triggers the intervention of a First-Aid Squad (FAS) composed of doctors, nurses, and paramedics, whose task is to visit the patient at home in case of emergency. To alert the squad, the TA process invokes the operation alarm of the FAS. The message *pButtonMsg* caused by pressing a panic button also generates an alarm sent to the FAS. Finally, the message *stopMsg* indicates that the patient may decide to cancel the TA service.

4. Reliability Modeling via DTMCs

Different models may be used to reason about different non-functional properties of a software architecture. All such models require that certain parameters characterizing the final running system should be specified. Although the KAMI methodology and its prototype implementation apply to any probabilistic non-functional quality attribute, hereafter we focus on reliability [21, 20] and on models based on DTMCs. KAMI also supports performance analysis via QNs. Run-time adaptation of QN parameters can be performed by applying the same statistical machinery we illustrate for DTMCs. The next section introduces DTMCs.

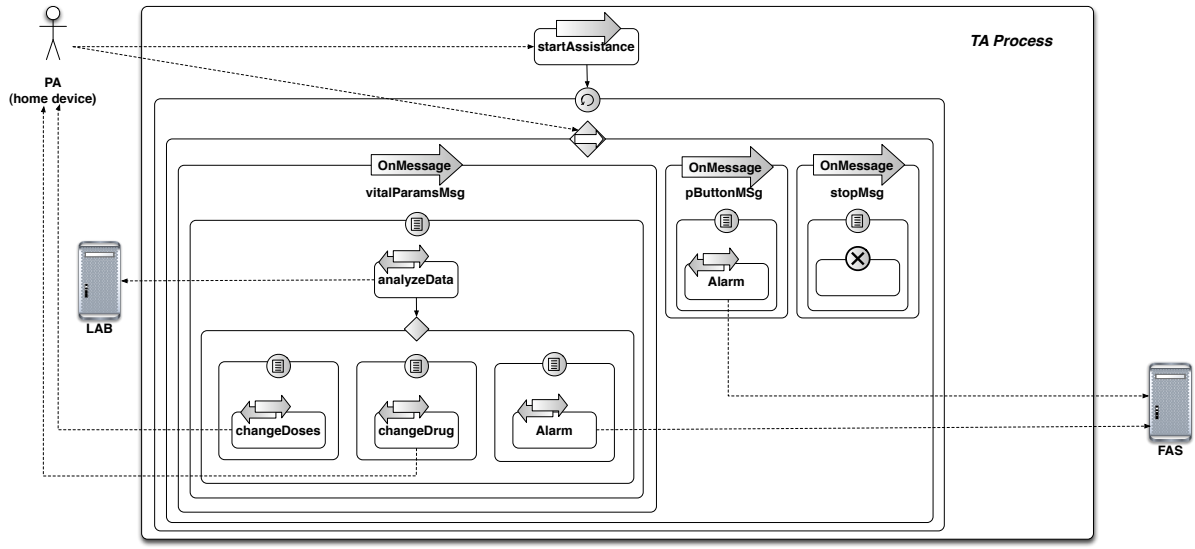


Figure 2. TA BPEL Process

Then we illustrate how one can model reliability aspects of the running example. Finally, we present a Bayesian technique for estimating DTMC numerical parameters by analyzing run-time data.

4.1. An Introduction to DTMCs

DTMCs are usually adopted to model system reliability [19]. They are stochastic processes with the Markov property defined as state-transition systems augmented with probabilities. *States* represent possible configurations of the system. *Transitions* among states occur at discrete time and the probability of making transitions is given by discrete probability distributions. Formally, a sequence of random variables $X_0, X_1, \ldots$ is a DTMC with tuple (S, s_{init}, M, L) if probabilities satisfies the following constraints:

$$P(X_0 = s_{init}) = 1$$

and

$$\begin{aligned} P(X_{n+1} = s' | X_n = s, X_1, \ldots, X_{n-1}) = \\ = P(X_{n+1} = s' | X_n = s) = m_{s,s'}, \end{aligned} \quad (1)$$

where

- S is a finite set of states: $S = \{1, \ldots, k\}$;
- $s_{init} \in S$ is the initial state;
- $\mathbf{M} : S \times S \rightarrow [0, 1]$ is a transition probability matrix; its element $m_{s,s'}$ represents the probability of passing from state *s* to state *s'* and $\sum_{s' \in S} m_{s,s'} = 1$;
- $L : S \rightarrow 2^{AP}$ is a function labelling states with atomic propositions.

Markov property (1) means that the probability of choosing the transition from state *s* to *s'* is independent of the past transitions. Moreover, if we observed a path $X_1, \ldots, X_n$ of a DTMC of length n, the likelihood function is:

$$\begin{aligned} P(X_1 = s_1, \ldots, X_n = s_n) = \\ = m_{s_1,s_2} m_{s_2,s_3} \ldots . m_{s_{n-1},s_n} = \prod_{i,j \in S} m_{i,j}^{N_{i,j}} \end{aligned}$$

where $N_{i,j}$ denotes the number of transitions between states i, j. A complete description of DTMCs is beyond the scope of this paper and can be found in [26]. When software engineers adopt DTMCs to model system reliability they must specify the set of states *S* and the transition probability matrix $\mathbf{M}$. Its probability values are numerical parameters and represent, for example, the failure probabilities of the system components. The modeling activity implies the choice of values for the probability matrix and engineers rely on an initial estimate $\mathbf{M}^{(0)} = \{m^{(0)}_{s,s'}\}_{s,s' \in S}$ called *prior transition probability matrix*. As we will describe in Section 5.1, $\mathbf{M^{(0)}}$ represents initial knowledge of an engineer and drives the initial design and system implementation.

4.2. Modeling the Example

In our running example we are interested in assessing the reliability of the process implementing the TA application. Following our approach, the designer is in charge

of: (1) deciding the structure of the model, (2) characterizing the model with appropriate parameters that represent his/her current knowledge about the system, (3) checking whether the model satisfies the desired requirements (and possibly changing design choices based on the outcome of the check), and (4) implementing the final BPEL process.

We adopted a DTMC to model reliability of our example. Figure 3 illustrates the result of our modeling activity (the prior transition probability matrix $\mathbf{M}^{(0)}$ can be easily derived from the figure). The model represents the structure of the TA process and assigns probabilities to branches and failure probabilities to service invocations. Our approach relies on initial estimates for reliability values that come from domain experts or extracted by previous or similar version of the system under design. In this example, we adopted numerical values chosen for illustrative purposes. Real-world medical applications would require lower failure probabilities. We assume that the system designer is

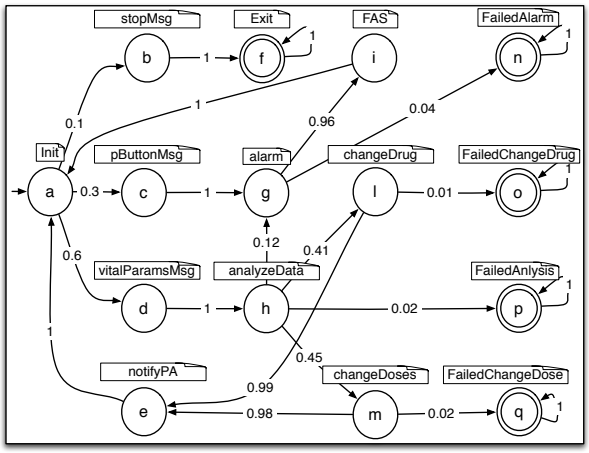


Figure 3. TeleAssistance DTMC Model

interested in verifying the following requirements:

- R1: *The probability P1 that no failures ever occurred is greater then* = 0.7
- R2: *If a changeDrug or a changeDoses has occurred the probability P2 that the next message received by the TA generates an alarm which fails (i.e., the FAS is not notified) is less then 0.015*
- R3: *Assuming that alarms generated by pButtonMsg have low priority while alarms generated by analyzeData have high priority, it is required that the probability P3 that a high priority alarm fails (i.e., it is not notified to the FAS) is less then 0.012*

The requirements can be proven to hold for the composite service. For example, by using the DTMC probabilistic model checker PRISM [22, 24], we obtained: *P1*=0.741, *P2*=0.014, *P3*=0.0048.

5. KAMI at Work for DTMCs

In this section we describe the statistical machinery for our proposed Bayesian estimation of DTMCs' parameters, which is supported by KAMI. We also show how the estimation may lead to discovery or prediction of failures.

5.1. Parameter Estimation for DTMCs

Let us consider a system modeled through a DTMC *H*. The focus of run-time adaptation is to exploit a statistical technique that estimates the matrix $\mathbf{M}$ of H, given run-time data and prior transitions $\{m^{(0)}_{s,s'}\}_{s,s'}$. Through run-time monitoring, we assume that information about the occurrence of every transition from state *i* to state *j* is available as en event trace. Run-time monitoring is beyond the scope this paper; the approach described in [4] can be used for this purpose. Let d be the number of running instances of the system modeled through d statistically independent DTMCs, all starting from a common initial state s_{init}. For every $h = 1, \ldots, d$, the h^{th} instance executes the transition from state i to state j $N^{(h)}_{i,j}$ times.

In a Bayesian perspective the transition matrix $\mathbf{M}$ is a random matrix and the statistical problem of updating each $m_{s,s'}$, using run-time data, corresponds to updating the *prior distribution* of $\mathbf{M}$ (depending on $\{m^{(0)}_{s,s'}\}_{s,s'}$) by computing a *posterior* conditional probability of $\mathbf{M}$, given the run-time data:

$$P(m_{s,s'}|N^{(h)}_{i,j}, m^{(0)}_{i,j},\ i,j \in S, 0 \leq h \leq d)\ .$$

Then the *posterior* distribution leads to an new estimate of $\mathbf{M}$. Hence, the Bayesian solution of updating $\mathbf{M}$ requires a statistical model (Likelihood Function) and a prior distribution of $\mathbf{M}$. It yields a posterior distribution from which we derived an updated transition matrix through an updating rule. A complete description of the mathematical steps involved in this process is beyond the scope of this paper. So, we confine our-selves to a very brief overview and refer the reader to [25, 32] for more details.

The likelihood function of collected data is

$$P(X^{(1)}_1 = s^{(1)}_1, \ldots, X^{(d)}_{n_d} = s^{(d)}_{n_d}) = \prod_{h=1}^{d} \prod_{i,j=1}^{k} m_{i,j}^{N^{(h)}_{i,j}}$$

Regarding the prior distribution of $\mathbf{M}$, we assume statistical independence among the rows of $\mathbf{M}$ and model each row $(m_{i,1}, \ldots, m_{i,k})$ with a *Dirichlet Distribution*[1]. In general, a Dirichlet distribution $Dir(a_1, \ldots, a_k)$, with parameters $a_1, \ldots, a_k$ all positive, is a joint distribution for a vector

[1]The choice of the Dirichlet distribution is justified by [12], which

$Y_1, \ldots, Y_k$ such that $\sum_{i=1}^{k} Y_i = 1$ and whose joint density evaluated in $y_1, \ldots, y_k$ is

$$\frac{\Gamma(c)}{\prod_{j=1}^{k} \Gamma(a_j)} y_1^{a_1-1} \cdots y_k^{a_k-1}, \quad c = \sum_{j=1}^{k} a_j$$

The j^{th} component Y_j has mean $\mathrm{E}(Y_j) = a_j/c$ and variance $\mathrm{Var}(Y_j) = a_j(c - a_j)/[c^2(c + 1)]$. The parameter c is called *total mass* of the Dirichlet distribution and is a precision parameter that regulates the concentration of each Y_j around its mean value since the variance varies inversely with c (see Section 3.1 in [18] for an full description). We model the prior distribution of the i^{th} row of $\mathbf{M}$ as a Dirichlet vector, with parameters $c_i^{(0)} m_{i,1}^{(0)}, \cdots, c_i^{(0)} m_{i,k-1}^{(0)}$ so that the total mass is $c_i^{(0)}$:

$$(m_{i,1}, \ldots, m_{i,k}) \sim Dir(c_i^{(0)} m_{i,1}^{(0)}, \cdots, c_i^{(0)} m_{i,k-1}^{(0)}).$$

The meaning of $c_i^{(0)}$ in our context and the criteria to choose it will be described later. By applying Bayes rule:

$$Posterior \propto Prior \times Likelihood$$

we obtain that *a posteriori* the transition matrix $\mathbf{M}$ is a product of independent Dirichlet rows with the following updated parameters:

$$c_i^{(N_i)} = c_i^{(0)} + N_i$$
$$m_{i,j}^{(N_i)} = m_{i,j}^{(0)} + \sum_{h=1}^{d} N_{i,j}^{(h)}$$

where: $N_i = \sum_{h=1}^{d} \sum_{j=1}^{k} N_{i,j}^{(h)}$.
Finally, we summarize the posterior distribution in a simple way, using the posterior means of $\{m_{i,j}\}_{i,j}$:

$$m_{i,j}^{(N_i)} = \frac{c_i^{(0)}}{c_i^{(0)} + N_i} \times m_{i,j}^{(0)} + \frac{N_i}{c_i^{(0)} + N_i} \times \frac{\sum_{h=1}^{d} N_{i,j}^{(h)}}{N_i} \quad (2)$$

Formula (2) is the *updating rule* which produces the new estimates. It is the weighted sum of two terms. The former is related to our initial estimate $m_{i,j}^{(0)}$ and represents our *a priori* knowledge. The latter term depends on run-time data and expresses the knowledge we extract from data collected from running instances. Parameters $\{c_i^{(0)}\}_i$ are called *smoothing parameters*: they quantify our confidence in a priori knowledge with respect to run-time data. A high value of $c_i^{(0)}$ means high confidence on a priori knowledge and in (2) the contribution of the data to the estimate is small. Vice-versa a low value of $c_i^{(0)}$ represents low confidence on a priori values and the run-time data are dominant. Notice that for $c_i^{(0)} \simeq 0$, the estimate $m_{i,j}^{(N_i)}$ in Formula (2) reduces to the classical *Maximum Likelihood Estimator* (MLE): $\sum_h N_{i,j}^{(h)} / \sum_h N_i^{(h)}$.

proves that, in a multinomial model, the prediction of a future event (in our case $m_{i,j}$) is linear in the number of past occurrences (in our case $N_{i,j}^{(h)}$) if and only if the prior distribution is Dirichlet.

5.2. Failure Detection and Prediction

In our approach, we can distinguish between *failure detection* and *failure prediction*. A failure is detected if the user of the system experiences a deviation from the expected behavior described by a requirement. For example, let us consider the TA application in Figure 2 and let us focus on requirement R3. A failure of R3 can be detected only by considering the number of failed high priority alarms whose notification to the FAS fails over the total number of high priority alarms. On the other hand, a failure is predicted if the model is able assess that a requirement is violated even if the actual events involved in the violation did not occur yet. The distinction between detection and prediction is crucial and provides further justification for keeping models alive at run time, as shown the following example.

Consider a trace t of 20 run-time data $x_1, x_2 \ldots x_{20}$ each representing an alarm invocation, and suppose that x_5, x_{15}, and x_{20} represent invocations that fail. A violation of R3 is detected if and only if all invocations have high priority. Being active at run time, KAMI detects the failure. In fact consider the DTMC in Figure 3 and assume $c_i^{(0)} = 10$ for every row in $\mathbf{M}^{(0)}$. After observing trace t KAMI computes the following model updates:

$$m_{g,n} = 0.04 \times \frac{10}{20 + 10} + \frac{3}{20 + 10} = \frac{3.4}{30} \simeq 0.113$$

Thus, KAMI detects that the probability associated with transition from state g to state n was underestimated at design time. By using the updated estimate 0.113 instead of 0.04, the probability of the path from state h to g to n would be $P = 0.12 \times 0.113 = 0.014$, which is greater then 0.012, thus violating R3. In this case KAMI detects the failure after it actually occurred, since the event that caused the detection is instead perceived by the user as a deviation from a requirement. After detecting the failure, KAMI activates the exception associated with the violated requirement which can only performs after-the-fact recovery actions.

KAMI can also predict failures. For example, the same trace t would lead KAMI to predict a failure of R3 even if all alarms in the trace are low priority and therefore no failure has been detected yet. In fact, the updated estimate $m_{g,n}$ provides the same value no matter what the priority of alarms invocations are. The Bayesian approach, embedded in KAMI, exploits failures of low priority alarms to update the probability of transition from state g to state n, which

is implicitly involved in requirement R3. By extending the trace t the updated estimate will eventually converge to the real failure rate which was unknown at design time.

6. KAMI: a Framework for Run-Time Model Adaptation

This section illustrates our ongoing implementation of KAMI. KAMI is a plugin-based software composed of: *(1) Model Plugins, (2) System Models, (3) Input Plugins*. The plugin-based architecture can support the necessary extensions that will be later introduced to enrich the environment.

System Models are text files describing the models on which KAMI operates. These files contain: (1) model descriptions with numerical parameters that KAMI is in charge of updating, (2) the requirements the user is interested in, and (3) a set of exceptions to be raised when a requirement is violated. For example, such file can contain a description of a QN and several requirements (e.g., a threshold on average residence time or on average queue length) with their associated exceptions aimed at managing the violations.

Model Plugins provide to KAMI the ability to handle different and new models, by interpreting model files and their requirements. Moreover they are in charge of analyzing models with respect to requirements. If a model violates a requirement, the corresponding exception specified in the system model is raised. The current version of KAMI comes with two pre-installed plugins, which can manage DTMCs and QNs. The former exploits the probabilistic model checker PRISM [22, 24] to verify reliability requirements. The latter is based on JMT, an open source suite for QN modeling and workload analysis [7, 8].

Finally, *Input Plugins* provide to KAMI the ability to connect models with the run-time world in which the implemented system is running. The running system feeds the model with monitored data. For example, in the case of DTMCs, it provides information about the occurrence of transitions among states. The purpose of the plugin is to handle different input formats and protocols for run-time data (e.g, socket, RMI, etc.).

Software engineers are in charge of modeling their systems and deploying the models in KAMI. Every model plugin defines the syntax they have to comply with. Once a model is deployed, KAMI automatically starts updating it with data provided by the running systems. Since KAMI supports run-time adaptation for multiple models and systems concurrently, data are tuples $< s, m, p, d >$ where s identifies the originating running system, m identifies the target model, p identifies the estimated parameter (in the case of a DTMC the transition probability), and d is the collected run-rime value. Input plugins are in charge of transforming run-time data in this tuple format. This mechanism supports the integration of existing monitoring tools to extract data form running instances of systems, because one simply needs to develop an input plugin that transforms run-time data as required by KAMI. Moreover, input plugins introduce a decoupling layer between KAMI and running systems which allows integration and correlation of distributed and heterogeneous data collected from the environment. The output of our framework is a new set of models corresponding to the updated versions of deployed system models that result from Bayesian estimation. These updated models are analyzed continuously by model plugins to verify requirements. When a violation is detected, KAMI raises the associated exception, logs the event, and calls the appropriate *exception handler*. In the current version of our framework, it consists of user-defined Java code. This way a system modeler can attach to exceptions predefined reactions that trigger alarms or perform reconfigurations which close the control loop between the model and the implementation.

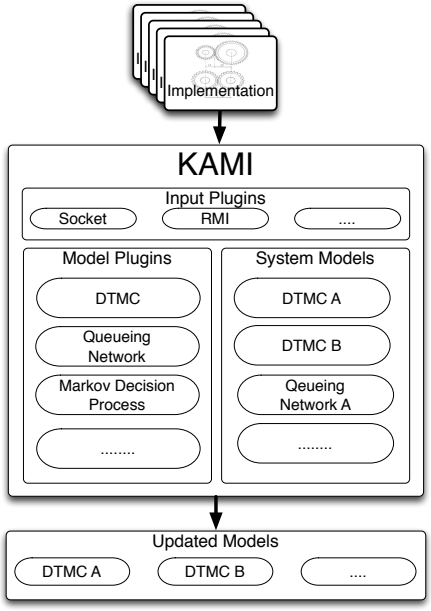


Figure 4. KAMI Architecture

7. Simulations and Numerical Results

This section describes how the KAMI approach has been evaluated through simulations. Of the experiments we made, here we focus on the TA example of Figure 2 and specifically on checking requirement R3. Let us initially assume that, at design time, the model contains a wrong guess of the value of the probability that an alarm fails (transition form state g to n in Figure 3 with probability 0.04).

Let us also assume that the actual probability is instead 0.15, which would lead to a requirement violation since the probability of the path from state h to g to n would be $P = 0.12 \times 0.15 = 0.018$, which is greater then 0.012.

The goal of simulation is to assess how the Bayesian estimate of the probability of the transition evolves over time, as more data are collected from the field. We achieve this goal by generating run-time data representing alarm invocations that follows a Bernoulli distribution with parameter $p = 0.15$ (failure probability). The results of simulations are shown in Figure 5. The figure represents the average estimate for the probability of transition form g to n (i.e., $m_{g,n}$) over 1000 simulations which use smoothing parameters $c_i^{(0)} = 10$. The horizontal axis represents the run-time data for the alarm invocations (i.e., N_g). The vertical axis represents the estimation value $m_{g,n}$, which starts from the prior value (i.e., 0.04) and gradually converges to the actual probability (i.e., 0.15). It is interesting to observe that a value of the probability equal to 0.1 is the threshold that indicates the violation for R3 [2] and the Figure 5 shows that a requirement violation occurs with less then 20 run-time data analyzed by KAMI.

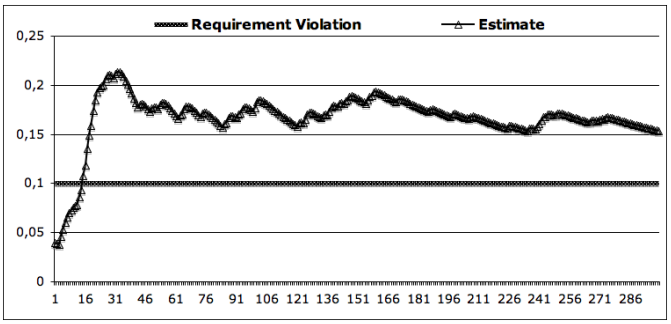


Figure 5. Average Estimate

Two natural questions arise from the previous simulations and require further investigations:

- we chose the value 10 for smoothing parameters in our simulations. How do different values for the parameter influence the results?
- we assumed that the probability to estimate is constant. What happens if the target environment changes in its operating conditions and therefore this probability evolves over time?

The next two sub-sections briefly address these issues.

[2] In fact the probability of the path from state g to h to n is $0.12 \times m_{g,n} < 0.012$, which implies $m_{g,n} < 0.1$.

7.1. Choosing the Smoothing Parameter

The effect of different choices for the values of the smoothing parameters can be measured through the estimation error (E) computed as:

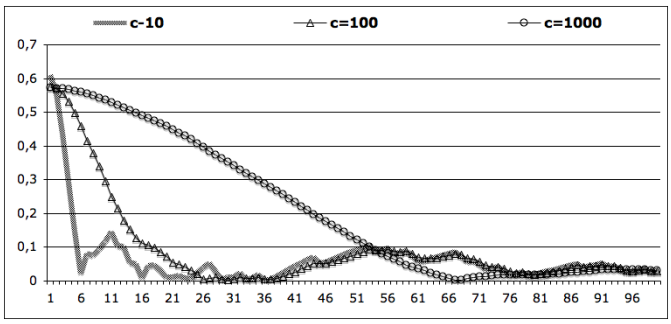
$$E = \frac{|\ \textit{estimated paramter-real parameter}\ |}{\textit{real parameter}}$$

We focus again on requirement R3. We assume, however, different values for the initially guessed probability (i.e., 0.3) and the real probability (i.e., 0.7) to put ourselves in a more extreme case, with a higher distance between the guessed and the real value than in the case of Figure 5. The parameter $c^{(0)}$ represents the system designer's confidence respect with a model parameter. The higher is the confidence, the larger $c^{(0)}$ must be. Figure 6(a) shows E for different choices of the smoothing parameter. It shows that lower values of $c^{(0)}$ imply a slightly faster convergence. However, in these cases E initially shows a less smooth convergence because of the strong dependence on initial run-time data. These fluctuations could lead to false positives in requirement verification (especially in the case of run-time data with high variance). Conversely, a larger value of $c^{(0)}$ shows that E converges slowly but with a smoother approximation.

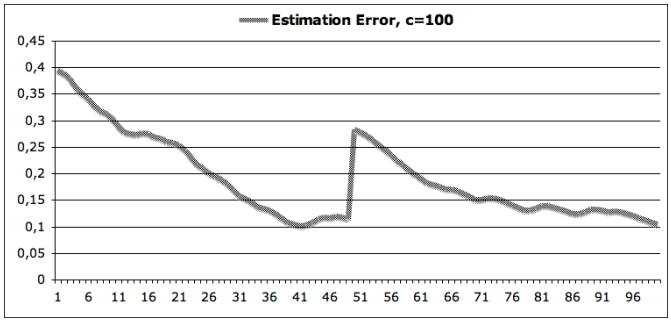
Summing up, in all our simulations estimates converge to real values of probabilities, but the speed of convergence depends on several factors. We investigated here the influence of smoothing parameters. Other simulations, which could not be reported here for space reasons, also show a dependence on the variance of run-time data.

7.2. Dynamic Environments

Let us consider a situation in which the target environment changes in its operating conditions and therefore the probability of failure of alarm notification evolves over time. In particular assume, in our example, that initially the probability of failure is equal to 0.7 and the prior guess is equal to 0.3. Further assume that a sudden a change in the running environment shifts the value of the probability to 0.3 (i.e. the initial guess). Figure 6(b) shows the results obtained with $c_i^{(0)} = 100$. The figure shows how the estimation error constantly decreases until there is a peak in the estimation error exactly when our simulation starts generating data from the new value 0.3. As soon as enough new run-time data are collected the estimation error starts decreasing again since the estimated parameter begins to converge to the new probability characterizing the new situation.



(a) Estimation Error with Different Smoothing Values



(b) Estimation Error in a Dynamic Environment

Figure 6. Estimation Errors

8. Related Work

Many techniques and methodologies support predictions or analyses of non-functional properties. Basically, two existing approaches are possible: (1) *measurement* and (2) *modeling*. The former is based on direct measurement of the desired requirement of an existing implementation through the use of dedicated tools (e.g., profiler, tracer, etc.). For example JMeter [23] performs profiling of Java applications aimed at identifying bottlenecks. Another example is Load Runner [15], which was conceived to perform load testing for scalability analyses. Data extracted help in identifying critical parts of the system that require a refinement to achieve the desired non-functional behavior. Direct measurements become increasingly harder as the complexity of the system get higher; for example, in the case of large distributed systems. Modeling comes into play to solve limitations of direct measurements because it may abstract away from the intricacies of systems. Moreover, a model may be built before a measurable system exists in reality. However, pure modeling of non-functional properties suffers from the defects illustrated in Section 1. As a consequence measurements and modeling are rather complementary than alternative techniques.

Concerning the application field of our example, many existing research efforts focus on modeling service compositions. However most of them focus only on functional properties. For example, [16] and [27] describe approaches which aim at verifying and validating service compositions by means of workflow analysis through model checking. However their approach does not explicitly take into account non-functional properties and it does not exploit run-time data to refine models. Similarly, [14] describes an approach for verifying service compositions starting from UML descriptions and then transforming them into a specific representation that allows validation with respect to concurrency properties. A similar approach is described in [13], which shows how to verify BPEL processes in case of resource constraints, with respect to safety and liveness properties.

The work in [4] focuses on monitoring and derives run-time data that are analyzed to perform verification through an assertion language. This approach is not based on an explicit model and does not support verification of non-functional properties. Further work from the same authors [3] defines a modeling approach for service compositions and an assertion language whose evaluation also extends to run time. The language, called ALBERT, can be used to specify both functional and simple (non probabilistic) non-functional properties. ALBERT assertions are verified for BPEL workflows at design time via model checking, and turned into dynamically evaluated assertions at run time, a feature that is essential to support evolution in dynamic environments. The approach described by [29] supports on-line monitoring of service level agreements (SLAs) in a web-service environment. A language (SLAng) is introduced to specify quality of service, which includes non-functional attributes, such as timeliness, reliability, and throughput. This approach is close to ours because a model, based on timed-automata, operates while messages are exchanged at run time. The main difference is in our focus on probabilistic properties and run-time model adaptation.

Our proposal guarantees both the benefits provided by approaches based on measurement and those based on modeling. Models are kept alive at run time and, through measurements, they can become progressively more accurate. Only few other similar approaches are described in the literature. In particular, [33] describes a methodology for estimation of model parameters through Kalman filtering. This work is based on a continuous monitoring that provides run-time data feeding a Kalman filter, aimed at updating the performance model. This approach differs from ours since it does not allow the encoding of initial knowledge that we provide through the smoothing parameter. Moreover, it does not explicitly support dynamic environments. Conversely, the approach is general with respect to the performance model, while in our proposal a specific statistical machinery has to be defined for every supported model and developed in KAMI through plugins.

The work in [30] developed a CTMC formulation of

composite services to predict performance and reliability bottlenecks by applying a sensitivity analysis technique. Although this work focuses on design time, we plan to include in our approach CTMCs for run time analyses.

A recent work [11] presents a framework for component reliability prediction whose objective is to construct and solve a stochastic reliability model allowing software architects to explore competing architectural designs. Specifically, the authors tackle the definition of reliability models at architectural level and the problems related to parameter estimation. The problem of correct parameter estimation is also discussed in [20, 31], where shortcomings of existing approaches are identified and possible solutions are proposed.

Concerning our Bayesian estimation technique, at the best of our knowledge, we do not know any existing approach that exploits this statistical technique to solve the problems presented in this paper. In [32], for example, a similar statistical approach was adopted in predicting word sequences for speech recognition and automatic translations. Moreover in [32] it is possible to find a complete description of the statistical concepts that we adopted in run-time adaption.

9. Conclusion and Future Work

In this paper we presented the KAMI approach to model evolution by run-time adaptation. Our proposal exploits Bayesian estimators which produce updated model parameters. The updated models provide an increasingly better representation of the system that allows continuous automatic verification of requirements at run time. Models updated at run time support failure detection and prediction, and may contribute to achieving self-adaptive autonomic systems.

Our contribution is twofold. We provided the statistical machinery to perform run-time adaptation of DTMCs. We also provided a working framework supporting the methodology. Our future work will consist of refining the KAMI approach investigating its scalability in real-world distributed applications. We plan to enrich the ongoing implementation by enlarging the set of supported models (e.g., Continuous Markov Chains, Markov Decision Processes, etc.) and defining a language aimed at managing multi-model consistency. We will also conduct further simulation campaigns to shed light on issues like the mutual effects of the choice of different values for smoothing parameters, and the distance between guessed and real values of parameters, and the variance in run-time data. Currently our approach modifies models through the estimation of their numerical parameters and we do not take into account structural changes. We plan to investigate this issue in the future and to support it in KAMI. In addition, we plan to investigate the reaction phase in the control loop, which is presently not supported. Our final goal is to support software engineers during all the development process to obtain evolvable and dependable systems in which models coexist with implementations to achieve run-time adaptability.

Table 1. BPEL Graphical Notation

Activity	Shape	Activity	Shape	Activity	Shape
receive		*wait*		*pick*	
invoke		*terminate*		*flow*	
reply		*sequence*		*fault handler*	
assign		*switch*		*event handler*	
throw		*while*		*compensation handler*	

10. Appendix

BPEL, *Business Process Execution Language*, is an XML-based workflow language conceived for the definition and the execution of service compositions. Table 1 shows the graphical notation adopted in this paper to represent BPEL constructs. BPEL processes comprise variables, with different visibility levels, and the workflow logic expressed as a composition of elementary activities. Activities comprise tasks like: *Receive*, *Invoke*, and *Reply* that are related to the interaction with other services. Moreover it is possible to perform assignments (*Assign*), throwing exceptions (*Throw*), pausing (*Wait*) or stopping the process (*Terminate*). Branch, loop, while, sequence and switch constraints manage the control flow of BPEL processes. The pick construct is peculiar to the domain of concurrent and distributed systems, and waits for the first out of several incoming messages, or timer alarms to occur, to execute the activities associated with such an event. Each scope may contain the definition of the several handlers: (1) an Event Handler that reacts to an event by executing a specific activity, (2) a Fault Handler catches faults in the local scope, and (3) a Compensation Handler aimed at restoring the effects of a previously unsuccessful transaction. For a complete description of BPEL language see [1, 10].

Acknowledgments

This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom. We thank the anonymous reviewers and Lars Grunske for their helpful comments on an earlier version.

References

[1] A. Alves, A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, Sterling, D. König, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web services business process execution language version 2.0. OASIS Committee Draft, May 2006.

[2] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[3] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *IET Software*, 1(6):219–232, December 2007.

[4] L. Baresi and S. Guinea. Towards Dynamic Monitoring of WS-BPEL Processes. *Proceedings of the 3rd International Conference on Service Oriented Computing*, 2005.

[5] S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the palladio component model. In *WOSP '07: Proceedings of the 6th International Workshop on Software and Performance*, pages 54–65, New York, NY, USA, 2007. ACM.

[6] J. O. Berger. *Statistical Decision Theory and Bayesian Analysis*. Springer, 2 edition, 1985.

[7] M. Bertoli, G. Casale, and G. Serazzi. The jmt simulator for performance evaluation of non-product-form queueing networks. In *Annual Simulation Symposium*, pages 3–10, Norfolk,VA, US, 2007. IEEE Computer Society.

[8] M. Bertoli, G. Casale, and G. Serazzi. An overview of the jmt queueing network simulator. Technical Report TR 2007.2, Politecnico di Milano - DEI, 2007.

[9] G. Bolch, S. Greiner, H. de Meer, and K. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. Wiley-Interscience New York, NY, USA, 1998.

[10] BPEL. http://www.oasis-open.org/.

[11] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik. Early prediction of software component reliability. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 111–120. ACM, 2008.

[12] P. Diaconis and D. Ylvisaker. Conjugate priors for exponential families. *Ann. Statist*, 7(2):269–281, 1979.

[13] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. Rosenblum, and S. Uchitel. Model checking service compositions under resource constraints. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 225–234, New York, NY, USA, 2007. ACM.

[14] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. *Automated Software Engineering*, 0:152, 2003.

[15] C. Fraser and D. Hanson. Mercury LoadRunner Monitor Reference. *Mercury Interactive*, 2004.

[16] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM.

[17] A. Ganek and T. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[18] J. K. Ghosh and R. V. Ramamoorthi. *Bayesian Nonparametrics*. Springer, 2003.

[19] S. Gokhale and K. Trivedi. Structure-Based Software Reliability Prediction. *Proc. of Fifth Intl. Conference on Advanced Computing (ADCOMP97)*, pages 447–452, 1997.

[20] S. S. Gokhale. Architecture-based software reliability analysis: Overview and limitations. *IEEE Trans. Dependable Sec. Comput.*, 4(1):32–40, 2007.

[21] D. Hamlet, D. Mason, and D. Woit. Theory of software reliability based on components. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 361–370, Washington, DC, USA, 2001. IEEE Computer Society.

[22] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS06)*, 3920:441–444, 2006.

[23] JMeter. http://jakarta.apache.org/jmeter/.

[24] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 2.0: a tool for probabilistic model checking. *Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. First International Conference on the*, pages 322–323, 2004.

[25] D. MacKay and L. Peto. A hierarchical Dirichlet language model. *Natural Language Engineering*, 1(3):1–19, 1995.

[26] S. Meyn and R. Tweedie. *Markov chains and stochastic stability*. Springer-Verlag London, 1993.

[27] S. Nakajima. Model-checking verification for reliable web service. *OOPSLA 2002 Workshop on Object-Oriented Web Services, Seattle, Washington*, 2002.

[28] M. Papazoglou and D. Georgakopoulos. Service-Oriented Computing. *Communications of the ACM*, 46(10):25–28, 2003.

[29] F. Raimondi, J. Skene, L. Chen, and W. Emmerich. Efficient Monitoring of Web Service SLAs. *UCL, Dept. of Computer Science. Research Note RN/07/01. Gower St, London WC1E 6BT, UK*, 2007, to appear at FSE 2008.

[30] N. Sato and K. S. Trivedi. Stochastic modeling of composite web services for closed-form analysis of their performance and reliability bottlenecks. In *ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing*, pages 107–118, Berlin, Heidelberg, 2007. Springer-Verlag.

[31] C. U. Smith and L. G. Williams. *Performance solutions: a practical guide to creating responsive, scalable software*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.

[32] C. C. Strelioff, J. P. Crutchfield, and A. W. Hübler. Inferring markov chains: Bayesian estimation, model comparison, entropy rate, and out-of-class modeling. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 76(1), 2007.

[33] T. Zheng, M. Woodside, and M. Litoiu. Performance model estimation and tracking using optimal filters. *IEEE Transactions on Software Engineering*, 34(3):391–406, 2008.