Self-Adaptive Software Meets Control Theory: A Preliminary Approach Supporting Reliability Requirements

Antonio Filieri, Carlo Ghezzi, Alberto Leva, Martina Maggio
Dipartimento di Elettronica e Informazione, Politecnico di Milano
Piazza L. da Vinci, 32; 20133 Milano, Italy
{filieri, ghezzi, leva, maggio}@elet.polimi.it

***Abstract*—This paper investigates a novel approach to derive self-adaptive software by automatically modifying the model of the application using a control-theoretical approach. Self adaptation is achieved at the model level to assure that the model—which lives alongside the application at run-time— continues to satisfy its reliability requirements, despite changes in the environment that might lead to a violation. We assume that the model is given in terms of a Discrete Time Markov Chain (DTMC). DTMCs can express reliability concerns by modeling possible failures through transitions to failure states. Reliability requirements may be expressed as reachability properties that constrain the probability to reach certain states, denoted as failure states.**

We assume that DTMCs describe possible variant behaviors of the adaptive system through transitions exiting a given state that represent alternative choices, made according to certain probabilities. Viewed from a control-theory standpoint, these probabilities correspond to the input variables of a controlled system—i.e., in the control theory lexicon, "control variables". Adopting the same lexicon, such variables are continuously modified at run-time by a feedback controller so as to ensure continuous satisfaction of the requirements despite disturbances, i.e., changes in the environment. Changes at the model level may then be automatically transferred to changes in the running implementation.

The approach is methodologically described by providing a translation scheme from DTMCs to discrete-time dynamic systems, the formalism in which the controllers are derived. An initial empirical assessment is described for a case study. Conjectures for extensions to other models and other requirements concerns (e.g., performance) are discussed as future work.

***Keywords*-Adaptive software; control theory; dynamic systems; non-functional requirements; reliability; run-time verification.**

I. Introduction

Software systems are increasingly required to be self-adaptive. If certain requirements change, they must be able to adapt their behavior to keep satisfying them. Moreover, they should be able to detect changes in the environment in which they are embedded and automatically adapt their behavior to prevent violations of the expected quality attributes—functional and nonfunctional—they must fulfill.

In this paper we focus on systems that must guarantee certain *reliability requirements*, expressed quantitatively as constraints on the probability of reaching certain failure states. Furthermore, we focus on changes that occur in the environment in which the application is situated, which may lead to reliability vulnerabilities. The typical scenario we implicitly refer to is a *service-oriented application* (SOA) that composes a number of existing internal or external services through a workflow (*service orchestration*). The external services may have their own failure profiles, which affect the overall reliability of the composite application. External services are viewed as part of the environment, since they are not under control of the application and may evolve autonomously. For example, their failure profile may change dynamically, due to the upload of a new version of the application or changes in the delivery infrastructure.

User profiles are another possible environment phenomenon affecting the application. In our context, user profiles can be modelled as the probability that different options are selected by the user, whenever there are choices in the interaction with the composite service.

In this paper, we focus on self adaptation at the model level. That is, we assume that a model of the application is kept alive at run-time [1], [2]. We further assume that adaptation is performed at the model level, and then reified at the implementation level. We assume that the composite application is formally modeled by a *Discrete Time Markov Chain* (DTMC). DTMCs are known as a useful formalism to describe systems from the reliability viewpoint and to support reasoning about it. Since a DTMC is just a finite state machine with probabilities attached to transitions, it can easily model service invocations that may fail—by two transitions exiting a certain state, respectively representing success and failure—and user profiles—by transitions exiting a given state and representing different choices made by users. Reliability requirements may be typically expressed as *reachability properties*, i.e., as a relational formula constraining the probability of reaching certain states that represent failure situations.

To support self-adaptation, this paper explores a novel approach based on control theory (see Figure 1). We assume that our goal is to maintain a certain reliability profile for the application being designed. For simplicity, let us assume the profile to simply be a certain *target* value of the overall probability of failure. The *output* is a sequence of events that represent successes or failures. The sequence of events is transformed into the *actual* probability of failure by the *Learning Block*. This block transforms a sequence of failure

events into a probability (typically, using a Bayesian approach, as discussed in [3]). Blocks *System* and *Controller* in Figure 1 represent the modeled system and the controller, respectively. The goal of the controller is to provide *input* values to the system so that the resulting output (the observed sequence of failure and success events) does not violate the requirement expressed by the target, despite *disturbances*.

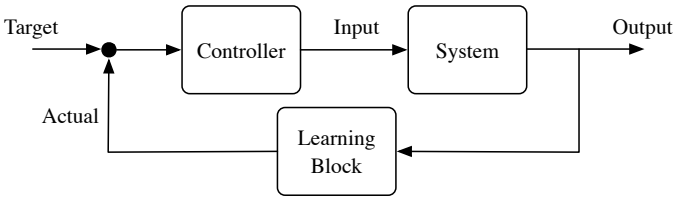

Fig. 1. Block diagram of the controlled system.

To understand what inputs and disturbances are in our context, we must first discuss how we deal with adaptation at the model level. We assume that the software model describes all possible variations that may be chosen to support adaptation. That is, the modeler anticipates a number of ways through which the system may self adapt its behavior. In a DTMC framework, choices can be expressed by using probabilities, which label transitions corresponding to the choice of different behaviors. By changing these probabilities it is possible to either increase or decrease the chance that a certain functionality is selected. In the extreme case, by setting a probability to 0 (or 1) a certain functionality is either excluded or included. These probabilities are inputs of our controlled system, generated by the controller. By changing them, the controller tries to ensure continuous satisfaction of the target reliability despite disturbances. Disturbances, in turn, are changes in the independent variables, also modeled by transition probabilities, that represent physical phenomena, like changes in the failure probability of external services or in the user profiles.

To the best our knowledge, the control-theoretical approach illustrated in this paper is a novel contribution to self-adaptive system models. In this paper, we illustrate the approach and provide an initial experimental assessment. The paper is organized as follows. Section II introduces the claims of this work and sketches the use of software models and abstractions for dynamic adaptation and control. In Section III a DTMC model for reliability is described and the case study used in the rest of this paper is presented. Section IV proposes a way to translate DTMC models into discrete time dynamic systems. The control of the resulting dynamic system is shown in Section V, that provides formal properties assessment and shows the application of the proposed technique to the chosen case study, evaluated in Section VI. Related works are described in Section VII while section VIII concludes the paper.

II. Control Theory and Software Modeling

The dynamics of software execution are very complex. Nonetheless, being able to control those dynamics would mean having a software capable to adapt and on-line tune itself to meet the specified requirements. However, the presence of intrinsic non linearities, the variety of usage profiles, the distribution process and the interconnection of heterogeneous components are some of the reasons why it is so hard to directly provide a comprehensive behavioral model suitable for control. At the same time, the need for continuous verification of specific properties lead to the definition of simpler models. These models are simple enough to allow the systematic synthesis of controllers capable of driving the modeled dynamics and still able to capture a number of aspects of the running software that significantly characterize the software behavior and support assessment of some of its properties.

In this paper we refer to a controller as any system that, properly coupled to the software system, makes it fulfill its requirements whenever they are feasible. Requirements can be strict constraints on the behavior (e.g. reliability equal to a certain value) or related to the optimization of certain metrics on the observed software executions (e.g. minimization of outsourcing costs or maximization of throughput).

This work is aimed at supporting the claim that control theory provides a number of instruments that software engineers can exploit to ensure the achievement of extra-functional design goals in presence of changes in the environment. To do so, we focus on the following main kinds of "reaction" the controlled system should be able to provide:

1) change of the target requirements. If for some reason the required nominal value of the overall reliability of the composed system changes, the controller should be able to drive the system toward a new operative state satisfying the requirements.
2) robustness to sudden changes or fluctuations around the nominal operative point assumed at design-time for the environment phenomena. Interdependence among software parts and components involves the use of third-party services, remote storage, computing resources out of the control of each company, and so on. All these parts are characterized by the values of certain QoS metrics, usually stated in convenient service level agreements. During normal execution those values may deviate from nominal values because of external factors hardly predictable a priori (e.g. load conditions or hardware failures). Actual values can be estimated on line via monitoring.
3) robustness to accuracy errors in measurement and monitoring. To capture relevant metrics of the execution we rely on monitoring and/or other measurement procedures. Each of these might get stuck into temporary bias or might require a certain time to produce an appropriate accuracy. We look for a controller able to provide a reasonable behavior even in presence of transitory errors on measured values. Such an ability, besides reducing

the sensitivity to measurement errors, allows for the use of less invasive monitoring instruments sometimes required for high accuracy, but expensive as performance overhead.

In the following we provide a formal assessment of these properties, based on mathematical modeling and we show the practical implications with a case study.

III. Control-Oriented Reliability Modeling

In this paper we focus on reliability, to ground the exposition with a practical example of the application of the proposed methodology. The approach can be extended to similar analytical models for performance and costs [4] (see Section V).

Reliability is an intrinsically probabilistic property, since it depends on the usage of the system and on external resources, whose characteristics are often uncertain [5]. In order to capture the behavior of the system a widely adopted model is a Discrete Time Markov Chain (DTMC) [6]–[9]. A DTMC can be used to represent all the relevant states of a software execution and the probability to move from each state to another. Among all the states, one state is the *initial state*[1]. Among the other states, one or more represent the successful completion of the execution or the occurrence of a failure. In this paper we assume there is a single *success state* s_R . Such an assumption does not reduce the expressiveness of the model[2]. Failure and success states are modeled as *absorbing states*, i.e. states with a self-loop transition labeled with probability 1. The rationale here is is that we view success and failure as definitive conditions: as the system reaches either condition, it cannot progress any more. All states that are not absorbing are called *transient states*.

Formally, a (finite) DTMC is a tuple $(S, s_0, \mathbf{P}, L)$ where:

- S is a finite set of states
- s_0 is the *initial state*
- $\mathbf{P} : S \times S \to [0,1]$ is a stochastic matrix (i.e. $\forall s_i \in S$ $\sum_{s_j \in S} \mathbf{P}(s_i, s_j) = 1$)
- $L : S \to 2^{AP}$ is a labeling function that marks every state s_i with the Atomic Propositions (AP) that are true in s_i.

We will interchangeably use both the notation $P(s_i, s_j)$ and p_{ij} to refer to the entry (i, j) of the matrix $\mathbf{P}$, corresponding to the probability of moving from state s_i to state s_j. A path through a DTMC is a (possibly infinite) sequence of states $\pi = s_0 s_1 s_2 ...$, where s_{i+1} is reachable from s_i through a transition. The probability $\mathbf{P}$ induces a probability space on the set of all possible paths [10]. The probability of a path π with length n can be defined as:

$$\begin{cases} Pr(\pi) = 0 \\ \quad \text{if } n = 0 \text{ i.e. } \pi = s_0 \\ \\ Pr(\pi) = \mathbf{P}(s_0, s_1) \cdot \mathbf{P}(s_1, s_2) \cdot ... \cdot \mathbf{P}(s_{n-2}, s_{n-1}) \\ \quad \text{if } n > 0 \end{cases}$$

For the purposes of this analysis, an execution of a software is completely described by its *trace*, that is, the corresponding path through the DTMC.

Reliability is the probability of successfully accomplishing an assigned task. In our setting it can be defined as the probability of reaching the state representing successful completion (s_R) given that the execution started from its initial state (s_0). Thus reliability is defined as $\sum_{\pi \in \Pi^*} Pr(\pi)$, where Π^* is the set of all possible paths starting from s_0 and ending in s_R. Given that s_R is an absorbing state, reliability can be paraphrased as the probability of reaching s_R, since, once reached, s_R cannot be left. Properties of a DTMC can be expressed using the PCTL probabilistic temporal logic language [11]. Our reliability property is a particular kind of reachability property expressed by the PCTL formula $true\ U\ state = s_R$, where U is the *Until* temporal operator.

The vector $\bar{x}$ whose entries x_i correspond to the probabilities of reaching s_R from state s_i is computed as solution of the linear equation system in variables $\{x_i | s_i \in S\}$:

$$x_i = \begin{cases} 1 & \text{if } s_i = s_R \\ 0 & \text{if } s_i \neq s_R \text{ is absorbing} \\ \sum_{s_j \in S} \mathbf{P}(s_i, s_j) \cdot x_j & \text{otherwise} \end{cases} \quad (1)$$

thus the item x_0 corresponds to the probability of the PCTL formula $true\ U\ state = s_R$ evaluated in the initial state.

In our DTMC model, some transitions are labeled with *variable* probability values. These transitions belong to two classes. One class represents unknown or changing behaviors exhibited by the environment, which may influence the overall behavior of the application, and hence satisfaction of the overall requirements. An example may be the set of transitions exiting a given state, which represent different options that a user may select in an interactive state, whose labels represent the probabilities that different choices are selected. In the realistic case where user profiles may change, transition probabilities are labeled as variables. The value of such variables (representing the disturbances) is dynamically computed by monitoring the application at run-time, collecting concrete real values, and then inferring the associated probability through a learning component like the one introduced in Figure 1. The other class represents *control* variables (already introduced earlier), whose value is generated by the controller and is input to the system's model (see also Figure 1). These variables can be assigned any value compatible with design requirements and the constraints determined by their nature. That is, every control variable can assume a value in the set $[0, 1]$ and the sum of the probabilities of the transitions leaving a certain state must be one.

[1]It is also possible to extend the definition by introducing an *initial distribution* $\bar{d}$ providing for each state s_i the probability $\bar{d}_i$ that the process will start there. The same effect can be achieved in our case by introducing a special initial state s_0 such that $\forall s_i\ P(s_0, s_i) = \bar{d}_i$.

[2]In fact, one could add a single success state reachable with probability 1 by every other state considered as successful in the general case and obtain the mentioned case.

Our objective is thus to design a controller that decides system's settings (i.e. decides control variables) given the current situation (i.e. knowledge of system structure and measures or estimates of environment situation) in order to keep the system satisfying its requirements. This objective can be achieved by exploiting well established control theoretical instruments, with a number of additional features relevant for the assessment of actual software quality, as will be explained in Sections IV and V.

A. A Representative Example

In this section we introduce a simple running case study, consisting of a model for an image processing application. The high level software model is shown in Figure 2. The purpose of the system is to apply a filter to incoming images, followed by a beautifying post-processing phase. It is equipped with three different implementations of the filter: 1) direct filtering via internal software, 2) iterative filtering via internal software, and 3) direct filtering via outsourcing to an external service. The DTMC system model is provided in Figure 3. The figure shows that all operations have a certain probability of failure (represented by transitions entering state SF). State $S1$ represents the point of choice between the different filtering options. The probabilities that govern this choice and the probability of applying one more iteration after the execution of the iterative filter (represented by state $S2$) are the control variables in our setting. Control variables are indicated by probability variables Ci in Figure 3 (referring to Figures 2 and 3, c_{1a} is the probability of choosing *the iterative filter*, c_{1b} is the probability of choosing *the internal direct filter* and thus $1 - c_{1a} - c_{1b}$ the probability of outsourcing; c_5 is the probability of requesting another iteration of the iterative filter). These values can be changed online by the controller while the software is executing. The controller in fact observes the overall behavior (i.e., the overall probability of success or failure) and tries to guarantee the requested global reliability requirement by adjusting the invocation probabilities.

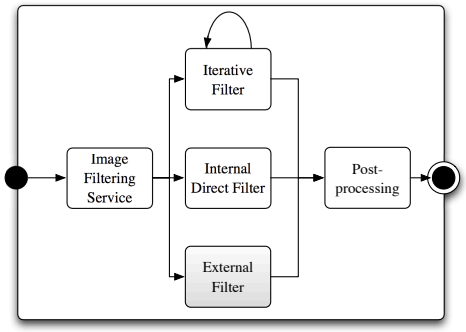

Fig. 2. Schema of the software system.

We assume that all the alternatives are implemented by black-box services that can be invoked and observed from outside only. For each of these services, a run-time monitor collects failure (or success) rates and estimates its reliability as the probability that a invocation to the service will fail[3]. It is necessary to postulate in the environment the existence of monitoring instruments . In fact, the reliability of the computational units is time-varying and the overall reliability depends on these values. Even if their nominal values are known at design time, unpredictable events could alter them, altering as a consequence also the software behavior. This is not uncommon, since the alteration could for example simply come from sharing components with other customers, so that, at different times, their availability depends on load conditions of computational resources. Service reliability for each server are thus just observable values subject to variations during time (disturbances).

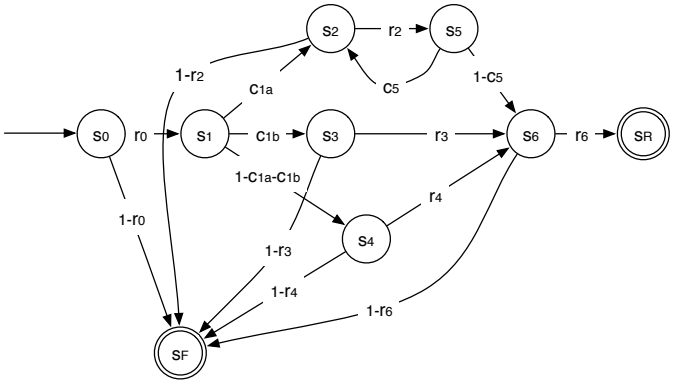

Fig. 3. DTMC mode for the example system.

By solving the equation system (1) for $\bar{x}_0$ it is possible to obtain a closed formula that describes the explicit dependency of reliability (s) on control variables ($\mathbf{c}$) and measured reliabilities ($\mathbf{r}$).

$$s = r_0 \cdot r_6 \cdot \left(\frac{c_{1a} \cdot (-1 + c_5) \cdot r_2}{-1 + c_5 \cdot r_2} + c_{1b} r_3 + (1 - c_{1a} - c_{1b}) \cdot r_4 \right) \quad (2)$$

The formula of s shown in the Equation will be later used to design the controller in Sections IV and V[4].

IV. Software Models as Dynamic Systems

In this section we show how the dynamic evolution of the running software, as observed via the corresponding DTMC model, can be cast in the simple control-theoretical framework of discrete-time dynamic systems [15], through which we achieve self-adaptation of the behavior to react to changing conditions in the environment. Due to space limitations, the background theory can not be fully stated here, but the

[3]Estimates are here assumed to be statistically correct [12] and representative of the average or worst case, depending on the desired analysis scenario. Interested readers can refer to [3] for a deeper discussion about DTMC parameters estimation at runtime, which is out of the scope of this paper.

[4]The same formula can be obtained by exploiting state of the art techniques from parametric model-checking and DTMC analysis in [13], [14]

interested reader can refer to books such as [16]. A discrete-time dynamic system is described by the equations

$$\begin{cases} x(k+1) &= f\left(x(k), u(k), d_x(k)\right) \\ y(k) &= g\left(x(k), u(k), d_y(k)\right) \end{cases} \tag{3}$$

where $x \in \Re^{n_x}$, $u \in \Re^{n_u}$ and $y \in \Re^{n_y}$ are called respectively the state, input and output vectors, $d_x \in \Re^{n_x}$ and $d_y \in \Re^{n_y}$ the state and output disturbance vectors (being those zeros if there are no variations with respect to the nominal conditions), $f(\cdot,\cdot,\cdot)$ and $g(\cdot,\cdot,\cdot)$ are real-valued vector functions of convenient dimensions, and k is an integer index counting the instants – not necessarily evenly spaced in time – when the system undergoes an evolution step. In a more general form, $f(\cdot,\cdot,\cdot)$ and $g(\cdot,\cdot,\cdot)$ could depend on an arbitrary number of real-valued parameters, possibly time-varying. The term "step k" denotes the time span between the k-th and the $(k+1)$-th instant.

Vector d_x represents disturbances corresponding to observable values in the environment that affect the system's state. Vector d_y accounts for measurement errors of the controlled variables.

The first equation in (3) is called the *state equation*, and dictates what the system state will be at the end of step k given what it was at the beginning, and given also the actions exerted on the system in that step, that are assumed to be correctly summarized by the values of u and d_x at its beginning. The state equation represents the dynamic system's character as *difference* equations, i.e., owing to the contextual presence of two subsequent index values. In other words, the state equation gives the system "memory of the past", and explains why the same action generally yields different effects depending on the system condition when it is applied. The input vector u represents *manipulated variables*, that can be used to influence the system's behavior, while the state disturbance d_x accounts for any action other than u, i.e., for any external entity that actually influences the system state, and that in some cases can possibly be measured, but never manipulated.

The second equation in (3) is instead called the *output equation*. It is not dynamic, as shown by the presence of a single index value, and in most problems of interest it describes what one measures (vector y) to appreciate the system's behavior. The disturbance vector d_y represents possible alterations of said measurements, due e.g. to noise, but *not* of the actual evolution of the system state.

A key concept of control theory is that modeling an object in the form (3) improves *per se* insight into that object by providing a formal model for its dynamic evolution. In fact, doing so naturally leads to distinguishing whether the same action yielded a different outcome than the previous time it was applied because the modeled object was in a different state, or something other than that affected it, or some of its parameters changed, or any combination thereof [15], [17]. As is well known for control system design, such a distinction is of paramount importance when *controlling* the system, as trying to "counteract the wrong cause" for an undesired effect can simply be disruptive. Also, models like (3) inherently give a quantitative and generally tractable meaning to the idea that "the system's reaction to a stimulus is related to its previous conditions".

On a similar front, observe that a model like (3) is concerned not only with the condition that the system can possibly reach under constant inputs as $k \to \infty$, but also with the way the system "moves" in time. More generally, in the absence of disturbances and assuming complete knowledge of the system (i.e., of $f(\cdot,\cdot,\cdot)$, $g(\cdot,\cdot,\cdot)$), the *initial state* $x(k_0)$ and the *input trajectory* $u(k)$, $k \geq k_0$, uniquely determine the *state trajectory* $x(k)$ and the *output trajectory* $y(k)$ for $k \geq k_0$. Disturbances and/or time-variances may alter that nominal behavior, and ultimately motivate the use of feedback, as explained in the next section.

Let us now specialize the general framework to the case of control-oriented software modeling. Suppose that the adaptation mechanism, no matter how designed, acts at instants identified by an index k, as in (3). Also, let the average duration of a step be significantly longer than the time scale of the controlled system's dynamics. Translating from the control jargon to the case of interest, this means for example that if at the beginning of a step some controller altered the transition probabilities of the DTMC, then at the end of the same step the effects of our actions can be *measured*. Hence, in this case model (3) reduces to

$$s(k+1) = \tilde{f}(r(k) + \Delta r(k), c(k)) \tag{4}$$

where $s(k+1)$ is the reliability in step k (the state, as defined in the general model of Equation 3), $\mathbf{c}(k)$ are the control variables set for step k (i.e., decided at its beginning and kept constant through the step), $\mathbf{r}(k)$ the expected reliabilities for step k (which in this example are estimated via monitoring, but in other case may also be nominal and possibly constant values), and $\Delta\mathbf{r}(k)$ accounts for any discrepancy between the real and expected reliabilities in step k. The form of function $\tilde{f}$ stems from the DTMC model solving the associated linear system (1), as for equation (2) in Section III-A.

Although model (4) is nonlinear and time-varying, it has the very interesting property of being a "pure delay" system, i.e., the state vector does not appear on the right hand side of the state equation, the output being s itself. In system-theoretical terms, what is here done is taking the steady-state model coming from DTMCs and using it in a dynamic framework under the assumption that any action at the beginning of a step has exhausted its effect at the end of that step.

V. Controlling the System's Dynamics by Feedback

In a nutshell, the idea of feedback can be summarized as plugging the controlled system into a larger one where its input is made dependent on its measured output, possibly its state or an estimation of it in the case it cannot be measured, and on the desired behavior for the controlled system. Additionally, disturbance measurements can be brought into play if available. Recalling model (3), this means in general setting up a control law in the form:

$$\begin{cases} x_c(k+1) &= f_c\big(x_c(k), w(k+1), \widehat{y}(k), \\ & \quad \widehat{x}(k), \widehat{d}_x(k), \widehat{d}_y(k)\big) \\ u(k+1) &= g_c\left(x_c(k), w(k+1), \widehat{y}(k)\right) \end{cases} \quad (5)$$

where the hat symbol recalls that in the real world only measurements (or estimates) of some quantities are available (i.e., from now on, $\widehat{q}$ means a measured or estimated value of q, for any variable q). In (5) x_c is the controller's state vector (x in Equation 3 that refers to a general system), w the *set point* – i.e., the desired behavior for (part of) the controlled system's state and output, e.g. the desired reliability. Notice that the controller state and the desired behavior are assumed to be known exactly, which is mere common sense. The control law can be defined explicitly or, as it is the case here, implicitly, stating the controller's state and output to be the solution of an optimization problem. Notice also that the effect of the control applied at the k-th step is measurable and visible in the feedback signal at time $k+1$.

By joining (3) and (5) we obtain a closed-loop system. Its inputs are the set point w and the disturbances d_x and d_y, its state is the union of x and x_c, and its output can be taken as the set of variables for which a desired behavior is specified. If the controller is properly designed, formal guarantees can be provided on the behavior of the closed loop, also in the presence of time variances and/or disturbances. Due to the lack of space, it is impossible here to report the underlying theory, therefore we move directly to the application for the solution of the addressed problem, for further theoretical details the reader can refer to [18]–[20].

Referring to the DTMC model as the software application model, there are transition probabilities that can be controlled (the control variables) and others that are not dependent on any action but are observable and measurable during software execution (disturbances), e.g. software failures. As previously anticipated, in the following the controllable transition probabilities are identified by variables $\mathbf{c}(k)$, i.e., the control variables. At the same time the reliabilities $\mathbf{r}(k)$ are (disturbed) non controllable transition probabilities.The obtained model is therefore in the form (4) and w is the target value for s.

Based on the current value $s(k)$, an estimation of the future value $\hat{s}(k+1)$ is available. Such an estimation is obtained by substituting the estimated or measured reliabilities in the function f and using the control variables computed for step k (notice that any measurements of s include the effect of d_y):

$$\hat{s}(k) = f(\hat{\mathbf{r}}(k), \mathbf{c}(k)). \quad (6)$$

Now let $J(k) = f_j(\mathbf{c})$ be a cost function on the control variables $\mathbf{c}(k)$. For example, consider the two control variables $c_{1a}(k), c_{1b}(k)$ that stand for the probabilities of sending the incoming request to three different services (the third one is constrained to be $1 - c_{1a}(k) - c_{2b}(k)$), the cost of those values could be the cost of sending the request to each of the available services. When there is no mapping between the software the DTMC is modeling and a cost function naturally derived from the domain of the application[5], the designer can introduce a non informative cost function (such as a constant value) to indicate no preferences among all the feasible solutions.The controller comes into play by solving the optimization problem

$$\min J(\mathbf{c}) \quad (7)$$

subject to the constraints

$$\begin{gathered} ||w - \hat{s}(k)|| \leq \alpha ||w - s(k-1)|| \\ \forall c_i(k), 0 \leq c_i(k) \leq 1 \end{gathered} \quad (8)$$

where α is a value in the range $(0, 1)$ that affects the convergence rate of the solution, that is in the next step we expect the absolute error to be reduced by a factor α. The set of constraints has to be extended with probabilistic constraints (the sum of outgoing transitions from each state has to be 1), as done for the control variables c_i.

Formal assessment

To start, notice that for each step k where a solution of the optimization problem (7) exists, the error $w - s(k)$ has an exponential decay. This is obtained by construction, based on how the controller was designed (first constraint in (8)). Moreover, under the same assumption, the time to converge to the desired solution is known. Let $e(0)$ be the initial error $w - s(0)$, then $e(k) = \alpha^k e(0)$, according to the exponential decay. If one assumes the system converged when the error $e(k) \leq \varepsilon$, then in nominal conditions this happens when:

$$k \geq \log_\alpha \frac{\varepsilon}{e(0)}. \quad (9)$$

If the system is no more in nominal conditions, i.e., if $\hat{\mathbf{r}}(k)$ is not the value for $r(k)$ expected at design-time, the proposed solution is robust whenever a solution is found for the optimization problem. In fact, in such a case, the first constraint of (8) holds. In the ideal case, $s(k) = \hat{s}(k)$. Suppose now that there is an additive term, due to a difference in the estimation of $\mathbf{r}(k) = \hat{\mathbf{r}}(k) + \Delta\mathbf{r}(k)$, and therefore $s(k) = \hat{s}(k) + \Delta s(k)$. The error norm becomes $||w - \hat{s}(k) - \Delta s(k)||$ and the following equations hold

$$\begin{gathered} ||w - \hat{s}(k) - \Delta s(k)|| \leq ||w - \hat{s}(k)|| + ||\Delta s(k)|| \\ ||w - \hat{s}(k)|| \leq \alpha ||w - s(k-1)|| \end{gathered} . \quad (10)$$

Notice that the second equation of (10) comes from the existence of a solution for the optimization problem. As a consequence, in the presence of a solution the stability still holds, while the convergence time equation does not hold anymore. However, if

$$||\Delta s(k)|| < (1 - \alpha) ||w - s(k-1)|| \quad (11)$$

the error norm is guaranteed to diminish, albeit not at the (unfeasible) desired rate.

Some words deserve to be spent on the role of α. The closer α is to zero, the faster is the system convergence. However, when the error is closer to zero there could be

[5]Cost estimation methodologies in the context of software reliability is an open research field. The interested reader could refer, for example, to [21].

oscillations when changes occur, as testified by the inequality (11). On the contrary, when α approximates one the system convergence is slow, the oscillations could occur potentially before the error approaches zero, but are less intense. Notice however, that the performed analysis is very powerful and we are using the triangle inequality to upper estimate the norm of the difference with the sum of the norms. This is definitely a coarse overbound, and makes the proposed assessment remarkably conservative. Such a characteristic is however shared by numerous robustness-related control-theoretical results, and historically accepted in exchange for easily applicable criteria. The interested reader can find a discussion in [22].

The situation in which the optimization problem has no feasible solutions triggers the intervention of a higher level controller (or even a human operator) because there is no control variable assignment that can further reduce the error. However, the employed solver goes *as close as possible* to the unfeasible constraint, therefore reaching the optimum value that is reachable in the system conditions. Although a complete treatment of the matter is deferred to future works, an intermediate – and all in all effective – solution can be to employ reliability estimates to possibly recompute the set point as the feasible value nearest to the desired one. If this is done, the previously devised results apparently apply, and permit a possibly suboptimal but very simple management of the situation.

VI. Experimental Evaluation

As anticipated, the design of the control system starts from the closed formula expression defined in equation (2). For the proposed case study the control system acts minimizing

$$J(\mathbf{c}) = (J_{1a}c_{1a} + J_{1b}c_{1b} + J_5c_5)^2 \tag{12}$$

where J_{1a}, J_{1b} and J_5 are equal to one, therefore assuming all costs are equals. In the case study, the first constraint of equation (8) is considered with an equal sign, supposing that the requirement is that the system expose exactly a certain reliability.

In our first experiment, the reliability required over time is changed to show how the controller reacts to changes in the desired value. The simulation is divided into four different slots, each having 25 time units. During the first slot, the reliability requirement is set to 0.7, while in the following one it is 0.8. In the third slot, the desired reliability is 0.5 and it increases to 0.6 in the last slot. All these numbers are feasible, considering the reliabilities of the involved services.

Figure 4 depicts the overall system reliability (s) over time. Figure 5 shows the control signals, c_{1a} is represented with a dashed line, while c_{1b} is the continuous curve; the dashed dotted line shows how c_5 changes over time.

Perturbations to the nominal model were added in the form of disturbances to the services reliabilities r_i, in order to show the controller convergence previously formally proved. The expressions of the reliabilities are

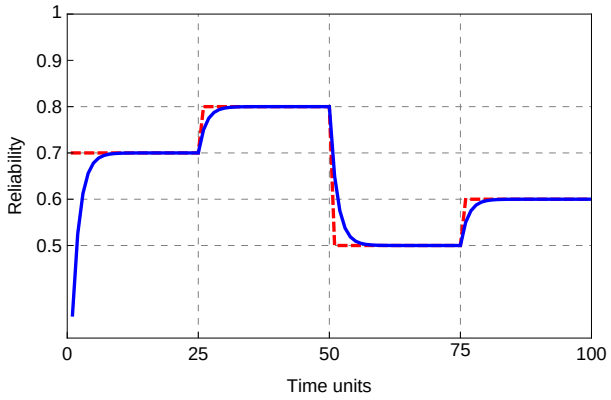

Fig. 4. Reliability of the system: set point (dashed) and achieved value (solid).

$$\begin{aligned} r_0 &= 0.95 + 0.02stp(k-25) - \\ &\quad 0.20stp(k-50) + 0.10stp(k-75) \\ r_2 &= 0.95 + 0.02stp(k-20) - \\ &\quad 0.20stp(k-70) + 0.15stp(k-85) \\ r_3 &= 0.95 + 0.02stp(k-15) - \\ &\quad 0.97stp(k-55) + 0.50stp(k-65) \\ r_5 &= 0.95 \\ r_8 &= 0.95 + 0.05stp(k-95) \end{aligned} \tag{13}$$

where stp represents the step function[6] and k counts the time units. The changes in the reliabilities are introduced to test the control system ability to respond to external variations. Notice that r_3 goes to zero in the time interval $55 \leq k \leq 65$ therefore accounting for the complete failure of the internal non-iterative filter. The control system sharply counteracts the failure of the *internal direct filter*, changing the control variables as can be noted in Figure 5 at time $k = 55$.

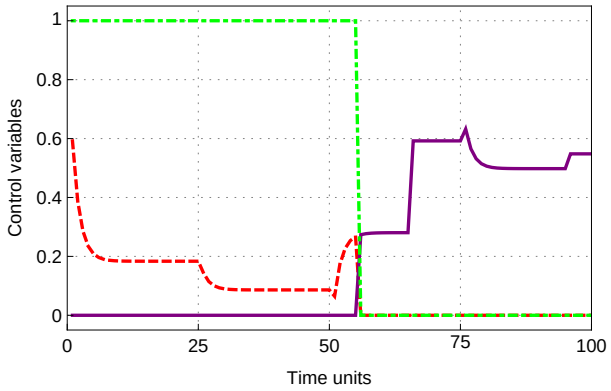

Fig. 5. Control variables of the system: c_{1a} dashed, c_{1b} solid and c_5 dashed dotted.

This experiment allows us to test the response to both transient behaviors, e.g., small variations of the operating conditions, and to changes of the operative scenario, e.g., the complete failure of a service.

[6] $stp(k) = 1$ if $k \geq 0$ and 0 otherwise.

One may also consider the cost of a service as a time varying parameter of the control system. We tested the same system where the cost of the iterative filter, J_{1b}, becomes 100 in the interval $70 \leq k \leq 90$. Figure 6 shows the control variables in this case. Notice that the reliability set point is attained, obtaining the same results shown in Figure 4. With this test we demonstrated that the system is able to attain the set point specification and to minimize the cost of the overall solution, also in the presence of different operating conditions; for example changes of service costs.

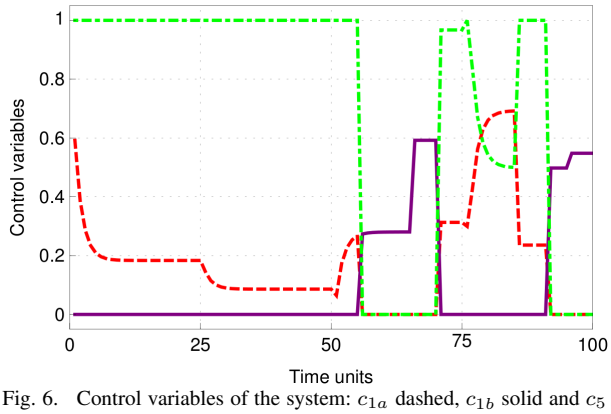

Fig. 6. Control variables of the system: c_{1a} dashed, c_{1b} solid and c_5 dashed dotted.

Notice that, intuitively, Figures 5 and 6 are equal except for the mentioned interval where costs are modified. In Figure 6 the control system changes the transition probabilities (to make the software system perform as in Figure 4) according to the differences in the cost function for the diverse software stages. The overall cost is therefore minimized.

VII. Related Work

Adaptation is playing a key role in the development of software applications [23], [24]. Compiler-level advancements have been developed to support adaptive implementations for performance [25], [26] or power [27], [28], and low level architectures are dynamically adjusted and targeted [29]–[32].

Control theory [33], [34] is capturing an increasing interest from the software engineering community that looks at self-management as a means to meet QoS requirements despite environmental changes and fluctuations of external phenomena. Examples of this trend can be seen in research on control of web servers [35], [36], data centers and clusters management [37], [38], and operating systems [39]–[43].

Self-management techniques are also prominent in industry; e.g., companies like IBM [44] (see projects like the IBM Touchpoint Simulator, the K42 Operating System [40]), Oracle (Oracle Automatic Workload Repository [45]), and Intel (Intel RAS Technologies for Enterprise [46]).

The application of control theory in software engineering, however, is still in a very preliminary stage. Developing accurate system models for software is in fact hard. Moreover, strong mathematical skills are needed in order to deal with complex non-linear dynamics of real systems [33], [47], [48]. These difficulties usually lead to the design of controllers focused on particular operating regions or conditions and ad hoc solutions that address a specific computing problem using control theory, but do not generalize [36], [49], [50]. For example, in [51] the specific problem of building a controller for a .NET thread pool is addressed.

To the best of our knowledge, control of software through a DTMC model capturing its reliability-related properties was not explored previously. Moreover, the approach proposed in this paper in general for any reachability property over DTMCs, regardless of the specific application. Hence it can be applied in general.

Concerning the control of Markov processes, most of the approaches in literature cover only special cases. A general control approach for Continuous Time Markov Chains (CTMCs) has been proposed by Brockett in 2008 [52]. The goal of the controller is to set the value of control transition rates in order to control selected transition rates, through minimizing a quadratic cost function over the controls. We are considering Brockett's model to apply the methodology presented in this paper for performance-driven adaptation of software systems.

VIII. Conclusions

The application of control theory to the systematic construction of adaptive software is a challenging and potentially very valuable approach. This paper has just scratched the surface by focusing on adaptation of a specific class of models (Discrete Time Markov Chains—DTMCs) for a specific class of requirements that need to be preserved (global reliability expressed through a reachability property), specific phenomena whose changes may lead to requirements violations (changes that may be expressed as updates in certain probabilities associated with transitions), and a specific way to attempt adaptation by controlling the model (generating new values for other probabilities, which represent control variables).

Future research will need to generalize beyond all the "specifics". For example, future research should be dealing with other formal models that may address other classes of non-functional properties (e.g., Continuous Time Markov Chains to deal with performance or Markov Reward Models to deal with costs) and with multi-objective goals to achieve and preserve. Moreover, on the modeling side, the overall chain could be modeled introducing also other aspects of the software execution, e.g. service rates. The resulting model would become nonlinear since the state is directly affected by the control variables (the inputs) in a multiplicative way. This brings into play more complex control solutions and opens interesting perspectives also for control researchers.

Acknowledgment

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

REFERENCES

[1] L. Baresi and C. Ghezzi, "The disappearing boundary between development-time and run-time," in *FSE/SDP - Future of software engineering research*. New York, NY, USA: ACM, 2010, pp. 17–22.

[2] G. Blair, N. Bencomo, and R. France, "Models@ run.time," *Computer*, vol. 42, no. 10, pp. 22 –27, 2009.

[3] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Model evolution by run-time parameter adaptation," in *ICSE*, 2009.

[4] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[5] R. C. Cheung, "A user-oriented software reliability model," *IEEE TSE*, vol. 6, no. 2, pp. 118–125, 1980.

[6] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik, "Early prediction of software component reliability," in *ICSE, Leipzig, Germany, May 10-18, 2008*. ACM, 2008, pp. 111–120.

[7] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola, "Reliability analysis of component-based systems with multiple failure modes," in *CBSE*, 2010, pp. 1–20.

[8] K. Goseva-Popstojanova and K. S. Trivedi, "Architecture-based approach to reliability assessment of software systems," *Performance Evaluation*, vol. 45, no. 2-3, pp. 179–204, 2001.

[9] R. Reussner, H. W. Schmidt, and I. Poernomo, "Reliability prediction for component-based software architectures," *JSS*, vol. 66, no. 3, pp. 241–252, 2003.

[10] J. Kemeny, J. Snell, and A. Knapp, *Denumerable Markov chains*. Springer, 1976.

[11] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *FAC*, vol. 6, no. 5, pp. 512–535, 1994.

[12] W. Pestman, *Mathematical statistics: an introduction*. Walter de Gruyter, 1998.

[13] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *ICSE*, 2011.

[14] E. Hahn, H. Hermanns, and L. Zhang, "Probabilistic reachability for parametric markov models," *Model Checking Software*, pp. 88–106, 2009.

[15] W. Levine, *The control handbook*. CRC Press, 2005.

[16] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems, 6th Edition*. Pearson, 2009.

[17] J. Doyle, B. Francis, and A. Tannenbaum, *Feedback control theory*. Basingstoke, UK: MacMillan, 1992.

[18] K. Åström and T. Hägglund, *Advanced PID Control*. Research Triangle Park, NC: ISA - The Instrumentation, Systems, and Automation Society, 2005.

[19] A. O'Dwyer, *Handbook of PI And PID Controller Tuning Rules*, 2nd ed. Imperial College Press, 2006.

[20] F. Lewis and V. Syrmos, *Optimal Control*, 2nd ed. John Wiley & Sons, 2004.

[21] H. Pham, "Software reliability and cost models: Perspectives, comparison, and practice," *EJOR*, vol. 149, no. 3, pp. 475–489, 2003.

[22] M. Morari and E. Zafiriou, *Robust process control*. Upper Saddle River, NJ: Prentice Hall, 1989.

[23] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *FOSE*, 2007, pp. 259 –268.

[24] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. Mller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, "Software engineering for self-adaptive systems: A research roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. LNCS, B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer Berlin / Heidelberg, 2009, vol. 5525, pp. 1–26.

[25] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger, "A framework for adaptive algorithm selection in STAPL," in *ACM PPoPP*. New York, NY, USA: ACM, 2005, pp. 277–288.

[26] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "PetaBricks: A language and compiler for algorithmic choice," in *ACM PLDI*, 2009.

[27] W. Baek and T. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *ACM PLDI*, 2010.

[28] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger, "Eon: a language and runtime system for perpetual systems," in *SenSys*, 2007, pp. 161–174.

[29] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 318–329.

[30] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *ASPLOS*, 2009, pp. 253–264.

[31] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Processor power reduction via single-isa heterogeneous multi-core architectures," *Computer Architecture Letters*, vol. 2, no. 1, pp. 2–2, 2003.

[32] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core fusion: accommodating software diversity in chip multiprocessors," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 186–197, 2007.

[33] J. L. Hellerstein, "Self-managing systems: A control theory foundation," *IEEE LCN*, vol. 0, pp. 708–708, 2004.

[34] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury, *Feedback Control of Computing Systems*. Wiley, 2004.

[35] M. Kihl, A. Robertsson, M. Andersson, and B. Wittenmark, "Control-theoretic analysis of admission control mechanisms for web server systems," *The World Wide Web Journal, Springer*, vol. 11, no. 1-2008, pp. 93–116, 2007.

[36] C. Lu, Y. Lu, T. Abdelzaher, J. Stankovic, and S. Son, "Feedback control architecture and design methodology for service delay guarantees in web servers," *IEEE TPDS*, vol. 17, no. 9, pp. 1014–1027, 2006.

[37] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck, "From data center resource allocation to control theory and back," *IEEE CLOUD*, vol. 0, pp. 410–417, 2010.

[38] D. Kusic and N. Kandasamy, "Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems," *Cluster Computing*, vol. 10, pp. 395–408, 2007, 10.1007/s10586-007-0022-y.

[39] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski, "Performance and environment monitoring for continuous program optimization," *IBM J. Res. Dev.*, vol. 50, no. 2/3, pp. 239–248, 2006.

[40] O. Krieger, M. Auslander, B. Rosenburg, R. W. J. W., Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig, "K42: Building a complete operating system," in *EuroSys '06: Proc. of the 1st ACM SIGOPS/EuroSys Euro. Conf. on Computer Systems*, 2006.

[41] S. Oberthür, C. Böke, and B. Griese, "Dynamic online reconfiguration for customizable and self-optimizing operating systems," in *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*. New York, NY, USA: ACM, 2005, pp. 335–338.

[42] C. Karamanolis, M. Karlsson, and X. Zhu, "Designing controllable computer systems," in *Proceedings of the 10th conference on Hot Topics in Operating Systems*. Berkeley, CA, USA: USENIX Association, 2005, pp. 9–15.

[43] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva, "Controlling software applications via resource allocation within the heartbeats framework," in *CDC*, 2010, pp. 3736–3741.

[44] IBM Inc., "IBM autonomic computing website," http://www.research.ibm.com/autonomic/, 2009.

[45] Oracle Corp., "Automatic Workload Repository (AWR) in Oracle Database 10g," http://www.oracle-base.com/articles/10g/AutomaticWorkloadRepository10g.php.

[46] Intel Inc., "Reliability, availability, and serviceability for the always-on enterprise," www.intel.com/assets/pdf/whitepaper/ras.pdf, 2005.

[47] R. Dorf and R. Bishop, *Modern control systems*. Prentice Hall, 2008.

[48] X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, P. Padala, and K. Shin, "What does control theory bring to systems research?" *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 62–69, 2009.

[49] M. Tanelli, D. Ardagna, and M. Lovera, "LPV model identification for power management of web service systems," in *IEE MSC*. Boston, MA: IEEE Control Systems Society, 2008, pp. 1171–1176.

[50] Q. Sun, G. Dai, and W. Pan, "LPV model and its application in web server performance control," in *CSSE*, vol. 3. Washington, DC, USA: IEEE Computer Society, 2008, pp. 486–489.

[51] J. Hellerstein, V. Morrison, and E. Eilebrecht, "Applying control theory in the real world: Experience with building a controller for the .net thread pool," *Sigmetrics Performance Evaluation Review*, pp. 38–42, 2009.

[52] R. Brockett, "Optimal control of observable continuous time markov chains," in *Decision and Control, 2008. CDC 2008. 47th IEEE Confer-*