

Dependability of Adaptable and Evolvable Distributed Systems

Carlo Ghezzi^(✉)

DEIB, DeepSE Group, Politecnico di Milano,
Piazza Leonardo da Vinci, 32, 20133 Milano, MI, Italy
`carlo.ghezzi@polimi.it`

Abstract. This article is a tutorial on how to achieve software evolution and adaptation in a dependable manner, by systematically applying formal modelling and verification. It shows how software can be designed upfront to tolerate different sources of uncertainty that cause continuous future changes. If possible changes can be predicted, and their occurrence can be detected, it is possible to design the software to be self-adaptable. Otherwise, continuous evolution has to be supported and continuous flow into operation has to be ensured. In cases where systems are designed to be continuously running, it is necessary to support safe continuous software deployment that guarantees correct operation in the presence of dynamic reconfigurations. The approaches we survey here have been mainly developed in the context of the SMScom project, funded by the European Commission –Programme IDEAS-ERC (<http://erc-smscom.dei.polimi.it/>) – and lead by the author. It is argued that these approaches fit well the current agile methods for development and operations that are popularized as DevOps.

Keywords: Distributed, ubiquitous, pervasive systems · Cyber-physical systems · Environment uncertainty · Requirements · Software evolution · Dynamic reconfiguration

1 Introduction and Motivations

Modern software systems increasingly live in a dynamic and open world [3]. The goals to fulfil and the requirements to meet evolve over time. The environment in which the software is embedded often behaves in ways that cannot be predicted upfront during design. And if it can, it might later change after the software has been developed and became operational. This situation is often encountered in the design of *cyber-physical systems*, in which the physical and the cyber worlds are intertwined, through many kinds of devices behaving as sensors and actuators. Interaction with the physical world introduces a great variety of possible contingencies, like noise, vibrations, humidity, or temperature, which may unexpectedly affect the system’s behavior. Sensors and actuators may also behave in a hard-to-predict manner, and this may change over time, e.g. their behavior may change because of the battery level. For these reasons, many different kinds

of uncertainty may be present when the system is being designed and uncertainty may ultimately affect the system's ability to satisfy the requirements.

Design uncertainty is increasingly becoming the norm also for many other kinds of system. User-intensive, highly interactive systems depend on users' behaviors, which also may change over time. The widespread availability of virtual environments, providing infrastructure/software-as-a-service, which raise the level of abstraction for system designers, add their own sources of uncertainty that must be properly handled. Furthermore, modern systems are increasingly multi-owner. They depend upon parts (components, services) that are not under the developers' full control, but rather they are owned, managed, and operated by others. They may run on platforms that developers do not own and do not run; for example, they may run on a cloud. Yet, software designers are responsible for the service they provide to their clients, and the level of service they must guarantee has to satisfy the contractual agreements they subscribed with their customers.

Requirements volatility and environment uncertainty are two main causes that drive software evolution. Software evolution is not a new problem. It has been recognized as a key distinguishing factor of software with respect to other technologies since the early work pioneered by Belady and Lehman since the 1970's [4, 17], although the phenomenon has reached today unprecedented levels of intensity. In the past software evolution was often viewed as a nuisance. The term *maintenance* was often used to capture the evolution of software needed to remedy inadequate requirements and wrong design choices. Evolution is instead intrinsic in software. Like evolution in nature, it has a positive connotation, which refers to the ability to adapt and improve in quality.

Today software is developed through evolutionary processes. Traditional pre-defined, monolithic, *waterfall* lifecycles are generally replaced by incremental, iterative, evolutionary, *agile* processes. Agility indicates a fast and flexible way to react to changes. At the same time, researchers developed approaches to embed in software capabilities to drive its own evolution, in an *autonomic* or *self-managed* manner [14].

Agile processes originated in the practitioners' world and have only been marginally investigated by researchers. As observed by [19], in their iconoclastic reaction to other approaches, the proponents of agile methods tend to dismiss some of the key principles of software engineering that lead to improved dependability. They dismiss requirements analysis —replaced by user stories— (formal) modelling —viewed as a sterile exercise— and the value of formal verification —fully replaced by continuous testing. The importance of formal methods in the context of self-managed systems has also been largely underestimated by most initial research efforts.

The body of work we survey in this paper is fully reliant on formal methods to enable dependable software evolution. Within the vast area of software evolution, this article focuses mostly on two aspects:

1. *Non-functional requirements*: The system's evolution is dictated by the need to satisfy certain non-functional requirements in the presence of changes that

would otherwise lead to violations. Among non-functional requirements we focus in particular on those that can be modelled in a mathematically precise, quantitative way. This includes requirements on response time, reliability, power consumption. Often these can be expressed in a probabilistic manner.

2. *Self-adaptation*: We analyze when and how the system can be made capable of collecting and analyzing run-time data that hint at changes in the behavior of the environment that may lead to requirements violations and are amenable to reactions that may be decided autonomously.
3. *Dynamic software updates*: In the case of self-adaptation, the system must reconfigure itself dynamically, while it is operational. This requirement also holds for systems where updates are performed offline by software engineers and installed online while the system is offering service. This situation is becoming very common today because many systems are required to be continuously running and operation cannot be interrupted to accommodate new updates. Dynamic updates must be performed both safely and efficiently, to ensure timely reaction to changes.

The paper is structured as follows. Section 2 presents a general framework to understand and reason about evolution and adaptation. In particular, it allows us to articulate the complex interactions that may occur between the software and the environment in which it is embedded, and how dependency on the environment may affect dependability and drive adaptation. Section 3 introduces a case study. Section 4 introduces background material on modeling and verification. Section 5 discusses how models and verification may be brought to run time to support self-adaptation. Section 6 addresses the problem of safe dynamic software updates. Finally, Sect. 7 illustrates final considerations and points to future research.

2 Reference Framework

In this section we describe a framework to understand and reason about software and change, which was proposed by the foundational work on requirements engineering developed by Jackson and Zave [13, 21]. Jackson and Zave observe that in requirements engineering one needs to carefully distinguish between two main concerns: the *world* and the *machine*. The machine is the system of interest that must 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. The goals to be met and the *requirements* are ultimately dictated by the world and must be expressed in terms of the phenomena that occur in it. Some of these phenomena are shared with the machine: they are either controlled by the world and observed by the machine –through sensors– or controlled by the machine and observed by the world –through actuators. The machine is built exactly for the purpose of achieving satisfaction of the requirements in the real world. Its *specification* is a prescriptive statement of the relation on *shared phenomena* that must be enforced by the system to be

developed. The machine that implements it must be correct with respect to the specification.

The task of software engineers is to develop first a specification and then an implementation for a machine that achieves requirements satisfaction. To this end, *domain* or *environment knowledge* plays an essential role. That is, the software engineer needs to understand the laws that govern the behavior of the environment and formulate the set of relevant assumptions that have to be made about the environment in which the machine is expected to work, which affect the achievement of the desired results. Quoting from [21],

“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 knowledge, assuming that S and D are both satisfied and consistent with each other, the designer’s responsibility is ultimately to ensure that

$$S, D \models R$$

i.e., the machine’s specification S we are going to devise must entail satisfaction of the requirements R in the context of the domain properties D . We call this the *dependability argument*.

Figure 1 provides a visual sketch of the Jackson/Zave approach. The domain knowledge D plays a fundamental role in establishing the requirements. We need to know upfront how the environment in which the software is embedded works, since the software to develop (the machine) can achieve the expected requirements only based on the assumptions on the behavior of the domain, described by D . Should the environment behave in a way that contradicts the statements in D , the current specification might lead to violation of R . The statements expressed by D may fail to capture the environment’s behavior for two reasons: either because the domain analysis was initially flawed (i.e., the environment behaves according to different laws than the ones captured by D) or because changes occurred, which cause the assumptions made earlier to become invalid. An example of the latter case may be an exceptional and unexpected traffic of submitted user requests that may generate a denial of service.

It is possible to further breakdown D into two components: *domain laws* – Dl – and *domain assumptions* – Da –. Laws indicate the physical or mathematical properties that have been *proved* for the domain, whose truth can only be invalidated by falsifying the theory. An example is the law of motion that says that the application of a force in a given direction to a body causes motion of the body in that direction. A designer relying on this property may specify that a command to a force actuator has to be issued by the software to satisfy the requirement that a body should be moved. This property holds and cannot be refuted. Assumptions instead are properties that are subject to some level of uncertainty and may be disproved. In some cases, they denote currently valid properties that may later change, as for example, traffic conditions changes. They represent the best of

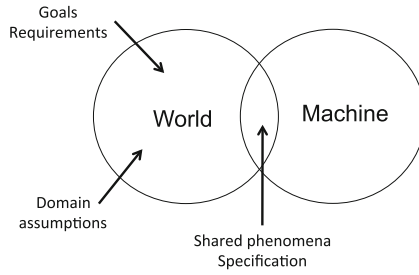


Fig. 1. The Jackson/Zave framework.

our knowledge at a given time. But because of design-time uncertainty and/or because variability in time, assumptions may become invalid.

Software evolution refers to changes that affect the machine, to enable it to respond to changes in the requirements and/or in the environment (we ignore in this paper the fact that implementation may be incorrect, i.e., the running software violates its specification S). The term *adaptation* is used in this work to indicate the specific case of evolution dictated by changes in the environment, while self-adaptation indicates changes that can be handled autonomously by the machine.

The management of evolution in traditional software is performed off-line, during the *maintenance* phase. The traditional classification in perfective, adaptive, and corrective maintenance can also be explained by referring to the Jackson/Zave framework. Changes in the requirements, dictated by changes in the business goals of organizations or new demands by users, cause *perfective maintenance*. Environmental changes affecting domain assumptions, which may represent organizational assumptions or conditions on the physical context in which the software is embedded, cause *adaptive maintenance*. *Corrective maintenance* is instead caused by failure of the dependability argument and forces the specification to change.

According to the traditional paradigm, in order to undergo a maintenance intervention, software returns into its development stage, where changes are analyzed, prioritized, and scheduled. Changes are then handled by modifying the design and implementation of the application. The evolved system is then verified, typically via some kind of regression testing.

This paradigm does not meet the requirements of current application scenarios, which are subject to continuous changes in the requirements and in the environment, and which require rapid reaction to such changes. By following an *agile development* style, software development became incremental and iterative. By following the currently widely advocated *DevOps* culture, agility extends in a seamless manner to delivery and deployment, viewing development and operation as an integrated perpetual process.

Figure 2 illustrates our envisioned process that supports continuous development and operation, through two main, interacting loops: the development loop and the self-adaptation loop. The process incorporates the run-time feedback loop advocated by the autonomic computing proposal [14], which enables self-adaptation. Designers are in the loop and drive evolution. They get informed about the system's dynamic behavior by leveraging monitored data. They are required to initiate evolution whenever self-adaptation fails. Whenever they decide that components should be transferred to the running system to replace faulty functionalities, add functionalities, or enhance existing ones, they can instruct the operational environment to reconfigure itself dynamically in a completely safe, non-disruptive, and efficient way.

In this paper we embrace this holistic view and discuss the role that formal methods can play to support continuous evolution in a dependable manner, i.e., where the designer's focus is constantly driven by the need to formally guarantee satisfaction of the dependability argument.

The next section introduces a practical application domain and a case study that provide concrete motivations for this work. We subsequently show how the run-time adaptation loop can be structured and how safe dynamic reconfigurations can be supported. Finally we will conclude by discussing how we might progress to achieve the global picture of Fig. 2 and by outlining a research agenda.

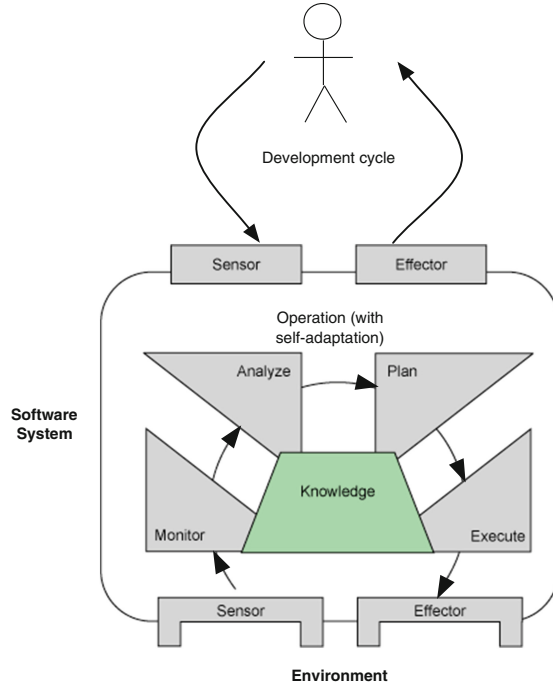


Fig. 2. The development and operation process.

3 A Case Study

Hereafter we illustrate a concrete example in which the approach described earlier is successfully applied. The example, which was originally introduced in [9], refers to a typical e-commerce application that sells merchandise on-line to end users by integrating several services offered by third-parties:

1. *Authentication Service*. This service manages the identity of users. It provides a *Login* and a *Logout* operation through which the system authenticates users.
2. *Payment Service*. This service provides a safe transactional payment service through which users can pay the selected merchandise via the *CheckOut* operation.
3. *Shipping Service*. This 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 *NewCustomer* (NC) or *ReturningCustomer* (RC), based on their usage profile.

The case study illustrates a situation that has become quite common, which is abstracted by Fig. 3. It is a *user-intensive* application, where end-users interact with the application in a hard-to-predict and time variable manner. For example, usage patterns may vary during the different periods of the year, and may have seasonal peaks (for example, around Christmas holidays). Moreover, the behavior of integrated services may be subject to variability, and even deviations from the expected quality of service. Figure 4 provides a high-level view of the flow of interaction between users and the e-commerce application, expressed as an activity diagram.

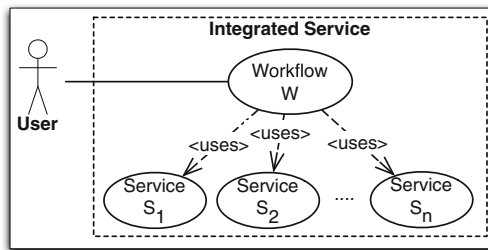


Fig. 3. A class of applications

This application has to guarantee a certain quality of service to customers. In particular, here we focus on *reliability*. Services may in fact fail to provide an answer by timing out incoming requests in situations where the load exceeds their capacity.

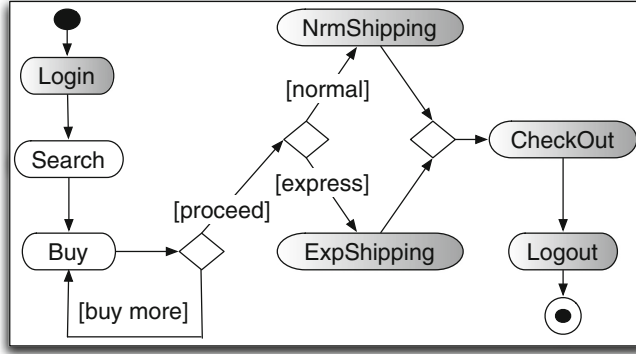


Fig. 4. Operational description of the specification via an activity diagram

Reliability requirements can be typically expressed in probabilistic terms: for example, the probability that a user-triggered transaction completes successfully must be higher than a given value. In the case of the e-commerce application, fulfilment of a reliability requirement clearly depends on certain assumptions about the environment, such as the reliability of the third-party services that are integrated into the application and usage profiles (e.g., the ratio between new and returning customers), which may affect the satisfaction of specific requirements that may refer to the different categories.

As mentioned, environment phenomena of these kinds are quite hard to predict when the system is initially designed. Even in cases where the expected failure rate of services may be stated in the contract with the service provider, values are subject to uncertainty and may very likely change over time (for example, due to a new release of the service). Likewise, usage profiles are hard to predict upfront and are very unstable.

Let us assume that the e-commerce application must satisfy the following reliability requirements:

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

Let us further assume that development time domain analysis tells us that expected usage profile can be reasonably described as in Table 1. The notation $P(x)$ denotes the probability of “ x ”. Table 2 instead summarizes the results of domain analysis concerning the external services integrated in the e-commerce application. $P(Op)$ here denotes the probability of failure of service operation Op . The environment assumptions expressed in Tables 1 and 2 may derive from different sources. For example, reliability properties of third-party services may be published as part of the service-level agreement with service providers. Usage profiles may instead be derived from previous experience of the designers or knowledge extracted from previous similar systems.

Table 1. Domain assumptions on usage profiles

Description	Value
$P(\text{User is a RC})$	0.35
$P(\text{RC chooses express shipping})$	0.5
$P(\text{NC chooses express shipping})$	0.25
$P(\text{RC searches again after a buy operation})$	0.2
$P(\text{NC searches again after a buy operation})$	0.15

Table 2. Domain assumptions on external services

Description	Value
$P(\text{Login})$	0.03
$P(\text{Logout})$	0.03
$P(\text{NrmShipping})$	0.05
$P(\text{ExpShipping})$	0.05
$P(\text{CheckOut})$	0.1

4 Modeling and Verification Preliminaries

As we discussed earlier, the software engineer’s goal is to derive a specification S which leads to satisfaction of requirements R , assuming that the environment behaves as described by D . From the activity diagram in Fig. 4 and the information contained in the tables regarding environment assumptions, we can derive an enriched state-machine model that summarizes a formal description both of the application and of the environment. The state machine transitions describe the possible sequences of interactive operations, according to the protocol specified by the activity diagram of Fig. 4. Domain assumptions are modeled as probabilities that label the transitions. The model also represents failure and success states for the external services.

Formally, the model in Fig. 5 is a Discrete Time Markov Chain (DTMC). 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 internal logical states (e.g., state 2).

Once a formal model is provided, like the DTMC in Fig. 5, it is possible to formally verify whether requirements are satisfied, provided they are expressed in suitable language for which a verification procedure exists. In the case of DTMCs, requirements may be formalized using the probabilistic temporal logic language PCTL, and then checked against the model using a probabilistic model checker, like PRISM [12, 16]. By doing so on our example, we obtain the following results:

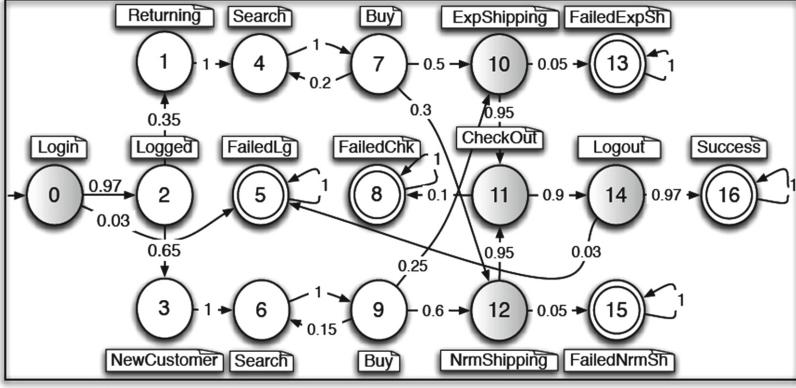


Fig. 5. DTMC model for the case study

- Probability of success = 0.804
- Probability of a *ExpShipping* failure for a user recognized as *ReturningCustomer* = 0.031
- Probability of an authentication failure (i.e., *Login* or *Logout* failures) = 0.056

which ensure satisfaction of the requirements.

Hereafter we explain how this can be done, by first briefly reviewing DTMCs and then introducing PCTL.

4.1 Discrete Time Markov Chains

DTMCs are defined as state-transition systems augmented with probabilities. *States* represent possible configurations of the system. *Transitions* among states occur at discrete time and have an associated probability. DTMCs are discrete stochastic processes with the Markov property, according to which the probability distribution of future states depend only upon the current state.

Formally, a (labeled) DTMC is tuple (S, S_0, P, L) where

- S is a finite set of states
- $S_0 \subseteq S$ is a set of initial states
- $P : S \times S \rightarrow [0, 1]$ is a stochastic matrix, where $\sum_{s' \in S} P(s, s') = 1 \ \forall s \in S$. An element $P(s_i, s_j)$ represents the probability that the next state of the process will be s_j given that the current state is s_i .
- $L : S \rightarrow 2^{AP}$ is a labeling function which assigns to each state the set of *Atomic Propositions* which are true in that state.

For reasons that will become clear later, we implicitly extend this definition by also allowing transitions to be labeled with variables (with values in the range 0..1) instead of constants. A state $s \in S$ is said to be an *absorbing state* if $P(s, s) = 1$. If a DTMC contains at least one absorbing state, the DTMC itself is said to be an *absorbing DTMC*.

In an absorbing DTMC with r absorbing states and t transient states, rows and columns of the transition matrix P can be reordered such that P is in the following *canonical form*:

$$\mathbf{P} = \begin{pmatrix} Q & R \\ 0 & I \end{pmatrix}$$

where I is an r by r identity matrix, 0 is an r by t zero matrix, R is a nonzero t by r matrix and Q is a t by t matrix.

Consider now two distinct transient states s_i and s_j . The probability of moving from s_i to s_j in exactly 2 steps is $\sum_{s_x \in S} P(s_i, s_x) \cdot P(s_x, s_j)$. Generalizing, for a k -steps path and recalling the definition of matrix product, it follows that the probability of moving from any transient state s_i to any other transient state s_j in exactly k steps corresponds to the entry (s_i, s_j) of the matrix Q^k . As a natural generalization, we can define Q^0 (representing the probability of moving from each state s_i to s_j in 0 steps) as the identity t by t matrix, whose elements are 1 iff $s_i = s_j$ [10].

Due to the fact that R must be a nonzero matrix, and P is a stochastic matrix, Q has uniform-norm strictly less than 1, thus $Q^n \rightarrow 0$ as $n \rightarrow \infty$, which implies that eventually the process will be absorbed with probability 1.

In the simplest model for reliability analysis, the DTMC will have two absorbing states, representing the correct accomplishment of the task and the task's failure, respectively. The use of absorbing states is commonly extended to modeling different failure conditions. For example, different failure states may be associated with the invocation of different external services. Once the model is in place, we may be interested in estimating the probability of reaching an absorbing state or in stating the property that the probability of reaching an absorbing failure state should be less than a certain threshold. In the next section we discuss how these and other interesting properties of systems modeled by a DTMC can be expressed and how they can be evaluated.

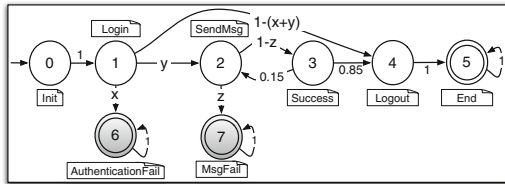


Fig. 6. DTMC example.

Let us consider the simple example of DTMC in Fig. 6, which represents a system sending authenticated messages over the network. States 5, 6, and 7 are absorbing states; states 6 and 7 represent failures associated respectively to the authentication and to message sending. We use variables as transition labels to

indicate that the value of the corresponding probability is unknown, and may change over time.

In matrix form, the same DTMC would be characterized by the following matrices Q and R :

$$Q = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & y & 0 & 1-x-y \\ 0 & 0 & 0 & 1-z & 0 \\ 0 & 0 & 0.15 & 0 & 0.85 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} 0 & 0 & 0 \\ 0 & x & 0 \\ 0 & 0 & z \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

This is a toy example that we use hereafter instead of the more complex original case study to exemplify the approach within a constrained space.

4.2 Formally Specifying Requirements

Formal languages to express properties of systems modeled through DTMCs have been studied in the past and several *model checkers* have been designed and implemented to support property analysis. Through model checking one can verify that a given model (representing domain assumptions and the specification) satisfies the requirements, provided they are formalized in a language, such as PCTL, for which a verification procedure exists. In particular, PCTL [2] –which is briefly introduced hereafter– proved to be useful to express a number of interesting reliability properties.

PCTL extends the branching-time temporal logic language CTL [2] to deal with probabilities. Instead of the existential and universal quantification of CTL, PCTL provides the probabilistic operator $\mathcal{P}_{\bowtie p}(\cdot)$, where $p \in [0, 1]$ is a probability bound and $\bowtie \in \{\leq, <, \geq, >\}$.

PCTL is defined by the following syntax:

$$\begin{aligned} \Phi &::= \text{true} \mid a \mid \Phi \wedge \Phi \mid \neg \Phi \mid \mathcal{P}_{\bowtie p}(\varphi) \\ \varphi &::= X \Phi \mid \Phi U \Phi \mid \Phi U^{\leq t} \Phi \end{aligned}$$

Formulae Φ are named *state formulae* and can be evaluated over a boolean domain (true, false) in each state. Formulae ψ are named *path formulae* and describe a pattern over the set of all possible paths originating in the state where they are evaluated.

The satisfaction relation for PCTL is defined for a state s as:

$$\begin{aligned}
s &\models \text{true} \\
s &\models a && \text{iff } a \in L(s) \\
s &\models \neg\Phi && \text{iff } s \not\models \Phi \\
s &\models \Phi_1 \wedge \Phi_2 && \text{iff } s \models \Phi_1 \text{ and } s \models \Phi_2 \\
s &\models \mathcal{P}_{\bowtie p}(\psi) && \text{iff } Pr(s \models \psi) \bowtie p
\end{aligned}$$

A formal definition of how to compute $Pr(s \models \psi)$ is presented in [2]. The intuition is that its value corresponds to the fraction of paths originating in s and satisfying ψ over the entire set of paths originating in s . The satisfaction relation for a path formula with respect to a path π originating in s ($\pi[0] = s$) is defined as:

$$\begin{aligned}
\pi &\models X\Phi && \text{iff } \pi[1] \models \Phi \\
\pi &\models \Phi U \Psi && \text{iff } \exists j \geq 0. (\pi[j] \models \Psi \wedge (\forall 0 \leq k < j. \pi[k] \models \Phi)) \\
\pi &\models \Phi U^{\leq t} \Psi && \text{iff } \exists 0 \leq j \leq t. (\pi[j] \models \Psi \wedge (\forall 0 \leq k < j. \pi[k] \models \Phi))
\end{aligned}$$

PCTL is an expressive language that allows reliability-related properties to be specified. A taxonomy of all possible reliability properties is out of the scope of this paper. The most important case is a *reachability property*. A reachability property states that a state where a certain characteristic property holds is eventually reached from a given initial state. In most cases, the state to be reached is an absorbing state. Such state may represent a *failure state*, in which a transaction executed by the system modeled by the DTMC eventually (regrettably) terminates, or a *success state*. Reachability properties are expressed as $\mathcal{P}_{\bowtie p}(\text{true } U \Phi)$ ¹, which expresses the fact that the probability of reaching any state satisfying Φ has to be in the interval defined by constraint $\bowtie p$. Φ is assumed to be a simple state formula that does not include any nested path formula. In most cases, it just corresponds to the atomic proposition that is true in an absorbing state of the DTMC. In the case of a failure state, the probability bound is expressed as $\leq x$, where x represents the upper bound for the failure probability; for a success state it would be instead expressed as $\geq x$, where x is the lower bound for success.

PCTL allows more complex properties than plain reachability to be expressed. Such properties would be typically domain-dependent, and their definition is delegated to system designers. For example, referring to the example in Fig. 6, we express the following reliability requirements:

- **R1:** “The probability that a *MsgFail* failure happens is lower than 0.001”
- **R2:** “The probability of successfully sending at least one message for a logged in user before logging out is greater than 0.001”
- **R3:** “The probability of successfully logging in and immediately logging out is greater than 0.001”
- **R4:** “The probability of sending at least 2 messages before logging out is greater than or equal to 0.001”

¹ Note that this is often expressed as $\mathcal{P}_{\bowtie p}F\Phi$, using the *finally* operator.

Notice that **R1** is an example of reachability property. Also notice that these requirements have different sets of initial states: **R1**, **R3**, and **R4** must be evaluated starting from state 0 (i.e., $S_0 = \{0\}$) while **R2** must be evaluated starting from state 1. Formalization of requirements R1-R3 using PCTL is left as an exercise.

5 Supporting Self-adaptation via Run-Time Verification

Let us refer to the process model represented in Fig. 2, which shows the interplay between the run-time adaptation and the off-line evolution feedback loops. To support dependable self-adaptation, we root the analysis phase taking place during operation (see Fig. 2) into model checking. The model that represents both the software system and the environment –such as the one shown in Fig. 6– is kept alive at run time and is updated according to the data gathered by monitoring, which can be used to infer possible environment changes through a machine learning component. In our case study, new values for the probabilities of certain transitions representing service failures may be inferred by monitoring the failure rate of service invocations. Likewise, user profiles may be inferred by monitoring log-in customers' data. Inference can be based on standard statistical approaches, like the Bayesian learning method we used in [6]. Once the model is updated, the properties of interest can be checked. Violation of a given property is a trigger for self-adaptation, which is successful if changes of the implementation can be found that can eliminate the problem through a dynamic reconfiguration.

The key concepts upon which this approach is based are that (1) the models of interest are kept at run time and continuously updated, and (2) model checking provides continuous verification support to detect the need for adaptive reactions. Reactions are often subject to hard real-time constraints: they must lead to a valid software reconfiguration before the violation of requirements leads to unacceptable mishaps. The conventional model checking techniques are not really suitable for use at run time. They require the model checker to be run from scratch after any model change. It is thus necessary to re-think model checking algorithms to make them suitable for run-time use.

Our work has focused on making DTMC model checking for PCTL *incremental*. An incremental approach avoids re-analysis of the entire model by pre-computing the effects of changes. To achieve this goal, we make the assumption that changes are local and not disruptive. This is a reasonable assumption in most practical cases, assuming that the source model for the update is a reasonable approximation of the target. For DTMC models this assumption boils down to the hypothesis that the structure of the model does not change: only transition parameters may change. Furthermore, although in principle all such parameters may change, the solution we found works very efficiently if the number of transition parameters that may change is a small fraction of all transitions.

In the next section we present an incremental approach to probabilistic model checking that is based on parameterization. Changeable transition probabilities are treated as variables and a mathematical procedure computes a symbolic analytic expression for the properties we want to verify at run time. The underlying

idea is that computation of the analytic expression, which takes place at design time, can be computationally expensive, but then evaluation of the pre-computed analytic expression, which occurs at run time, can be very efficient.

5.1 Run-Time Efficient Parametric Model Checking

The most commonly studied property for reliability analysis concerns the probability of reaching a certain state, which typically represents the success of the system or some failure condition. Both success and failure are modeled by absorbing states. The reachability formula in this case has the following form: $\mathcal{P}_{\bowtie p} Fl$, where l is the label of the target absorbing state. Hereafter we focus our discussion on how to pre-compute at design time a reachability formula for an absorbing state of a DTMC. All the details and the extension of the approach to cover all PCTL can be found in [7, 8].

We assume that a DTMC can contain both numerically and symbolically labeled transitions. Since the sum of probabilities of all transitions exiting any given state must be 1, in the case where one transition is a variable, we require that all transitions exiting the state be also variable. We refer to such state as *variable state*.

For an absorbing DTMC, the matrix $I - Q$ has an inverse N and $N = I + Q + Q^2 + \dots = \sum_{i=0}^{\infty} Q^i$ [10]. The entry n_{ij} of N represents the expected number of times the Markov chain reaches state s_j , given that it started from state s_i , before getting absorbed. Instead, q_{ij} represents the probability of moving from the transient state s_i to the transient state s_j in exactly one step.

Given that $Q^n \rightarrow 0$ when $n \rightarrow \infty$ (as discussed in Sect. 4.1), the process will always be absorbed with probability 1 after a large enough number of steps, no matter from which state it started off. Hence, our interest is to compute the probability distribution over the set of absorbing states. This distribution can be computed in matrix form as:

$$B = N \times R$$

where r_{ik} is the probability of being absorbed in state s_k given that the process started in state s_i .

B is a $t \times r$ matrix and it can be used to evaluate the probability of each termination condition starting from any DTMC state as an initial state. In particular the element b_{ij} of the matrix B represents the probability of being absorbed into state s_j given that the execution started in state s_i .

The design-time computation of an entry b_{ij} requires mixed symbolic and numeric computation, since variable states may be traversed to reach state s_j . Let us evaluate the complexity of such computation. Inverting matrix $I - Q$ by means of the Gauss-Jordan elimination algorithm [1] requires t^3 operations. The computation of the entry b_{ij} once N has been computed requires t more products, thus the total complexity is $t^3 + t$ arithmetic operations on polynomials. The computation could be further optimized by exploiting the sparsity of $I - Q$. Notice that the symbolic nature of the computation makes the design-time phase quite costly [11].

The complexity can be significantly reduced if the number of variable components c is small and the matrix describing the DTMC is sparse, as very frequently happens in practice. Let $W = I - Q$. The elements of its inverse N are defined as follows:

$$n_{ij} = \frac{1}{\det(W)} \cdot \alpha_{ji}(W)$$

where $\alpha_{ji}(W)$ is the cofactor of the element w_{ji} . Thus:

$$b_{ik} = \sum_{x \in 0..t-1} n_{ix} \cdot r_{xj} = \frac{1}{\det(W)} \sum_{x \in 0..t-1} \alpha_{xi}(W) \cdot r_{xj}$$

Computing b_{ik} requires the computation of t determinants of square matrices with size $t - 1$. Let τ be the average number of outgoing transitions from each state ($\tau \ll n$ by assumption). Each of the determinants can be computed by means of Laplace expansion. Precisely, by expanding first the c rows representing the variable states (each has τ symbolic terms), we need to compute at most τ^c determinants and then linearly combine them. Each submatrix of size $t - c$ does not contain any variable symbol, by construction, thus its determinant can be computed with $(t - c)^3$ operations among constant numbers (LU-decomposition), thus much faster than the corresponding ones among polynomials. The final complexity is thus:

$$\tau^c \cdot (t - c)^3 \sim \tau^c \cdot t^3$$

which significantly reduces the original complexity and makes the design-time pre-computation of reachability properties feasible in a reasonable time, even for large values of t .

As a term of comparison, the computation of reachability properties performed by probabilistic model-checkers is based on the solution of a system of n equations in n variables [2], which has, in a sequential computational model, a complexity equal to n^3 [5].

Summing up, we discussed the computation of properties in the form $\mathcal{P}_{\bowtie p}(Fs_k)$, where s_k is an absorbing state, starting in any initial transient state of the system². With this procedure, it is possible to obtain closed formulae for a number of interesting reliability properties.

For example, evaluating **R1** on our toy system, that is the probability of reaching the state *MsgFail* failure in any number of execution steps corresponds to evaluating b_{07} as:

$$\mathbf{R1:} \quad \frac{(yz)}{(0.85 + 0.15z)} \leq 0.001$$

The approach can be extended to computing the probability of successfully reaching a non-absorbing state. This extension supports verification of properties like “the probability of reaching state s_j without reaching any failure” or “the

² Actually we discussed the computation of the probability associated with the property, to which the constraint $\bowtie p$ has to be applied.

probability of a successfully performing a certain operation or service”. In our example, the probability of reaching the *Logout* state 7 after any number of steps is expressed by the following formula: $f_{04} = \frac{0.85-0.85x+0.15z-0.15xz-yz}{0.85+0.15z}$. This extension, as well as the ones needed to cover the entire PCTL are presented in [8].

6 Achieving Safe Dynamic Software Update

Once the need for a change in running software is identified, an alternative solution has to be found and then instantiated. A number of different approaches have been proposed to address the problem, focusing on changes at different levels of granularity. In this section we assume that the implementation has a distributed component-based architecture, where components interact via remote invocations. We do not address here the issue of how the alternative solution may be identified, but instead focus on how the architectural update may be instantiated at run time in a safe way, while the system is running.

Traditional approaches to software update are static. They require (1) to shut down the currently running version, (2) deploy the new version, and (3) restart the system. This allows safe replacement if off-line verification has proved that the new version satisfies the new requirements, but cannot be applied in the increasingly common cases where the system cannot be shut down and the update must be performed while the system is running.

Dynamic software update must satisfy two main requirements. It has to have *low disruption*, i.e. it must have low overhead and minimize the delay with which the system is updated. It also has to be *safe*, i.e. it must not lead the system into an unexpected erroneous state.

The rest of this section summarizes the work presented in [18], where different criteria for dynamic update are assessed and a new criterion, called *version consistency* is proposed. This criterion leads to a safe and efficient dynamic update approach for distributed component-based architectures.

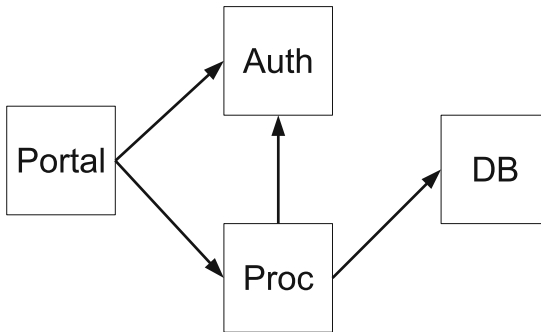


Fig. 7. Our example system.

Let us consider, as an example, the architecture shown in Fig. 7. A portal component (**Portal**) interacts with an authentication component (**Auth**) and a business processing component (**Proc**), while **Proc** interacts with both **Auth** and a database component (**DB**). This means that **Portal** statically depends on (i.e., can invoke) **Proc** and **Auth**, and **Proc** depends on **Auth** and **DB**.

A component can host (execute) transactions. A transaction is a sequence of actions that completes in bounded time. Actions include local computations and message exchanges. A transaction T can be initiated by an outside client or by another transaction T' . T is called a *root transaction* in the former case and a *sub-transaction* (of T') in the latter case. The term $\text{sub}(T', T)$ denotes that T is a direct sub-transaction of T' . The set $\text{ext}(T) = \{x \mid x = T \vee \text{sub}^+(T, x)\}$ is the *extended transaction set* of T , which contains T and all its direct and indirect sub-transactions. The extended transaction set of a root transaction models the concept of *distributed transaction* that can span over multiple components. The host component of transaction T is denoted as h_T . Transactions are also always notified of the completion of their sub-transactions. This implies that a transaction T cannot end before its sub-transactions T_i . All other exchanged messages between h_T and h_{T_i} —because of T_i — are temporally scoped between the two corresponding messages that initiate the sub-transaction and notify its completion.

Figure 8 shows a usage scenario for the example system. The **Portal** first gets an authentication token from **Auth** and then uses it to require services

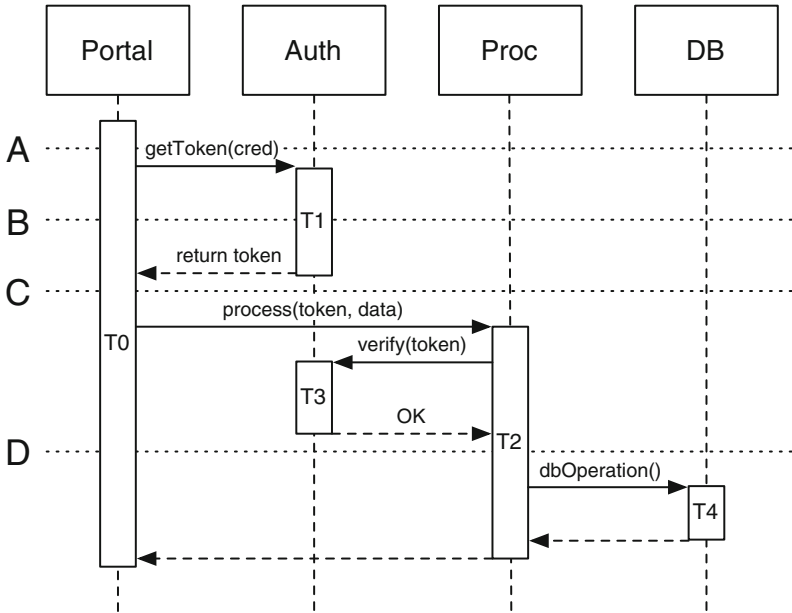


Fig. 8. Detailed scenario.

from **Proc**. **Proc** verifies the token through **Auth** and then starts computing, and interacting with **DB**. If we consider the root transaction T_0 at **Portal**, its extended transaction set is $ext(T_0) = \{T_0, T_1, T_2, T_3, T_4\}$, where T_1 at **Auth** is in response to the `getToken` request, T_2 at **Proc** in response to `process`, T_3 at **Auth** in response to `verify`, and T_4 at **DB** for T_2 's request of database operations.

A dynamic update can be specified as an operation that substitutes one or more components of the original configuration with new versions. We assume components to be stateless; i.e., there is no need to transfer the state from one component to its replacement during the update. We also assume the update to be correct, i.e., the update satisfies the requirements in the current environment conditions. The update leads to a dynamic reconfiguration, where new bindings are established between the existing components and a newly installed component. We assume that re-binding is performed as an atomic operation.

Let \mathbb{S} be the current specification of the requirements to be satisfied by the system, and let \mathbb{S}' be the updated specification that must be satisfied after the update. The dynamic reconfiguration is defined to be correct if:

- The transactions that end before the update satisfy \mathbb{S} ;
- The transactions that begin after the update satisfy \mathbb{S}' ;
- The transactions that begin before the update, and end after it, satisfy either \mathbb{S} or \mathbb{S}' .

In our example, suppose that **Auth** has to be updated to exploit a stronger encryption algorithm and prevent weaknesses in system security. Although the new algorithm is incompatible with the old one, the other components need not to be updated because all encryption/decryption operations are done within **Auth**. If the update is allowed to happen any time, however, it may be impossible to ensure correctness. An obvious restriction on *when* the update can happen is that components targeted for update must be *idle*, that is, they are not hosting transactions. This constraint is a necessary but not sufficient condition for safe dynamic update. In fact, if we consider the scenario of Fig. 8, and substitute **Auth** when idle, but after serving `getToken`, the resulting system would behave incorrectly since the security token would be created with an algorithm and validated by another.

It can be proved that correctness of arbitrary runtime updates is undecidable, even if the corresponding off-line update is correct and the on-line update only happens when components are idle. However, it is possible to derive automatically checkable sufficient correctness conditions.

In a seminal paper, Kramer and Magee [15] proposed a criterion called *quiescence* as a sufficient condition for a component to be safely replaced in dynamic reconfigurations. Their approach models a distributed system as a directed graph, whose nodes represent components and edges represent static dependencies. A node can initiate transactions on itself, or initiate two-party transactions on another node if there is an edge between the two nodes. A node's state can only be affected by transactions. Every two-party transaction is a sequence of message exchanges between the two nodes. A (dependent) transaction T can

“contain” other (consequent) transactions T_i : the completion of T depends on the completion of all the T_i . Transactions always complete in bounded time and the initiator is always notified about their termination.

Definition 1 (Quiescence). *A node is quiescent if:*

1. *It is not currently engaged in a transaction that it initiated;*
2. *It will not initiate new transactions;*
3. *It is not currently engaged in servicing a transaction;*
4. *No transactions have been or will be initiated by other nodes which require service from this node.*

A component node satisfying the first two conditions is said to be *passive*. A node is required to respond to a passivate command from the configuration manager by driving itself into a passive state in bounded time. The last two conditions further make the node independent of all existing or future transactions, and thus it can be manipulated safely. To drive a node into a quiescent status, in addition to passivating it, all the nodes that statically depend on it must also be passivated to ensure the last two conditions.

According to this approach, a node cannot be quiescent before completion of all the transactions initiated by statically dependent nodes. This means that the actual update could be deferred significantly. In our example, **Auth** cannot be quiescent before the end of the transactions initiated by **Portal** and **Proc**. Moreover, all the other nodes that could potentially initiate transactions, which require service from **Auth**, directly or indirectly, are passivated, and their progress blocked till the end of the update. Again, in our example **Portal** and **Proc** are to be passivated. This means that the this approach can introduce significant disruption in the service provided by the system.

To reduce disruption, Vandewoude et al. [20] proposed an alternative criterion, called *tranquillity*. The idea is that there is no need for waiting a transaction to complete if it will not further request the service provided by the node targeted for update, even if the node has been involved in the transaction. Symmetrically, it is also permitted to update a node even if some on-going transactions will require the service provided by the node in the future, but they have not interacted with it yet.

Definition 2 (Tranquillity). *A node is tranquil if:*

1. *It is not currently engaged in a transaction that it initiated;*
2. *It will not initiate new transactions;*
3. *It is not actively processing a request;*
4. *None of its adjacent nodes are engaged in a transaction in which it has both already participated and might still participate in the future.*

If applied to our example, however, tranquillity would lead to unsafe updates. In fact, after **Auth** returns the token to **Portal**, it will not participate in the session initiated by **Portal** anymore. Before the request for verification is sent, **Auth** has not participated in the session initiated by **Proc**. So **Auth** is tranquil at time \textcircled{A} .

However, if **Auth** is updated at this time a failure may occur since the token was issued by the old version of **Auth** with an incompatible encryption algorithm. This failure would not happen if the system was either entirely in the old or in the new configuration.

To conclude, we can say that the quiescence is a general and safe criterion, but it can be disruptive. Tranquillity is less disruptive, but it can be applied safely in a restricted set of cases assumption, otherwise it can be unsafe.

Version consistency is a new criterion introduced in [18], which tries to get the best of the previous two proposals and achieves safety while reducing disruption. The criterion can be stated as follows:

Definition 3 (Version Consistency). *Transaction T is version consistent iff $\nexists T_1, T_2 \in \text{ext}(T) \mid h_{T_1} \in \omega \wedge h_{T_2} \in \omega'$. A dynamic reconfiguration of a system is version consistent if all its transactions are kept version consistent.*

This means that a dynamic reconfiguration of a system is correct if it happens at a time instant where all its transactions, including those started before and ended after the update, are kept version consistent. This is because of the correctness of the old and new configurations and the fact that any version-consistent transaction is served—along with all its sub-transactions—as if it entirely completed within the old or the new configuration, no matter when the update actually happens. Also note that a transaction that ends before (starts after) the update cannot have a direct or indirect sub-transaction hosted by the new (old) version of a component being updated.

For our example, if the update of **Auth** happens after transaction T_0 begins but before it sends a **getToken** request to **Auth**, all transactions in $\text{ext}(T_0)$ (i.e., all transactions in Fig. 8) are served in the same way as if the update happened before they all began. If it happens at any time after **Auth** replies to the **verify** request issued by **Proc** (time \textcircled{B}), all transactions in $\text{ext}(T_0)$ are served the same way as if the update happened after they all ended. However, if it happens at time \textcircled{A} , then $h_{T_1} = \text{Auth}$, but $h_{T_3} = \text{Auth}'$. As both T_1 and $T_3 \in \text{ext}(T_0)$, T_0 would not be version-consistent.

Since version consistency is not directly checkable, we need to identify a condition that is checkable on a component (or a set of components) and that ensures that its (their) runtime update does not break version consistency.

Dynamic dependences are the means to define such a condition, and they can easily be added to the diagram of Fig. 7 through properly-labelled edges besides those that represent the static dependencies. A **static**-labelled edge represents both a static dependence and the communication channel between the two components; **future** and **past** edges represent dynamic dependences. Future and past edges are also labelled with the identifier of a root transaction. We use $C \xrightarrow[T]{\text{future(past)}} C'$ to denote a **future(past)** edge labelled with the identifier of root transaction T , from component C to component C' . This means that because of T , some transactions in $\text{ext}(T)$ hosted by C will use (has used) the service provided by C' by initiating sub-transactions on it.

Definition 4 (Valid Configuration). A valid configuration with dynamic dependences, hereafter configuration, must satisfy the following constraints:

1. (LOCALITY) For each future or past edge $C \xrightarrow[T]{future(past)} C'$ there is a static edge between C and C' ;
2. (FUTURE-VALIDITY) A future edge $C \xrightarrow[T]{future} C'$ must be in place before the first sub-transaction $T' \in ext(T)$, where $T' \neq T$, is initiated, and continues to exist at least until no transactions hosted by C will initiate further $T'' \in ext(T)$ on C' ;
3. (PAST-VALIDITY) A past edge $C \xrightarrow[T]{past} C'$ must be in place at the end of any transaction $T' \in ext(T)$ initiated by a transaction hosted by C on C' and continues to exist at least until the end of T .

Figure 9 shows some configurations of the example system. Active components that are executing a transaction are marked with a *, and numbers correspond to the order with which edges are added.

The configuration of Fig. 9 (A) corresponds to time point ① in Fig. 8: transaction T_0 is executing on Portal, which is *-annotated. The dynamic edges indicate that to serve transactions in $ext(T_0)$, Portal might use Auth and Proc in the future, and also Proc might use Auth and DB. Figure 9 (B) corresponds to time ② and says that a transaction in $ext(T_0)$ (T_1) is currently running on

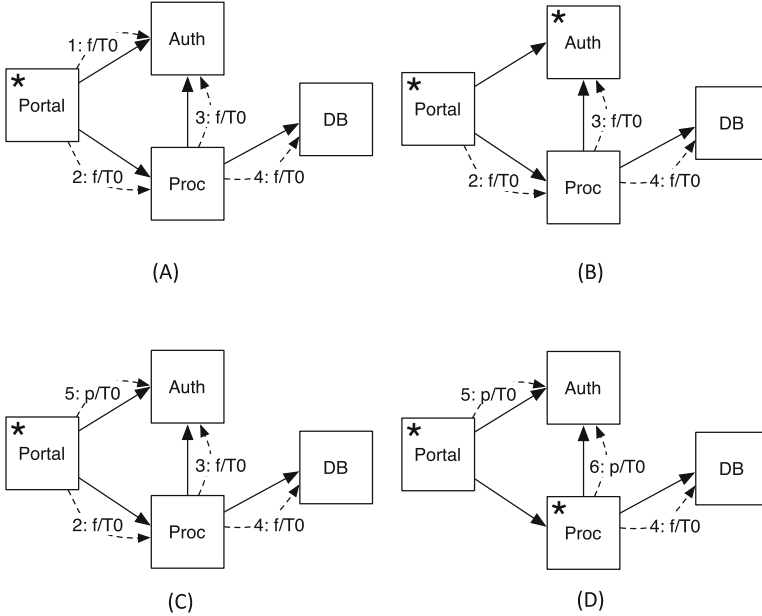


Fig. 9. Some configurations of the example system with explicit dynamic dependencies.

Auth, but no further transaction in $ext(T_0)$ hosted on **Portal** will initiate any sub-transaction on **Auth** anymore because there is no T_0 -labelled future edge between the two nodes. Figure 9 (C), which corresponds to time \odot in Fig. 8, indicates that **Auth** might have hosted transactions in $ext(T_0)$ initiated by **Portal** in the past, and might host further transactions in $ext(T_0)$ initiated by **Proc** in the future. Figure 9 (D) corresponds to time \ominus in Fig. 8 and shows that **Auth**, although it might have hosted transactions in $ext(T_0)$, is not hosting and will not host these transactions anymore.

Given a valid configuration, we can identify a locally checkable condition that is sufficient for the version consistency of dynamic reconfigurations.

Definition 5 (Freeness). *Given a configuration Σ , a component c is said to be free of dependencies with respect to a root transaction T iff c is not hosting any transaction in $ext(T)$ and there does not exist a pair of T -labelled future/past edges entering c . c is said to be free in Σ iff it is free with respect to all the root transactions in the configuration.*

In our example, **Auth** is free with respect to T_0 in the configurations of Fig. 9 (A) and (D), but not in the one of Fig. 9 (C) since there exist two f/T_0 and p/T_0 edges that enter **Auth**. Moreover, since **Auth** is active, it is also not free in Fig. 9. Intuitively, for a valid configuration Σ , the freeness condition for a component c —with respect to a root transaction T —means that the distributed transaction modeled by $ext(T)$ either has not used c yet (otherwise there should be a past edge), or it will not use c anymore (otherwise there should be a future edge). This leads to the following proposition, which is not proved here³.

Proposition 1. *Given a valid configuration Σ of a system, a dynamic update of a component c is version consistent if it happens when c is free in Σ .*

Without entering into details, for which we refer to [18], our solution proposes a distributed algorithm for efficiently managing dynamic dependencies that: (1) keeps the configuration valid and (2) ensures version consistency with limited disruption. Dynamic dependencies are maintained in a distributed way. Each component only has a local view of the configuration that includes itself and its direct neighbors. A component is responsible for the creation and removal of the outgoing dynamic edges, but it is also always notified of the creation and removal of the incoming ones. This is achieved by exchanging management messages that keep the consistency among the views of neighbor components.

The management of dynamic dependencies may slightly delay the execution of the actual transactions, but it guarantees that no transaction will be blocked forever. The underlying message delivery is assumed to be reliable, and the messages between two components are kept in order. Dynamic edges are labelled with the identifiers of the corresponding root transactions to allow for the management of the dynamic edges of a root transaction independently of those of other transactions.

³ A proof can be found in [18].

To assess version consistency we used simulation to evaluate its disruption for a wide set of randomly generated component-based distributed systems that varied in the number of components, service time, and network latency. The results showed that dynamic updates based on version consistency are on average more than 50 % less disruptive than those based on quiescence.

7 Conclusions

The objective of this paper was to give a high-level view of the problems involved in supporting software evolution without compromising its correctness, i.e., continuous requirements satisfaction. After setting the problem of software evolution in the context of Jackson and Zave's framework [13], we dug into the problem of achieving self-adaptation via models and verification at run time. Focusing on requirements that ask for probabilistic models and properties, we have shown how probabilistic model checking can be brought to run time to drive self-adaptation. We have then focused on another important problem that must be solved to support both self-adaptation and also, more generally, any kind of dynamic reconfiguration that is a consequence of evolution.

The approaches presented in this paper are a first step in the direction of integrating development and operation (DevOps), conceived as two interacting feedback loops that are funded on mathematically precise models and continuous formal verification. Models and verification are necessary in both loops, and they must be handled in an iterative and incremental manner. Agile development is often hostile to modeling and verification, sometimes they are even viewed as *deprecated upfront activities* [19]. Requirements are replaced by user stories. Although they realize that continuous verification is necessary, verification is simply equated to testing. Likewise, modeling and verification are often conceived as heavy-weight monolithic processes. For example, verification of partial and incomplete models is seldom supported, while incremental development intrinsically goes through incomplete descriptions. Verification is seldom incremental, support to understanding the effect of changes and reasoning on them is rarely provided. The two worlds, however, should get together, and this urgently calls for a sustained research agenda that goes widely beyond the initial steps presented in the paper.

References

1. Althoen, S.C., McLaughlin, R.: Gauss-Jordan reduction: a brief history. *Am. Math. Monthly* **94**(2), 130–142 (1987)
2. Baier, C., Katoen, J.-P.: *Principles of Model Checking*. The MIT Press, Cambridge (2008)
3. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: issue and challenges. *Computer* **39**(10), 36–43 (2006)
4. Belady, L.A., Lehman, M.M.: A model of large program development. *IBM Syst. J.* **15**(3), 225–252 (1976)

5. Bojanczyk, A.: Complexity of solving linear systems in different models of computation. *SIAM J. Numer. Anal.* **21**(3), 591–603 (1984)
6. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time adaptation. In: *Proceedings of the 31st International Conference on Software Engineering*, pp. 111–121. IEEE Computer Society (2009)
7. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: *Proceedings of the 33rd International Conference on Software Engineering* (2011)
8. Filieri, A., Tamburrelli, G., Ghezzi, C.: Supporting self-adaptation via quantitative verification and sensitivity analysis at run time. *IEEE Trans. Softw. Eng.* **42**(1), 75–99 (2016)
9. Ghezzi, C., Tamburrelli, G.: Reasoning on non-functional requirements for integrated services. In: *Proceedings of the 17th International Requirements Engineering Conference*, pp. 69–78. IEEE Computer Society (2009)
10. Grinstead, C., Snell, J.: *Introduction to probability*. Amer Mathematical Society, Providence (1997)
11. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric markov models. In: Păsăreanu, C.S. (ed.) *Model Checking Software*. LNCS, vol. 5578, pp. 88–106. Springer, Heidelberg (2009)
12. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: a tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) *TACAS 2006*. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
13. Jackson, M., Zave, P.: Deriving specifications from requirements: an example. In: *ICSE 1995: Proceedings of the 17th international conference on Software engineering*, pp. 15–24, New York, NY, USA. ACM (1995)
14. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *IEEE Comput.* **36**(1), 41–50 (2003)
15. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. *IEEE Trans. Softw. Eng.* **16**(11), 1293–1306 (1990)
16. Kwiatkowska, M., Norman, G., Parker, D.: Prism 2.0: a tool for probabilistic model checking. In: *Proceedings of First International Conference on the, Quantitative Evaluation of Systems, QEST 2004*, pp. 322–323 (2004)
17. Lehman, M.M., Belady, L.A. (eds.): *Program Evolution: Processes of Software Change*. Academic Press Professional Inc., Cambridge (1985)
18. Ma, X., Baresi, L., Ghezzi, C., Manna, V.P.L., Lu, J.: Version-consistent dynamic reconfiguration of component-based distributed systems. In: *ESEC/FSE 2011: The 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and the 13rd European Software Engineering Conference*, pp. 245–255. ACM (2011)
19. Meyer, B.: *Agile!: The Good, the Hype and the Ugly*. Springer Science and Business Media, Berlin (2014)
20. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: a low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Softw. Eng.* **33**(12), 856–868 (2007)
21. Zave, P., Jackson, M.: Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.* **6**(1), 1–30 (1997)

Formal Methods for the Quantitative Evaluation of
Collective Adaptive Systems

16th International School on Formal Methods for the
Design of Computer, Communication, and Software
Systems, SFM 2016, Bertinoro, Italy, June 20-24, 2016,
Advanced Lectures

Bernardo, M.; De Nicola, R.; Hillston, J. (Eds.)

2016, VII, 261 p. 67 illus., Softcover

ISBN: 978-3-319-34095-1