Reasoning on Non-Functional Requirements for Integrated Services

Carlo Ghezzi and Giordano Tamburrelli
DeepSE Group
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza L. da Vinci, 32 20133 Milano, Italy
(ghezzi|tamburrelli)@elet.polimi.it

Abstract

We focus on non-functional requirements for applications offered by service integrators; i.e., software that delivers service by composing services, independently developed, managed, and evolved by other service providers. In particular, we focus on requirements expressed in a probabilistic manner, such as reliability or performance. We illustrate a unified approach—a method and its support tools—which facilitates reasoning about requirements satisfaction as the system evolves dynamically. The approach relies on run-time monitoring and uses the data collected by the probes to detect if the behavior of the open environment in which the application is situated, such as usage profile or the external services currently bound to the application, deviates from the initially stated assumptions and whether this can lead to a failure of the application. This is achieved by keeping a model of the application alive at run time, automatically updating its parameters to reflect changes in the external world, and using the model's predictive capabilities to anticipate future failures, thus enabling suitable recovery plans.

1. Introduction

Modern software applications increasingly take the form of services. Services are software components that are developed, managed, deployed, and operated by service providers, and are offered for use through the network [27, 30]. Their specification is published in registries, which may be searched by potential clients. Services, in turn, may be used by service integrators and composed to provide new added-value services. In this paper, we take the viewpoint of service integrators and illustrate an approach which may enable them to fulfill their requirements even in presence of certain failures.

Specifications play a crucial role in software development, especially when multiple parties are involved. According to [25], a specification defines a *contract* between the provider and the user of a piece of software. Specifications play an even more crucial role in the case of services, because they define Service Level Agreements (SLAs) that can have a legal value in the relationship between service providers and their clients. On the client's side, the specification should characterize the *required service interface*; that is, it should state all the properties (or *quality attributes*) the client needs to know to reliably select and use the service. On the provider's side, it should characterize the *provided service interface*; that is, it should state precisely what the provider promises to offer. If the user is a service integrator, the contracts with service providers are, in turn, the basis for a contract between the service integrator and its users. A possible violation of the former contracts may indirectly cause a violation of the latter.

The world of services is inherently *open* [5]. By this we mean that the environment in which the application is embedded evolves continuously. New services are created and exposed for use; existing services are discontinued. Services may also undergo unanticipated changes. Providers may modify the implementation of an exported service, thereby affecting one of its quality attributes; for instance, its promised response time. Although in principle we should expect that the published service specifications reflect the changes in the implementation, there is no assurance that the two are kept coherent over time. Usage profiles also may change continuously in an open world, and these changes may affect quality attributes. For example, an unforeseen sudden heavy load on one of the externally used services integrated by a composite service may cause an unanticipated performance failure of the external service, which in turn causes a performance failure of the composite service.

We argue that service integration in an open-world setting challenges our current ability to develop dependable software. A service integrator must guarantee a certain contractual quality of service (QoS) in presence of changes in the external environment, such as the services bound to the application or user profiles, which may produce unpredictable effects and may occur at unpredictable times.

In this paper, we assume that suitable languages are available to specify services. In particular, we focus on *non-functional requirements*[1] that deal with *dependability*[2], specified in a probabilistic manner. For example, *reliability* of a given service may be expressed as probability of failure, i.e., probability that the result of a service invocation braks the contract stated by the service's specification. More complex dependability requirements may be expressed as probabilistic properties in some temporal logic which predicates over sequences of interaction events with the service, as we will see later.

Traditionally, the verification that a given software system satisfies its requirements is performed at development time. The goal of verification is to ensure that software satisfies its requirements before its delivery. If the need for changes arises, a new version is developed, verified, and eventually delivered, deployed, and put in operation. Only in a few special cases, changes are made and injected in the application as it is running.

The open-world requirements that characterize services invalidate the above scenario. For instance, the external services composed by a service integrator may change as the application is running. Because changes may invalidate their original specification, the overall requirements of the composed service might in turn be violated. It is thus necessary to continue to verify the application at run time: development-time verification may in fact be undermined by unexpected changes in the external environment, which subvert the assumptions made at design time.

The purpose of our work is to provide a framework—a method and supporting tools—to structure and reason about requirements of open-world systems built as integrated services which rely on existing, externally provided services. Our focus is on non-functional requirements expressed in a probabilistic manner and in particular in this paper we concentrate on reliability analysis. The main goal of analysis is to support mechanical ways of detecting reliability requirements violations. As we will see in Section 2, it is important to distinguish between two kinds of violation:

1. *requirements failure detection*, meaning that the violation of the requirement is visible to the client of the integrated service as a violation of the contract that was established with the service integrator;
2. *requirements failure prediction*, meaning that a violation of the contract between the user and the service integrator is predicted to occur in the future, but is not experienced yet.

[1]M. Glinz [17] cogently argues that the term "non-functional requirement" does not convey a precise notion, and perhaps should be avoided. Here we continue to use it informally as a shortcut.

[2]The notion of dependability, as defined by the IEEE Dependable Computing and Fault-Tolerance and IFIP Working Group, is the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers.

To support the aforementioned mechanical reasoning on requirements failures and faults, we provide a run-time toolset that is based on three components: A *monitor* collects data on the behavior of the external environment; a *model* of the application runs to verify requirements; a *model updater* updates the model's parameters to reflect changes in the environment. By keeping the model alive at run time, we will show that we can perform mechanical reasoning to detect requirements failures and requirements faults.

This paper builds on previous work reported in [12]. In [12] we focused on the theoretical underpinnings of automatic model update by parameter adaptation. The parameters of the model of a composite service used for verification at development time are updated at run time to reflect changes in the behavior of external services. The model updater is based on a Bayesian estimator [7], which is fed with data collected from the running system, and produces updated parameters. In [12], we show that the updated model provides an increasingly better representation of the system, as more data are collected from the field at run-time.

The new contribution of this paper consists of exploiting the results of [12] in the context of requirements engineering for open systems. We define a reference framework for requirements of open systems which extends the well-known framework of Jackson and Zave [21, 36] and generalizes to requirements the notions of fault and failure that are often used in the context of programming.

The paper is organized as follows. Section 2 provides an extended description of the proposed approach. Section 3 describes KAMI, our framework supporting verification of non-functional requirements. Section 4 describes the application of our approach to a realistic example. Section 5 shows the usefulness of our distinction between requirements fault and failure on supporting prediction of contract violations in the context of our running example. Section 6 discusses related work. Section 7 concludes the paper with a discussion of current limitations and an outlook to future work.

2. A Reference Framework for Requirements of Service Integrations

M. Jackson and P. Zave, in their seminal work on requirements [21, 36] , distinguish between two main concerns: the *world* and the *machine*. The machine is the system to be developed; the world (the environment) is the portion of the real-world affected by the machine. The ultimate purpose of the machine is always to be found in the world. *Requirements* thus refer to phenomena occurring in the world, as opposed to phenomena occurring inside the machine. Some of such phenomena are shared with the machine: they are either controlled by the world and observed by the machine, or controlled by the machine and observed by the world. A *specification* (for the machine) is a prescriptive statement of the relation on shared phenomena that must be enforced by the system to be developed. Finally, it is also important to understand the relevant *domain knowledge*, i.e., the set of relevant assumptions that can be made about the environment in which the machine is expected to work, which affect the achievement of the desired results. Quoting from [36],

> *"The primary role of domain knowledge is to bridge the gap between requirements and specifications."*

If R and S are the prescriptive statements that formalize the requirements and the specification, respectively, and D are the descriptive statements that formalize the domain assumptions, it is necessary to prove that

$$S, D \models R \tag{1}$$

i.e., S ensures satisfaction of the requirements R in the context of the domain properties D.

We focus on systems whose expected behavior S is achieved by integrating externally available services provided by third parties, invoked as black boxes. In the well-known and practically relevant case of Web services, external services are composed through a service composition (or workflow) language, such as the BPEL [1, 9] de-facto standard. The situation is illustrated in Figure 1(a). Figure 1(b) illustrates a high-level view of a workflow which integrates external services.

Hereafter we further detail the Jackson/Zave framework to match the specific issue that characterize an open-world setting, and more specifically service integration. Within domain properties D, we can identify two main disjoint subsets, D_u and D_s, which capture the assumptions made on the external environment upon which the machine is expected to achieve the stated requirements, and which are also likely to change over time. D_u denotes the *assumptions on usage profiles*. It consists of properties that characterize how the final integrated system is expected to be used by its clients. D_u properties must be taken into account during design, but they may turn out to differ from what the running system eventually exhibits, and may change dynamically. A possible example of a usage profile is:

> The volume of purchases during the two weeks before Christmas is 10 times the average volume of all other weeks of the year.

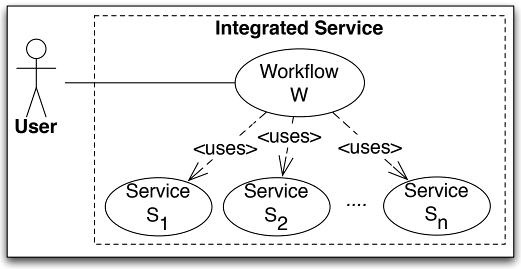
D_s instead denotes the *assumptions on the external services* invoked by the application. If $S_1, S_2, \ldots, S_n$ are the specifications of these external services, $D_s = S_1 \cup S_2, \ldots \cup S_n$.

Elicitation of D_u and D_s is a crucial phase of requirements engineering for open-world systems. Environment changes are a primary source of the need for continuous evolution in open-world systems (other sources concern requirement changes). By capturing D_u and D_s we identify the environment's data that must be monitored at run time and the checks to be performed to verify that the environment behaves as expected. If it doesn't, (1) might be violated.

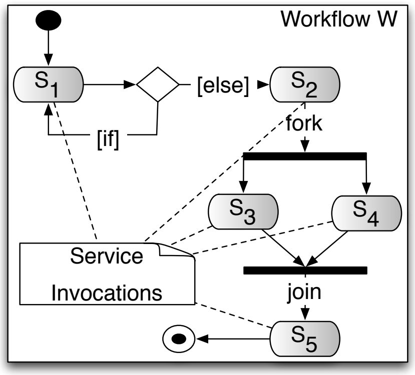
We do not make any specific assumptions on the languages in which one can express the statements R, D, S. No commonly used notations, let alone standards, exist for them. For example, in the case of Web services, the WSDL [34] specification language only focuses on syntactic aspects of a service interface, and hence it does not fit our purpose. Attempts to support specification of quality attributes, such as [32], are still not adopted in practice.

To support reasoning about requirements, we rather take a more pragmatic approach that does not require the definition of a new notation for R, S, and D, but instead chooses notations that can be supported by existing analysis tools. Hereafter we discuss the choices we made in our framework to express reliability-related statements, which can then be processed by the tool we chose for formal analysis, PRISM [20, 23]. Accordingly, we assume that a model of the system-to-be is provided in terms of a *state-transition system*. The state-transition system is exactly an operationalized description of S. The operational model is then augmented with probabilities, yielding a stochastic process represented as a *Markovian model*. We use such probabilities to express both D_s and D_u. The former describes the probability of failure of external services, whereas the latter describes the probability distribution of the user requests that are handled by the system-to-be, as we will illustrate on a running example in Section 4. To describe the required properties of the integrated service (R), two probabilistic temporal logic languages are supported by PRISM: Probabilistic Computation Tree Logic (PCTL) [19] and Continuous Stochastic Logic (CSL) [3]. At this stage our framework supports *Discrete Time Markov Chains* and properties expressed in PCTL. Support for other Markovian models is the subject of our ongoing research.

As we mentioned, our main goal is to support dependable service provisioning in dynamically evolving environments. This requires that at design time we can prove the validity of (1). However, because of possible deviations of the environment's behavior with respect to design-time as-



(a) Integrated services scenario.



(b) Workflow Example

Figure 1. Integrated Services

sumptions, verification of (1) must be extended to run time. In other terms, the model is kept alive at run time and fed with updated values of the parameters that characterize the behavior of the environment, i.e., D_s and D_u, as we show in the next section. At run time, analysis of the model can reveal requirements faults and requirements failures. The terms *fault* and *failure* are traditionally used in the context of program testing and debugging. It is crucial to distinguish between them also in the case of requirements. Although there is no consensus on terminology the two terms are often used in testing and debugging to denote different concepts. In [16], a failure is what an oracle detects as incorrect output values, while a fault is the incorrect internal state that a debugger can identify and display, to support identification of the faulty statements (*error*) that may be ultimately responsible for the failure. The same semantic distinction is made in [37], although the term *infection* is adopted instead of fault.

These concepts may be generalized to the context of requirements engineering. A requirement failure is a violation of a requirement which has a visible effect in the real world; typically, it is perceived by the users of an application. A requirement fault instead occurs in the machine or in the environment and may not be visible in the real world. Assuming that the machine is correctly implemented, a fault represents a deviation of the environment's properties D_s and D_u from their expected values. It denotes a situation that *may* eventually lead to a failure. In the case of service integration of Figure 1(b), assuming that the workflow program is correct, a fault may be caused by a failure of an external service $S_1, S_2, \ldots, S_n$; i.e., one of the external services invoked by the workflow does not satisfy its specification. It can also be caused by an unexpected usage profile, which does not follow the assumption made at design time.

The distinction between fault and failure is key to dependability. For example, consider the case where an external service deviates from its contractual obligations, stated by specification S_i. Let us assume that the deviation is detected and that the parameters of the model are updated accordingly, as we mentioned above (and as we will se later on). By running the updated model, the possible cases can arise:

1. The updated model shows that requirements R continue to be satisfied. In this case the fault (violation of the contract by external service) has no effect on satisfaction of the requirements of the integrated service. Thus, no action needs to be taken to ensure correctness although the external service provide might incur in a penalty because of a break in the contract.

2. The updated model shows that requirements R are violated. This violation us also visible to the external user, who thus perceives a violation of the contract with the service provider. This is the case we called *requirements failure detection*. In this case, the fault (violation of contract by the external service) results in a failure caught by the model and visible to the user.

3. The updated model shows that requirements R are violated, but this violation did not manifest itself in the real world. The user does not perceives any violation of the contract with the service provider, but the model shows that a failure will eventually occur. This is the case we called *requirements failure prediction*

To support automated reasoning about requirements faults and failures and to support prediction, we exploit the functionalities offered by KAMI framework, which is illustrated in the next section. Subsequently, we show an application of our approach to an example, which further investigates the notions of failure detection and failure prediction.

3. Supporting Run-Time Model Evolution with KAMI

KAMI is the design methodology and prototype environment we are developing to support automated evolution of the system's model as the external service world changes.

KAMI supports models for dependability properties, such as performance and reliability. Presently, we support Discrete Time Markov Chains (DTMCs) [18, 26] and we plan to incorporate other Markovian models in the future. We are also currently working on incorporating Queueing Networks (QNs) [8, 24], which support analysis of performance properties.

Figure 2 shows a high-level view of the KAMI plug-in based architecture. KAMI is composed by: (1) System Models, (2) Model Plugins, and (3) Input Plugins.

System Models are text files that contain model descriptions with numerical parameters that KAMI is in charge of updating and the requirements the user is interested in. System models also contain a set of exceptions to be raised when a requirement is violated.

Model Plugins provide to KAMI the ability to handle different and new models, by interpreting model files and their requirements. Each model plugin is in charge of analyzing a specific kind of model with respect to its requirements. Models that violate a given requirement cause the corresponding exception to be raised by KAMI.

Input Plugins connect KAMI with the run-time world in which the implemented system is running. The running system feeds KAMI with data extracted by its execution. For example, in the case of DTMCs, it provides information about the occurrence of transitions among states. Input plugins are in charge of managing different input formats and protocols for run-time data (e.g, socket, RMI, etc.).

All the classes of models supported by KAMI are characterized by and depend on numerical parameters. For example, consider a DTMC used to model the behavior of a service composition to reason about reliability properties. The external services used by the service integrator must be characterized by their probability of failure, which is part of the service's specification. Let us assume that, by using these parameters, we can prove correctness of the system; i.e., Formula (1) is verified.

Because of the open-world assumption, we may expect a failure of the external services at run time. KAMI is able to detect such failure because it monitors service invocations and estimates the real values of the parameters based on the observations. KAMI does this through a Bayesian estimator [7], discussed and evaluated in [12]. A change in the parameter for the i^{th} external service may indicate a violation of specification S_i, and hence S_i's failure. The modified DTMC which models the service composition with updated parameters may be run to check if any requirement is violated, according to Formula (1). This may lead to recovering a failure.

KAMI must be connected to probes which channel real-world data generated by monitors. In our experiments with Web services we can use the monitoring framework of Baresi et al [6].

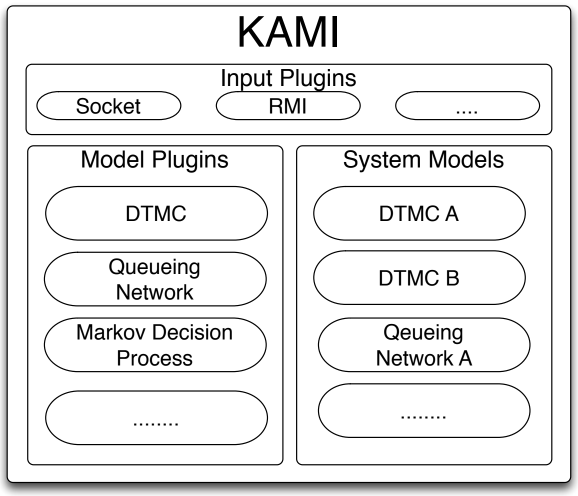


Figure 2. KAMI Architecture

4. An Example

This section illustrates a concrete example in which our approach is successfully applied. The example represents a typical e-commerce application that sells on-line goods to its users by integrating the following third-parties services: (1) *Authentication Service*, (2) *Payment Service*, and (3) *Shipping Service*. The Authentication Service manages the identity of users (e.g., [29]). It provides a *Login* and *Logout* operation through which the system authenticates users. The Payment Service provides a safe transactional payment service through which users can pay goods. A *CheckOut* operation is provided to perform such payments. The Shipping Service is in charge of shipping goods to the customer's address. It provides two different operations: *NrmShipping* and *ExpShipping*. The former is a standard shipping functionality while the latter represents a faster and more expensive alternative. Finally, the system classifies the logged users as *BigSpender* (BS) or *SmallSpender* (SS), based on their usage profile. Indeed, if the money spent by a given user in his/her previous orders reaches a specific threshold he/she is considered as BigSpender, otherwise users are considered as SmallSpenders.

At design-time the system is built in order to meet the desired requirements. Since we focus on reliability, let us suppose we elicited the following requirements:

- *R1: "Probability of success is greater then 0.8"*
- *R2: "Probability of a ExpShipping failure for a user recognized as BigSpender is less then 0.035"*
- *R3: "Probability of an authentication failure is less then 0.06"*

Table 1. Domain Knowledge D_u

$D_{u,n}$	Description	Value
$D_{u,1}$	*P(User is a BS)*	0.35
$D_{u,2}$	*P(BS chooses express shipping)*	0.5
$D_{u,3}$	*P(SS chooses express shipping)*	0.25
$D_{u,4}$	*P(BS searches again after a buy operation)*	0.2
$D_{u,5}$	*P(SS searches again after a buy operation)*	0.15

Table 2. Domain Knowledge D_s

$D_{s,n}$	Description	Value
$D_{s,1}$	*P(Login)*	0.03
$D_{s,2}$	*P(Logout)*	0.03
$D_{s,3}$	*P(NrmShipping)*	0.05
$D_{s,4}$	*P(ExpShipping)*	0.05
$D_{s,5}$	*P(CheckOut)*	0.1

Concerning the domain knowledge D_u, we summarize what domain experts or previous similar systems tell us in Table 1. The notation $P(x)$ denotes the probability of "x". Table 2 describes the properties that characterize the domain knowledge about external services (i.e., D_s). $P(Op)$ here denotes the probability of failure of service operation Op.

As we recalled, the software engineer's goal is to derive a specification S which leads to satisfaction of requirement R, assuming that the environment behaves as described by D. We assume that software engineers provide an operational description of S through an activity diagram (see Figure 3), which describes a service composition via a workflow. Figure 3 is an abstract description of the integrated service, given by using a technology-neutral notation (as opposed, for example, to a specific workflow language like BPEL [1, 9]).

In order to verify Formula (1), we derive from the activity diagram a richer target model that merges S and D into a unique model, which can then be analyzed to check the properties of interest.

Since our goal is to reason about reliability requirements, the target model we chose is a Discrete Markov Chain (DTMC). The resulting model is illustrated in Figure 4. It contains one state for every operation performed by the system plus a set of auxiliary states representing potential failures associated with auxiliary operations (e.g., state 5) or specific system's sates (e.g., state 2). Existing tools are available to support automatic derivation of model-to-model trasformations, including trasformations from activity diagrams to markovian models (e.g., [15]).

Through the model in Figure 4, it is possible to verify Formula (1). To accomplish this task we adopt automatic

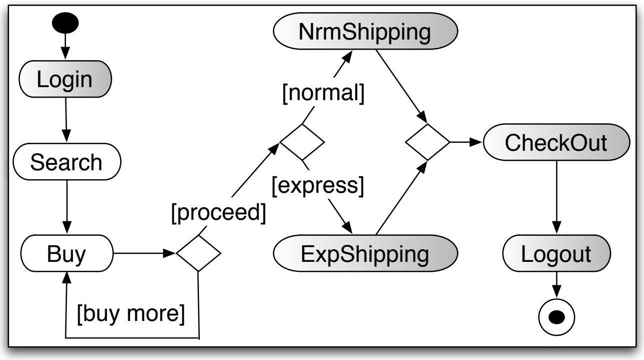


Figure 3. Operational Description of the Specification via an Activity Diagram

model checking techniques. In particular, we use the probabilistic model checker PRISM [20, 23], which in the example computes the following values:

- *"Probability of success is equal to 0.804"*
- *"Probability of a ExpShipping failure for a user recognized as BigSpender is equal to 0.031"*
- *"Probability of an authentication failure (i.e., Login or Logout failures) is equal to 0.056"*

Let us now focus on requirement $R2$. The compliance of the system with respect to $R2$ is checked at design time by computing the probability of the set of paths:

$$\Pi = (1, (4, 7)^{+}, 10, 13) \tag{2}$$

These paths include the transition from state 7 to state 10, which represents the probability that a BS user chooses an express shipping (i.e., $D_{u,2}$), and the transition from state 10 to state 13, which represents the probability that an invocation to the ExpShipping service fails (i.e., $D_{s,4}$). These two parts influence the total probability that a BS user who invokes an express shipping experiences a failure (i.e., $R2$).

Since the model checker computed at design time a value for paths Π that is smaller then the value stated by $R2$, we can conclude that the implementation of Figure 3 is correct with respect to $R2$. Similar considerations hold for $R1$ and $R3$. These conclusions rely on the assumption that assertions D hold at run time. However, as we noticed in Section 1, there is no assurance that design-time assumptions are valid at run time and remain immutable over time. More precisely, considering again $R2$ we could experience at run time different values for $D_{u,2}$ or $D_{s,4}$ and consequently we could experience at run time a probability of paths Π that violate $R2$.

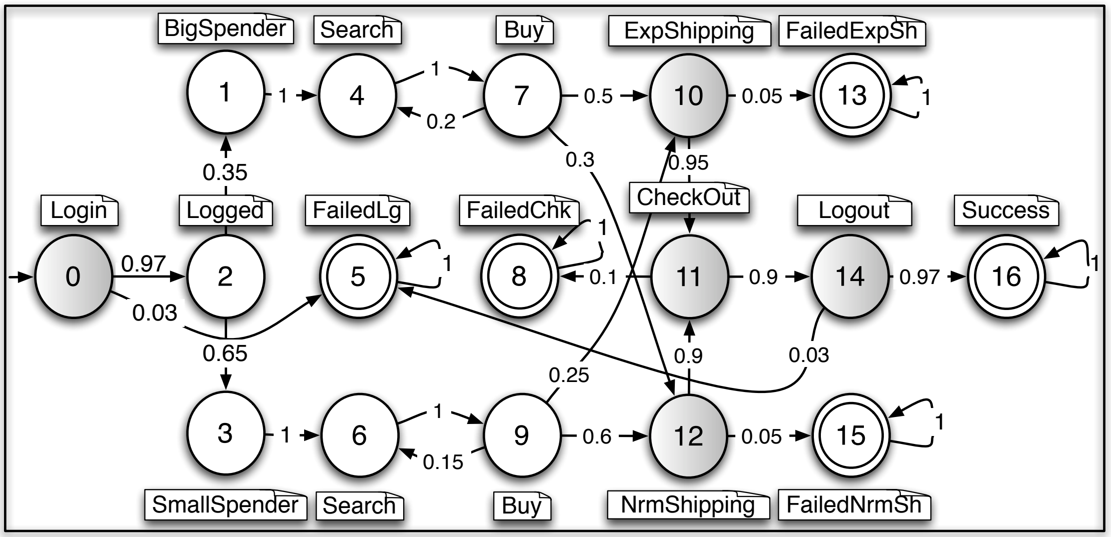


Figure 4. Example DTMC Model

For example, let us assume that the reliability of the Payment Service is lower then the value published by the service provider. More precisely, let us suppose that the probability of failure for each ExpShipping invocation is indeed slightly higher and equal to 0.067 ($D_{s,4} = 0.067$). In this case probability of Π is equal to 0.042 which is greater then 0.035 and thus violates the requirement. In this scenario KAMI collects run-time data from the running instances of the system as described in [12] and produces estimates for the transition probabilities of the DTMC, using a Bayesian approach. In particular, assuming a trace of events detected by the monitoring containing 600 invocations to the ExpShipping operation with 40 failing invocations (40 over 600 is approximately 0.067) the Bayesian estimator produces a new estimate for $D_{s,4}$ equal to 0.06. KAMI replaces the old estimate in the DTMC with this new value and automatically re-checks the requirements discovering that $R2$ is violated, since the probability of paths Π is estimated as equal to 0.037, which is greater then 0.035.

Similarly, a domain property $D_{u,i}$ could differ at run time from what we expected at design time, leading to a requirement violation. At design time these values are typically: (1) provided by domain experts, (2) extracted by previous version of the system under design, or (3) obtained by profiling users of similar existing applications. However, this initial estimate could be imprecise or even change over time for unpredictable reasons. For example, let us assume now that the probability of failure for the ExpShipping operation is equal to what we assumed at design time, whereas the value of the probability that a BS user chooses express shipping changes to 0.633 (i.e., $D_{u,2} = 0.633$). In this case, the the probability of paths Π is equal to 0.04, which is greater then 0.035; thus $R2$ is violated. In particular, assuming again a trace of events detected by the monitor that shows that over 600 users recognized as BigSpenders 380 of them chose the express shipping (380 over 600 is approximately 0.633) we obtain from KAMI an estimate for $D_{u,2}$ equal to 0.6 and the probability of Π is consequently estimated at run time to be equal to 0.037, which is a violation of $R2$.

In both cases (a deviation from the value of $D_{u,2}$ or $D_{s,4}$) KAMI detects the violation and can raise an exception to trigger an appropriate reaction. For example the reaction could signal to the system administrator that the contract between the system and the Authentication Service has been violated. Conversely, a more complex reaction could trigger an automatic reconfiguration that binds the system to another more reliable and semantically equivalent service provider.

Let us now focus on the first part of the above example in which $R2$ is violated because of an unpredicted change of value of the domain property $D_{s,4}$. An important consideration is that KAMI detects the violation in two different scenarios: (1) *Failure Detection* and (2) *Failure Prediction* that we briefly describe in the sequel on the example and we generalize and formalize in the next section.

First of all, it is important to notice that stakeholders, on behalf of whom we must guarantee probabilistic requirements R, perceive a failure by comparing the number of successful invocations to the system with respect to a (sufficiently large) number of total invocations[3]. In the specific

[3]The number of observations needed to evaluate probabilistic require-

case of $R2$, stakeholders just count the number of BSs that request an express shipping (BS_{exp}) and the ones which experience a failure ($BS_{exp,f}$) in a specific interval of time. Consequently, $R2$ is considered as violated if:

$$\frac{BS_{exp,f}}{BS_{exp}} \geq 0.035 \qquad (3)$$

Considering again the trace of 600 events with 40 failed invocations to ExpShipping (40 is a reasonable value since $D_{s,4} = 0.067$) and letting x be the number of BS users that experienced the failed invocations in the trace, we could have the following cases:

- $2 \leq x < 40$: Several BS users invoked the ExpShipping service experiencing a failure, R2 is detected as violated by KAMI and also by stakeholders
- $x = 1$: One BS user invoked the ExpShipping service experiencing a failure, R2 is detected as violated only by KAMI (i.e., failure prediction)[4]
- $x = 0$: No BS users invoked the ExpShipping service experiencing a failure, R2 is detected as violated only by KAMI (i.e., failure prediction) even if no BS users have been yet invoked the operation for which the requirement in stated

The next section further investigates the concepts introduced through this example with a particular emphasis on failure predictions.

5. Requirements Failure Prediction and Detection

In this section we investigate the reasons why a model can be able to predict a failure, by generalizing the specific case we observed in the example of the previous section. Let us suppose that any reliability requirement ρ in R can be described by the following simple template:

> *"The probability P that a certain input to the system eventually generates a certain output is less than a certain threshold value t."*

Requirement ρ typically states a probability of *failure* of a certain expected behavior. Should the requirement instead be expressed as a *success* condition, it could be written as $1 - \rho \geq t$.

ments depends on the variance associated to run-time data and on the level of confidence we want to have with respect to the evaluation. A complete investigation of this issue is part of our future work. The example we illustrate here is chosen because of its simplicity; it is not required to be realistic.

[4]It is important to notice that this scenario is not unlikely since the probability that a failed invocation to the ExpShipping service belongs to a BS user is even less then $\frac{1}{40}$ and equal to 0.0147, as computed by PRISM.

From the viewpoint of the stakeholder who generated requirement ρ, its violation can be verified by counting a sufficiently long sequence of input events of interest and comparing the number of successful output events E_f out of the total number of events of interest E:

$$\frac{E_f}{E} < t \qquad (4)$$

From the model's viewpoint, ρ corresponds to a set of paths $\Pi = (\pi_1, \pi_2, \ldots, \pi_n)$ between two specific states of the DTMC. For example, requirement $R2$ described in Section 4 corresponds to the path described by Formula (2). The verification of the model can detect a violation of ρ if and only if the probability associated with Π is greater or equal to t. The probability associated with Π is computed as:

$$P_{\Pi} = \sum P_{\pi_i} \qquad (5)$$

Assuming that π_i is a path $x_{i,1}, \ldots, x_{i,k}$ where $x_{i,j}$ and $x_{i,j+1}$ represent two consecutive transitions in the DTMC model along path π_i, the probability associated to this path is:

$$P_{\pi_i} = \prod_{j=1}^{k} P_{x_{i,j}} \qquad (6)$$

where $P_{x_{i,j}}$ is the probability associated with transition $x_{i,j}$ in the DTMC model. Notice that probability associated with Π exactly corresponds to the summary of path probabilities (i.e., P_{π_i}) since paths are mutually exclusive. By this we mean that the evolution of a system cannot be represented by the execution of multiple paths at the same time.

Requirements failure prediction occurs if the sequence of events captured by the monitor generates an update of the model's parameters that lead to a verification of a violation of ρ, whereas the failed occurrences of outputs E_f observed by the stakeholder do not violate Formula (4). Conversely, the case in which both the model and the stakeholder detect a violation of ρ is called *requirements failure detection*.

To understand the conditions under which failure prediction can happen, consider that the paths corresponding to ρ are partly shared with paths which correspond with other executions of the system which may be associated with other–possibly less critical–requirements. These executions may have actually occurred and consequently they may have updated the values of probabilities associated with transitions that are shared by the two sets of paths. The updated values may produce as a side effect the detection of the violation of ρ. In the extreme case the violation of ρ can happen even if no failed output events that contribute to E_f have been experienced yet by the stakeholder, as we showed in the example of Section 4.

6. Related Work

Service-centric systems have been attracting considerable research attention in the recent years [28, 30]. Such systems are pushing traditional software engineering problems—requirements, specification, verification, distribution, componentization, composition, and evolution—to their extreme. Software services are in fact distributed, their development and operation is decentralized, ownership involves different organizations. By composing services, new added value services may be generated. The key question is how this can be done dependably for the users of service-oriented applications. A number of current research directions are exploring areas close to the work reported here. For example, an important issue deals with how quality of service can be specified and how it can be the basis for verifiable contracts (service-level agreements—SLAs) between service providers and service users. A language for SLA has been proposed by [32]. Other related areas deal with monitoring and verifying services and service compositions [4, 13, 14], Run-time verification is another closely related research area (for example, see Chen et al in [11]). Finally, there is work that focuses on how service-oriented systems affect requirements engineering. For example [35] presents an approach to increasing the completeness of system requirements using information about designs and implementations of Web services. Another important research thread that relates to our efforts is model-driven systems development [2]. The approach we are pursuing with KAMI is strongly related to all of the aforementioned efforts. However, we are not aware of any approach that, like ours, tries to combines model-driven development of service centered systems with requirements-aware lifelong verification.

Several research efforts are complementary to ours. In particular, the ones which focus on reacting to failures via self-managed changes in the architecture, which aim at generating self-healing behaviors. This thread of research falls within autonomic computing [22]. Some promising approaches have been investigated by [10] and [31] Other complementary approaches investigate alternative methods for calibrating model parameters at run time in the context of performance models [33].

7. Conclusions and Future Work

In this paper we presented a method and supporting tools to reason about requirements of open-world systems built as integrated services that rely on existing, externally provided services. We focused on non-functional requirements expressed in a probabilistic manner. We casted the seminal work of Jackson and Zave [21] to this particular domain, in which design time assumptions can be inaccurate and may change over time, affecting the satisfaction of requirements of the final system.

Our proposal relies on non-functional models, continuously updated in their parameters by the use of data extracted by running instances of the system. We described how, reasoning on updated models, it is possible to detect or predict failures. This, in turn, may trigger suitable self-managed reactions, which generate an autonomic behavior. The approach has been described trough a concrete example in which we showed how run-time data can lead to detection or prediction of failures.

A complete validation of the proposed methodology is part of our future work. In addition, our future research will enrich the ongoing implementation of KAMI 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 also plan to further investigate failure prediction leading to a complete mathematical framework aimed at explaining the conditions under which this phenomenon can take place.

Finally, in the future we will investigate the possible reactions that can be triggered by failure detections and predictions. 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.

Acknowledgments

This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

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] D. Ardagna, C. Ghezzi, and R. Mirandola. Rethinking the use of models in software architecture. In *Comparch 2008*, Lecture Notes in Computer Science. Springer, 2008.

[3] A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous time markov chains. In *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 269–276. Springer, 1996.

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

[5] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39(10):36–43, 2006.

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

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

[8] 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.

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

[10] G. Canfora, M. Penta, R. Esposito, and M. Villani. QoS-Aware Replanning of Composite Web Services. In *ICWS 2005 Proc*, 2005.

[11] F. Chen, T. Serbanuta, and G. Rosu. jPredictor: a predictive runtime analysis tool for java. In *Proceedings of the 13th international conference on Software engineering*, pages 221–230. ACM New York, NY, USA, 2008.

[12] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time adaptation. In *to appear at ICSE '09: The 31th Internationl Conference on Software Engineering. Available at: http://home.dei.polimi.it/tamburrelli/icse09.pdf*, Vancouver, CANADA, 2009.

[13] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proceedings of the 13th international conference on World Wide Web*, pages 621–630. ACM New York, NY, USA, 2004.

[14] X. Fu, T. Bultan, and J. Su. Synchronizability of Conversations among Web Services. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, pages 1042–1055, 2005.

[15] S. Gallotti, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Quality prediction of service compositions through probabilistic model checking. In *QoSA '08: Proceedings of the 4th International Conference on the Quality of Software Architectures*, Karlsruhe, Germany, 2008.

[16] C. Ghezzi, M. Jazayeri, and D. Mandrioli. Fundamentals of software engineering. 2003.

[17] M. Glinz. On Non-Functional Requirements. *Proc. Requirements Engineering Conference, 2007 (RE '07)*, pages 21–26, 2007.

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

[19] H. Hansson and B. Jonsson. Hansson, h., jonsson, b. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[20] 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.

[21] M. Jackson and P. Zave. Deriving specifications from requirements: an example. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 15–24, New York, NY, USA, 1995. ACM.

[22] J. Kephart and D. Chess. The Vision of Autonomic Computing. *COMPUTER*, pages 41–50, 2003.

[23] 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.

[24] E. Lazowska, J. Zahorjan, G. Graham, and K. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1984.

[25] B. Meyer. Applying design by contract. *Computer*, 25(10):40–51, 1992.

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

[27] E. D. Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engg.*, 15(3-4):313–341, 2008.

[28] E. D. Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engg.*, 15(3-4):313–341, 2008.

[29] OpenID. http://www.openid.com/.

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

[31] M. Rouached and C. Godart. Requirements-driven verification of wsbpel processes. *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 354–363, July 2007.

[32] J. Skene, D. D. Lamanna, and W. Emmerich. Precise service level agreements. In *In: Proc. of 26th Intl. Conference on Software Engineering (ICSE*, pages 179–188. IEEE Press, 2004.

[33] C. M. Woodside and M. Litoiu. Performance model estimation and tracking using optimal filters. *IEEE Trans. Softw. Eng.*, 34(3):391–406, 2008.

[34] WSDL. http://www.w3.org/2002/ws/desc/.

[35] K. Zachos and N. Maiden. Inventing Requirements from Software: An Empirical Investigation with Web Services. *International Requirements Engineering, 2008. RE'08. 16th IEEE*, pages 145–154, 2008.

[36] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, 1997.

[37] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2006.