Perpetual Assurances in Self-Adaptive Systems

D. Weyns, N. Bencomo, R. Calinescu, J. Camara, C. Ghezzi, V. Grassi, L. Grunske, P. Inverardi, J.M. Jezequel, S. Malek, R. Mirandola, M. Mori, and G. Tamburrelli

In recent years, researchers have started studying the impact of self-adaptation on the engineering process [1][2][3]. The insight was that today's iterative, incremental, and evolutionary software engineering processes do not meet the requirements of many contemporary systems that need to handle change during system operation. In self-adaptive systems change activities are shifted from development time to runtime, and the responsibility for these activities is shifted from software engineers or system administrators to the system itself. Therefore the traditional boundary between development time and runtime blurs, which requires a complete reconceptualization of the software engineering process. An important aspect of the software engineering process of self-adaptive systems–in particular business or safety critical systems–is providing new evidence that the system requirements are satisfied during its entire lifetime, from inception to and throughout operation. This evidence must be produced despite the uncertainty that affects the requirements and/or the behavior of the environment, which lead to changes that may affect the system, its goals, and its environment [4][5]. To provide guarantees that the system requirements are satisfied, the state of the art in self-adaptive systems advocates the use of formal models as one promising approach, see e.g., [6][7][8][9][29]. Some approaches employ formal methods to provide guarantees by construction. More recently, the use of probabilistic models to handle uncertainties has gained interest. These models are used to verify properties and support decision-making about adaptation at runtime. However, providing assurances that the goals of self-adaptive systems are achieved during the entire life cycle remains a difficult challenge.

In this paper, we provide a background framework and the foundation for an approach to providing assurances for self-adaptive systems that we coined "perpetual assurances for self-adaptive systems." We elaborate on the challenges of perpetual assurances, requirements for solutions, realization techniques and mechanisms to make solutions suitable, and benchmark criteria to compare solutions. We then present a case study that researchers and practitioners can use to assess, challenge and compare their approaches to providing assurances for self-adaptation. Finally, we conclude the paper with a summary of the main challenges we identified for perpetual assurances.

1. Perpetual assurances

We use the following working definition of "assurances for self-adaptive systems":

Assurances for self-adaptive systems mean providing evidence for requirements compliance; this evidence can be provided off-line (i.e., not directly connected or controlled by the running system) and complemented online (i.e., connected or under control of the running system).

We use the following working definition for "perpetual assurances for self-adaptive systems":

Perpetual assurances for self-adaptive systems mean providing evidence for requirements compliance through an enduring process that continuously provides new evidence by combining system-driven and human-driven activities to deal with the uncertainties that the system faces across its lifetime, from inception to operation in the real world.

Thus, providing assurances cannot be achieved by simply using off-line solutions possibly complemented with online solutions. Instead, we envisage that perpetual assurances will employ a continuous process where humans and the system jointly and continuously derive and integrate new evidence and arguments required to assure stakeholders (e.g., end users and system administrators) that the requirements are met by the self-adaptive system despite the uncertainties it faces throughout its lifetime. Realizing the vision of perpetual assurances for self-adaptive systems poses multiple challenges, which we discuss next. From these challenges, we identify requirements, we overview relevant approaches for assurances and discuss mechanisms to make these approaches suitable for perpetual assurances, and we define benchmark criteria to compare different solutions.

1.1. Key challenges for perpetual assurances

Providing perpetual assurances for self-adaptive systems can be extremely hard. In this section we identify and discuss the main challenges that need to be faced.

The primary underlying challenge stems from *uncertainty*. Several definitions of uncertainty can be found in the literature, ranging from absence of knowledge to inadequacy of information and deficiency of the modeling process [10][11]. In the following we adopt the general definition of uncertainty in modeling given in [11] and used in [12]: "*any deviation from the unachievable ideal of completely deterministic knowledge of the relevant system.*" Such deviations can lead to an overall "lack of confidence" in the obtained results, based on a judgment that they might be "incomplete, blurred, inaccurate, unreliable, inconclusive, or potentially false" [13].

Hereafter, we make use of the taxonomy of uncertainty proposed in [12], where uncertainties are classified along three dimensions: *location*, *level,* and *nature*. The *location* dimension refers to where uncertainty manifests in the description (the model) of the studied system or phenomenon. We can specialize it into: (i) input parameters, (ii) model structure, and (iii) context. The *level* dimension rates how uncertainty occurs along a spectrum ranging from deterministic knowledge (level 0) to total ignorance (level 3). Awareness of uncertainty (level 1) and unawareness (level 2) are the intermediate levels. The *nature* dimension indicates whether the uncertainty is due to the lack of accurate information (*epistemic*) or to the inherent variability of the phenomena being described (*aleatory*). Other taxonomies of uncertainties related to different modeling aspects can be found [33][34].

Recognizing the presence of uncertainty and managing it can mitigate its potentially negative effects and increase the level of assurance in a given self-adaptive system. By ignoring uncertainties, one could draw unsupported claims on the system validity or generalize them beyond their bounded scope.

Table 1 shows a classification of uncertainty sources taken from the literature on self-adaptation (e.g., [14][15]). Each source of uncertainty is classified according to the *location* and *nature* dimensions of the taxonomy. The *level* dimension of the taxonomy is not shown in the table since each source of uncertainty can be of any level depending on the implemented capabilities in the system that should deal with the uncertainty.

Table 1 Classification of sources of uncertainty

Source of uncertainty	Classification	
	Location	**Nature**
Simplifying assumptions	Structural/context	Epistemic
Model drift	Structural	Epistemic
Incompleteness	Structural	Epistemic/Aleatory
Future parameters value	Input	Epistemic
Adaptation functions	Structural	Epistemic/Aleatory
Automatic learning	Structural/input	Epistemic/Aleatory
Decentralization	Context/structural	Epistemic
Specification of goals	Structural/input	Epistemic/Aleatory
Future goal changes	Structural/input	Epistemic/Aleatory
Execution context	Context/structural/ input	Epistemic
Noise in sensing	Input	Epistemic/Aleatory
Different sources of information	Input	Epistemic/Aleatory
Human in the loop	Context	Epistemic/Aleatory
Multiple ownership	Context	Epistemic/Aleatory

We have structured the sources of uncertainty into four groups: (i) sources of uncertainty related to the system itself; (ii) uncertainty related to the goals of the system; (iii) uncertainty in the execution context; and (iv) uncertainty related to human aspects. Hereafter we provide a brief definition of each source, focusing on *incompleteness, automated learning,* and *multiple ownership,* which especially apply to self-adaptive systems.

Sources of uncertainty related to the system itself.

- *Simplifying assumptions*: the model, being an abstraction, includes per-se a degree of uncertainty, since details whose significance is supposed to be minimal are not included in the model.
- *Model drift*: as the system adapts, the model of its structure also changes, so it is necessary to keep aligned systems and corresponding models to avoid reasoning on outdated models.
- *Incompleteness* manifests when some parts of the system or its model are knowingly missing. Incompleteness can show up at design time and also at runtime. Progressive completion at design time occurs in iterative, exploratory development processes. Completion at runtime occurs, for example, in dynamic service discovery and binding in service-oriented systems. Managing incompleteness is hard. Performing assurance on incomplete systems requires an ability to discover which sub-assurances must be delayed until completion occurs, and this may even happen at runtime [35]. As a consequence, assurance must be incremental and comply with the time constraints imposed by the runtime environment.
- *Future parameter value*: uncertainty in the future world where the system will execute creates uncertainties in the correct actions to take at present.
- *Automatic learning* in self-adaptive systems usually uses statistical processes to create knowledge about the execution context and most-useful behaviors. These learning processes can lead to applications with uncertain behavior. The location of this uncertainty may be in the model structure or input parameters, depending on how general the concepts are for which the application has been provided with machine-leaning capabilities. The nature of the uncertainty depends on the point of view. From the viewpoint of the application and its models of the world, as the machine has to learn from imperfect and limited data, the nature of the uncertainty is epistemic. From the user viewpoint, since the information undergoes a statistical learning process during model synthesis, automatic learning introduces randomness in the model and analysis results, and consequently it can be seen as an aleatory uncertainty.
- *Decentralization*: this uncertainty comes from the difficulty to keep a model updated if the system is decentralized and each subsystem has self-adaptation capabilities. The reason is that it can be hardly expected that any subsystem could obtain accurate knowledge of the state of the entire system.

Sources of uncertainty related to the goals of the system.

- *Specification of goals*: accurately expressing the preferences of stakeholders is difficult and prone to uncertainty. For example, the runtime impact of a utility function that defines the desired tradeoffs between conflicting qualities may not be completely known upfront.
- *Future goal changes*: the requirements of the system can change due to new needs of the customers, to new regulations or to new market rules, etc.

Sources of uncertainty related to the execution context of the system.

- *Execution context*: the context in which the application operates should be represented in the model, but the available monitoring mechanisms might not suffice to univocally determine this context and its evolution over time.
- *Noise in sensing*: the sensors/monitors are not ideal devices and they can provide, for example, slightly different data in successive measures while the actual value of the monitored data did not change.

Sources of uncertainty related to the humans aspects.

- *Human in the loop*: human behavior is intrinsically uncertain and systems commonly rely on correct human operations – both as system users and as system administrators. This assumption of correct operations may not hold because human behavior can diverge from the correct behavior.
- *Multiple ownership*: systems are increasingly assembled out of parts provided by different stakeholders, whose exact nature and behavior may be partly unknown when the system is composed. This is typical of service-oriented, dynamically composed systems. The nature of this uncertainty is mainly epistemic. However, it may also be aleatory since third-party components may change without notifying the owner of the integrated application.

As discussed above, uncertainty in its various forms represents as the ultimate source of both motivations for and challenges to perpetual assurance. Uncertainty manifests through *changes*. For example, uncertainty in capturing the precise behavior of an input phenomenon to be controlled results in assumptions made to implement the system. Therefore, the system must be calibrated later when observations of the physical phenomenon are made. This in turn leads to changes in the implemented control system that must be scrutinized for assurances.

1.2. High-level requirements for solutions to realize perpetual assurances

The provision of perpetual assurance for self-adaptive systems must satisfy additional requirements compared to those met by assurance solutions for non-adaptive systems. These additional requirements correspond to the challenges summarized in the previous section. In particular, perpetual assurance solutions must cope with a variety of uncertainty sources that depend on the purpose of self-adaptation and the environment in which the assured self-adaptive system operates. To this end, they must build and continually update their assurance arguments through integrating two types of assurance evidence. The first type of evidence corresponds to system and environment elements not affected significantly by uncertainty, and therefore can be obtained using traditional offline approaches [28]. The second type of evidence is associated with the system and environment components affected by the sources of uncertainty summarized in Table 1. This evidence must be synthesized at runtime, when the uncertainty is reduced, quantified or resolved sufficiently to enable such synthesis. The requirements described below stem from this need to treat uncertainty, to generate new assurance evidence, and to use it to update existing assurance arguments.

Requirement 1: *A perpetual assurance solution must continually observe the sources of uncertainty affecting the self-adaptive system.* Without collecting data about the relevant sources of uncertainty, the solution cannot acquire the knowledge necessary to provide the assurance evidence unachievable offline.[1] Addressing this requirement may necessitate, for instance, the instrumentation of system components, the sensing of environmental parameters, and the monitoring of user behavior.

Requirement 2: *A perpetual assurance solution must use its observations of uncertainty sources to continually quantify and potentially reduce the uncertainties affecting its ability to provide assurance evidence.* The raw data obtained through observing the sources of uncertainty need to be used to reduce the uncertainty (e.g., through identifying bounds for the possible values of an uncertain parameter). This uncertainty quantification and potential reduction must be sufficient to enable the generation of the assurance evidence that could not be obtained offline. For instance, rigorous learning techniques such as those introduced in [36][37][38][39] can be used to continually update and refine formal models whose runtime analysis generates the new assurance evidence.

Requirement 3: *A perpetual assurance solution must continually deal with overlapping sources of uncertainty, and may need to treat these sources in a composable fashion.* The

[1] If all necessary assurance evidence can be derived using knowledge available offline, then we assume that the system can be implemented as a non-adaptive system.

sources of uncertainty from Table 1 hardly ever occur apart or independently from each other. Instead, their observed effects are often compounded, consequently inhibiting the system from clearly assessing the extent to which it satisfies its requirements. Therefore, solutions, assurance arguments and the evidence they build upon need to be able to tackle composed sources of uncertainty.

Requirement 4: *A perpetual assurance solution must continually derive new assurance evidence.* Up-to-date knowledge that reduces the uncertainty inherent within the self-adaptive system and its environment must be used to infer the missing assurance evidence (e.g., to deal with model drift or handle goal changes as described in Section 1.2). As an example, new evidence can be produced through formal verification of updated models that reflect the current system behavior [7][8][40][41][42]. Complementary approaches to verification include online testing, simulation, human reasoning and combinations thereof.

Requirement 5: *A perpetual assurance solution must continually integrate new evidence into the assurance arguments for the safe behavior of the assured self-managing system.* Guaranteeing safe behavior requires the provision of assurance arguments that, in the case of self-adaptive systems, must combine offline evidence with new evidence obtained online, once uncertainties are resolved to the extent that enables the generation of the new evidence.

Requirement 6: *A perpetual assurance solution may need to continually combine new assurance evidence synthesized automatically and provided by human experts.* The development of assurance arguments is a complex process that is carried out by human experts and includes the derivation of evidence through both manual and automated activities. Online evidence derivation will in general require a similar combination of activities, although it is conceivable that self-adaptive systems affected by reduced levels and sources of uncertainty might be able to assemble all the missing evidence automatically.

Requirement 7: *A perpetual assurance solution must provide assurance evidence for the system components, the human activities and the processes used to meet the previous set of requirements.* As described above, perpetual assurance for self-adaptive systems cannot be achieved without employing system components and a mix of automated and manual activities not present in the solutions for assuring non-adaptive systems. Each of these new elements will need to be assured either by human-driven or machine-driven approaches. As an example, a model checker used to obtain new assurance evidence arguments at runtime will need to be certified for this type of use.

In addition to the functional requirements summarized so far, the provision of perpetual assurance must be timely, non-intrusive and auditable, as described by the following non-functional requirements.

Requirement 8: *The activities carried out by a perpetual assurance solution must produce timely updates of the assurance arguments.* The provision of assurance arguments and of the evidence underpinning them is time consuming. Nevertheless, perpetual assurance solutions need to complete updating the assurance arguments guaranteeing the safety of a self-adaptive system timely, before the system engages in operations not covered by the previous version of the assurance arguments.

Requirement 9: *The activities of the perpetual assurance solution, and their overheads must not impact the operation of the assured self-adaptive system.* The techniques employed by a perpetual assurance solution (e.g., instrumentation, monitoring and online learning, and verification) must not interfere with the ability of the self-adaptive system to deliver its intended functionality effectively and efficiently.

Requirement 10: *The assurance evidence produced by a perpetual assurance solution and the associated assurance arguments must be auditable by human stakeholders.* Assurance arguments and the evidence they build upon are intended for the human experts responsible for the decision to use a system, for auditing its safety, and for examining what went wrong after failures. In keeping with this need, perpetual assurance solutions must express their new assurance evidence in human-readable format.

Table 2 summarizes the requirements for perpetual assurance solutions.

Table 2 Summary requirements

Requirement	Brief description
R1: Monitor uncertainty	A perpetual assurance solution must continually observe the sources of uncertainty affecting the self-adaptive system.
R2: Quantify uncertainty	A perpetual assurance solution must use its observations of uncertainty sources to continually quantify and potentially reduce the uncertainties affecting its ability to provide assurance evidence.
R3: Manage overlapping uncertainty sources	A perpetual assurance solution must continually deal with overlapping sources of uncertainty and may need to treat these sources in a composable fashion.
R4: Derive new evidence	A perpetual assurance solution must continually derive new assurance evidence arguments.
R5: Integrate new evidence	A perpetual assurance solution must continually integrate new evidence into the assurance arguments for the safe behavior of the assured self-managing system.
R6: Combine heterogeneous evidence	A perpetual assurance solution may need to continually combine new assurance evidence synthesized automatically and provided by human experts.
R7: Provide evidence for the elements and activities that realize R1-R6	A perpetual assurance solution must provide assurance evidence for the system components, the human activities, and the processes used to meet the previous set of requirements.
R8: Produce timely updates	The activities carried out by a perpetual assurance solution must produce timely updates of the assurance arguments.
R9: Limited overhead	The activities of the perpetual assurance solution and their overheads must not impact the operation of the assured self-adaptive system.
R10: Auditable arguments	The assurance evidence produced by a solution and the associated assurance arguments must be auditable by human stakeholders.

1.3. Approaches to realize perpetual assurances

Several approaches have been developed in the previous decades to check if a software system complies with its functional and non-functional requirements. In this section, we give a brief overview of representative approaches. In the next section, we elaborate on mechanisms to make these approaches suitable to support perpetual assurances for self-adaptive systems, driven by the requirements described in the previous section.

Table 3 gives an overview of the approaches we discuss, organized in three groups: human-driven approaches (manual), system-driven (automated), and hybrid (manual and automated).

Table 3 Approaches for assurances

Group	Approaches
Human-driven approaches	- Formal proof - Simulation
System-driven approaches	- Runtime verification - Sanity checks - Contracts
Hybrid approaches	- Model checking - Testing

Human-driven approaches. We discuss two representative human-driven approaches for assurances: formal proof, and simulation and statistical analysis.

Formal proof is a mathematical calculation to prove a sequence of connected theorems or formal specifications of a system. Formal proofs are conducted by humans with the help of interactive or automatic theorem provers, such as Isabelle or Coq. Formal proofs are rigorous and unambiguous. However, they require from the user both detailed knowledge about how the self-adaptive system works and significant mathematical experience. Furthermore, formal proofs only work for highly abstract models of a self-adaptive system (or more detailed models of small parts of it). Still the approach can be useful to build up assurance evidence arguments about basic properties of the system or specific elements (e.g., algorithms). We give two examples of formal proof used in self-adaptive systems. In [46], the authors formally prove a set of theorems to assure atomicity of adaptations of business processes in cross-organizational collaborations. The approach is illustrated in an example where a supplier wants to evolve its business process by changing the policy of payments. In [45], the authors formally prove a set of theorems to assure safety and liveness properties of self-adaptive systems. The approach is illustrated for data stream elements that modify their behavior in response to external conditions through the insertion and removal of filters.

Simulation comprises three steps: the design of a model of a system, the execution of that model, and the analysis of the output. Different types of models at different levels of abstraction can be used, including declarative, functional, constraint, spatial or multimodel. A key aspect of the execution of a model is the way time is treated, e.g., small time increments or progress based on events. Simulation allows a user to explore many different states of the system, without being prevented by an infinite (or at least very large) state space. Simulation runs can provide assurance evidence arguments at different levels of fidelity, which are primarily based on the level of abstraction of the model used and the type of simulation applied. As an example, in [51] the authors simulate self-adaptive systems and analyze the gradual fulfillment of requirements during the simulation. The approach provides valuable feedback to engineers to iteratively improve the design of self-adaptive systems.

System-driven approaches. We discuss three representative system-driven approaches for assurances: runtime verification, sanity checks, and contracts.

Runtime verification is a well-studied lightweight verification technique that is based on extracting information from a running system to detect whether certain properties are violated. Verification properties are typically expressed in trace predicate formalisms, such as finite state machines, regular expressions, and linear temporal logics. Runtime verification is less complex than traditional formal verification approaches, such as model checking, because only one or a few execution traces are analyzed. As an example, in [51] the authors introduce an approach to estimate the probability that a temporal property is satisfied by a run of a program. The approach models event sequences as observation sequences of a Hidden Markov Model (HMM) and uses an algorithm for HMM state estimation, which determines the probability of a state sequence.

Sanity checks are calculations that check whether certain invariants hold at specific steps in the execution of the system. Sanity checks have been used for decades because they are very simple to implement and can easily be disabled when needed to address performance concerns. On the other hand, sanity checks provide only limited assurance evidence arguments that the system is not behaving incorrectly. As an example, in [47], the authors present an approach to manage shared resources that is based on an adaptive arbitrator that uses device and workload models. The approach uses sanity checks to evaluate the correctness of the decisions made by a constraint solver.

Contracts. The notion of Contract (introduced by B. Meyer in the Design by Contract approach in Eiffel) makes the link between simple sanity checks and more formal, precise and verifiable interface specifications for software components. Contracts extend the ordinary definition of abstract data types with pre-conditions, post-conditions and invariants. Contracts can be checked at runtime as sanity checks, but can also serve as a basis for assume-guarantee modular reasoning. In [43] the authors have shown that the notion of functional contract as defined in Eiffel can be articulated in four layers: syntactic, functional, synchronization and quality of service contracts. For each layer, both runtime checks and assume-guarantee reasoning are possible.

Hybrid approaches. We discuss two representative hybrid approaches for assurances: model checking and testing.

Model checking is an approach allowing designers to check that a property (typically expressed in some logic) holds for all reachable states in a system. Model checking is typically run offline and can only work in practice on either a high level abstraction of an adaptive system or on one of its components, provided it is simple enough. For example, in [48], the authors model MAPE loops of a mobile learning application in timed automata and verify robustness requirements that are expressed in timed computation tree logic using the Uppaal tool. In order to deal with uncertainty at design time and the practical coverage limitations of model checking for realistic system, several researchers have studied the application of model checking techniques at runtime. For example, in [8] QoS requirements of service-based systems are expressed as probabilistic temporal logic formulae, which are then automatically analyzed at runtime to identify and enforce optimal system configurations. The approach in [52] applies model checking techniques at runtime with the aim of validating compliance with requirements (expressed in LTL) of evolving code artifacts.

Testing allows designers to check that the system performs as expected for a finite number of situations. Inherently testing cannot guarantee that a self-adaptive system fully complies with its requirements. Testing is traditionally performed before deployment. As an example, in [49] the authors introduce a set of robustness tests, which provide mutated inputs to the interfaces between the controller and the target system, i.e. probes. The set of robustness tests are automatically generated by applying a set of predefined mutation rules to the messages sent by probes. Traditional testing allows demonstrating that a self-adaptive system is capable of satisfying particular requirements before deployment. However, it cannot deal with unanticipated system and environmental conditions. Recently, researchers from the area of self-adaptive systems have started investigating how testing can be applied at runtime to deal with this challenge. In [50], the authors motivate the need and identify challenges for the testing of self-adaptive systems at runtime. The paper introduces the so-called MAPE-T feedback loop for supplementing testing strategies with runtime capabilities based on the monitoring, analysis, planning, and execution architecture for self-adaptive systems.

1.4. Mechanisms to make approaches for perpetual assurances working

Turning the approaches of perpetual assurances into a working reality requires tackling several issues. The key challenge we face is to align the approaches with the requirements we have discussed in Section 1.2. For the functional requirements (R1 to R7), the central problem is how to collect assurance evidence arguments at runtime and compose them with the evidence acquired throughout the lifetime of the system possibly by different approaches. For the quality requirements (R8 to R10) the central problem is to make the solutions efficient and support the interchange of evidence arguments between the system and users. In this section, we first elaborate on the implications of quality requirements on approaches for perpetual assurances. Then we discuss two classes of mechanisms that can be used to provide the required functionalities for perpetual assurances and meet the required qualities: decomposition mechanisms and model-driven mechanisms.

1.4.1 Quality properties for perpetual assurances approaches

The *efficiency* of the approaches for perpetual assurances must be evaluated with respect to the *size* of the self-adaptive system (e.g., number of components in the system) and the *dynamism* it is subject to (e.g., the frequency of *change events* that the system has to face). An approach for perpetual assurances of self-adaptive systems is efficient if, for systems of realistic size and dynamism, it is able to:

- Provide results (assurances) within defined time constraints (depending on the context of use);
- Consume a limited amount of resources, so that the overall amount of consumed resources over time (e.g. memory size, CPU, network bandwidth, energy, etc.) remains within a limited fraction of the overall resources used by the system.
- Scale well with respect to potential increases in size of the system and the dynamism it is exposed to.

In the next sections, we discuss some promising mechanisms to make approaches for perpetual assurances working. Before doing so, it is important to note that orthogonal to the requirements for perceptual assurance in general, the level of assurance that is needed depends on the requirements of the self-adaptive system under consideration. In some cases, combining regular testing with simple and time-effective runtime techniques, such as sanity checks and contract checking, will be sufficient. In other cases, more powerful approaches are required. For example, model checking could be used to verify a safe envelope of possible trajectories of an adaptive system at design time, and verification at runtime to check whether the next change of state of the system keeps it inside the pre-validated envelope.

1.4.2 Decomposition mechanisms for perpetual assurances approaches

The first class of promising mechanisms for the perpetual provisioning of assurances is based on the principle of *decomposition*, which can be carried out along two dimensions:

1) *Time* decomposition, in which:
   a) Some preliminary/intermediate work is performed off-line, and the actual assurance is provided on-line, building on these intermediate results;
   b) Assurances are provided with some degree of approximation/coarseness, and can be refined if necessary.
2) *Space* decomposition, where verification overhead is reduced by independently verifying each individual component of a large system, and then deriving global system properties through verifying a composition of its component-level properties. Possible approaches that can be used to this end are:
   a) Flat approaches, that only exploit the system decomposition into components;
   b) Hierarchical approaches, where the hierarchical structure of the system is exploited;
   c) Incremental approaches, targeted at frequently changing systems, in which re-verification is carried out only on the minimal subset of components affected by a change.

We provide examples of state of the art approaches for each type.

1.a. Time decomposition: partially offline, assurances provided online. In [7] and [18], the authors propose approaches based on the pre-computation at design time of a formal and parameterized model of the system and its requirements; the model parameters are then instantiated at runtime, based on the output produced by a monitoring activity. Parametric probabilistic model checking [19] enables the evaluation of temporal properties on Parametric Markov Chains, yielding polynomials or rational functions as evaluation results. Instantiating the parameters in the result function, results for quantitative properties, such as probabilities or rewards can be cheaply obtained at runtime, based on a more computationally costly analysis process carried out offline. Due to limitations in practice of the parametric engines implemented in tools such as PARAM [19] or PRISM [20], the applicability of this technique may provide better results in self-adaptive systems with static architectures (or with a limited number of possible configurations), since dynamic changes to the architecture might require the generation of parametric expressions for new configurations.

1.b. Time decomposition: assurances with some degree of approximation. Statistical model checking [21][22][23] is a kind of simulation approach that enables the evaluation of probability and reward-based quantitative properties, similarly to probabilistic model checking. Specifically, the approach involves simulating the system for finitely many executions and using statistical hypothesis testing to determine whether the samples constitute statistical evidence of the satisfaction of a probabilistic temporal logic specification, without requiring the construction of an explicit representation of the state space in memory. Moreover, this technique provides an interesting tradeoff between analysis time and the accuracy of the results obtained (e.g., by controlling the number of sample system executions generated)[24]. This makes it appropriate for systems with strict time constraints in which accuracy is not a primary concern. Moreover, dynamic changes to the system architecture do not penalize resource consumption (e.g., no model reconstruction is required), making statistical model checking a good candidate for the provision of assurances in highly dynamic systems.

2.a. Space decomposition: flat approaches. In [28], the authors use assume-guarantee model checking to verify component-based systems one component at a time, and employs regular safety properties both as the assumptions made about system components and the guarantees they provide to reason about probabilistic systems in a scalable manner. [31] employs probabilistic interface automata to enable the isolated analysis of probabilistic properties in environment models (based on shared system/environment actions), which are preserved in the composition of the environment and a non-probabilistic model of the system.

2.b. Space decomposition: hierarchical approaches. The authors of [25] and [26] consider the decomposition of models to carry out incremental quantitative verification. This strategy enables the reuse of results from previous verification runs to improve efficiency. While [25] considers decomposition of Markov Decision Process-based Models into their strongly connected components (SCCs), [26] employs high-level algebraic representations of component-based systems to identify and execute the minimal set of component-wise re-verification steps after a system change.

2.c. Space decomposition: incremental approaches. In [26][27], the authors use assume-guarantee compositional verification to efficiently re-verify safety properties after changes that match some particular patterns, such as component failures.

Discussion. The discussed decomposition mechanisms lend themselves to the identification of possible division of responsibilities between human-driven and system-driven activities, towards the ultimate goal of providing guarantees that the system goals are satisfied. As an example, considering time decomposition approaches based on parametric model checking, the adopted parametric models can result from a combination of human-driven activities and artifacts automatically produced from other sources or information (architectural models, adaptation strategy specifications, etc.). Similarly, in case of space decomposition approaches based on assume-guarantee compositional verification, the assumptions used in the verification process could be automatically "learned" by the system [32], or provided by (or combined with input from) human experts.

1.4.3 Model-based mechanisms for perpetual assurances approaches

For whatever division of responsibilities between human and systems in the perpetual assurance process, an important issue is how to define in a clean, well-documented and traceable way the interplay between the actors involved in the process. Model-driven mechanisms can be useful to this end, as they can support the rigorous development of a self-adaptive system from its high-level design up to its running implementation. Moreover, they can support the controlled and traceable modification by humans of parts of the system and/or of its self-adaptive logic, e.g. to respond to modifications of the requirements to be fulfilled. In this direction [30] presents a model-driven approach to the development of adaptation engines. This contribution includes the definition of a domain-specific language for the modeling of adaptation engines, and a corresponding runtime interpreter that drives the adaptation engine operations. Moreover, the approach supports the combination of on-line machine controlled adaptations and off-line long-term adaptations performed by humans to maintain and evolve the system.

Steps towards the definition of mechanisms that can support the human-system interplay in the perpetual assurance process can also be found in [9]. The authors propose an approach called ActivFORMS in which a formal model of the adaptation engine (MAPE-K feedback loop) based on timed automata and adaptation requirements expressed in timed computation tree logic are complemented by a suitable virtual machine that can execute the models, hence guaranteeing at runtime the compliance of properties verified offline. The approach allows dynamically checking the adaptation requirements, complementing the evidence arguments derived from human-driven offline activities with arguments provided online by the system that could not be obtained offline (e.g., for performance reasons). The approach supports deployment of new models at runtime elaborated by human experts to deal with new goals.

1.5. Benchmark criteria for perpetual assurances techniques

This section provides benchmark criteria to compare different approaches that provide perpetual assurances. We identify benchmark criteria along four aspects: capabilities of approaches to provide assurances, basis of evidence for assurances, stringency of assurances, and performance of approaches to provide assurances. The criteria cover both functional and quality requirements for perpetual assurances techniques.

Capabilities of approaches to provide assurances. The first benchmark aspect compares the extent to which approaches differ in their ability to provide assurances for the requirements (goals) of self-adaptation. We distinguish the following benchmark criteria: variability (in requirements and the system itself), inaccuracy & incompleteness, conflicting criteria, user interaction, and handling alternatives.

Variability. Support for variability of requirements is necessary in ever-changing environments where requirements are not statically fixed, but are runtime entities that should be accessible and modifiable. An effective assurance approach should take into account requirements variations, including addition, updating, and deletion of requirements. For instance, variations to service-level agreement (SLA) contract specifications may have to be supported by an assurance approach. Variations may also involve the system itself, in term of unmanaged modifications; that is, an assurance approach should support system variability in terms of adding, deleting, and updating managed system components and services. This process may be completely automatic or may involve humans.

Inaccuracy & incompleteness. The second benchmark criterion deals with the capability of an assurance approach to provide evidence with models that are not 100% precise and complete. Inaccuracy may refer to both system and context models. For instance, it may not be possible to define requirements models at design time, thus making it necessary to leave a level of uncertainty within the requirement. On the other hand context models representing variations to context elements may be inaccurate, which may affect the results of an assurance technique. Similarly, incompleteness may either concern system or context models. For instance, either new services or new context elements may appear only at runtime. An assurance technique should provide the best possible assurance level regarding inaccurate and incomplete models. Additionally, an assurance technique may support re-evaluation at runtime when the uncertainty related to inaccuracy and incompleteness is solved (at least partially).

Conflicting criteria. Assurance approaches should take into account possible conflicting criteria. While it is desirable to optimize the utility of evidence (coverage, quality) provided by an assurance approach, it is important that the approach allows making a tradeoff between the utility of the evidence it provides and the costs for it (time, resource consumption).

User interaction. Assurance is related also to the way users interact with the system, in particular with respect to changing request load and changing user profiles/preferences. For instance, when the request load for a service increases, the assured level of availability of that service may be reduced. Self-adaptation may reconfigure the internal server structure (i.e., adding new components) to maintain the required level of assurance. In the case of changing user preference/profile variations, the level of assurances may either have to be recomputed or they may be interpreted differently. An assurance technique should be able to support such variations in the ways the system is used.

Handling alternatives is another important benchmark criterion for assurance of self-adaptive systems. An assurance technique should be able to deal with a flat space of reconfiguration strategies but also with one obtained by composition. In this context an important capability is to support assurances in the case one adaptation strategy is pre-empted by another one.

Basis of Assurance Benchmarking. The second aspect of benchmarking techniques for perpetual assurances defines the basis of assurance, i.e., the reasons why the researchers believe that the technique makes valid decisions. We consider techniques based on historical data only, projections in the future only, and combined approaches. In addition, we consider human-based evidence as a basis for assurance.

Historical data only. At one extreme, an adaptation decision can be solely based on the past historical data. An issue with such adaptive systems is that there are may be no guarantees that the analyses based on historical data will be reified and manifested in the future executions of system. Nevertheless, in settings where the past is a good indicator of the system's future behavior, such approaches could be applicable.

Projections into the future only. At the other extreme, a system may provide assurances that project into the future, meaning that the analysis is not based on the historical data, but based on predictive models of what may occur in the future. The predictions may take different forms, including probabilistic models and fuzzy methods to address the uncertainty with which different situations may occur. A challenge with this approach is the difficulty of constructing predictive models.

Combined approaches. Finally, in many adaptive systems, combinations of the aforementioned techniques are used, i.e., some past historical data is used together with what-if analysis of the implications of adaptations over the predicted behaviors of the system in its future operation.

Human-based evidence. Since users may be involved in the assurance process, we consider human-based evidence as a further basis for assurance. Input from users may be required before performing adaptations, which may be critical in terms of provided services. Consider a situation in which a self-adaptive service-oriented system needs to be reconfigured by replacing a faulty service. Before substituting this service with a new one, selected by an automated assurance technique, the user may want to check its SLA to confirm its suitability as a replacement according to a criterion that cannot be easily applied automatically. Such a criterion may be subjective, not lending itself to quantification. Thus, by means of accepting the SLA of the new service, the user provides a human-based evidence to complement the assurances provisioned through an automated technique, which can be based on past historical data, predictive models or a combination of both.

Discussion. When publishing research results, it is important to explicitly discuss and benchmark the basis of assurances. For instance, if historical data is used for making decisions, the research needs to assess whether past behavior is a good indicator of future behavior, and if so, quantify such expectations. Similarly, if predictive models are used for making decisions, the researchers need to assess the accuracy of the predictions, ideally in a quantitative manner.

Stringency of assurances. A third important benchmarking aspect of techniques for perpetual assurances of self-adaptive systems is the nature of rationale and evidence of the assurances, which we term *stringency of assurances*. The rationale for assurances may differ greatly depending on the purpose of the adaptive system and its users. While for one system the only proper rationale is a formal proof, for another system, proper rationale may be simply statistical data, such as the probability with which the adaptation has achieved its objective in similar prior situations. For example, a safety-critical system may require a more stringent assurance rationale than a typical consumer software system. Note that the assurance rationale is a criterion that applies for different phases of adaptation (monitoring, analysis, planning, and execution). Different phases of the same adaptive system may require different level of assurance rationale. For instance, while a formal proof is a plausible rationale for the adaptation decisions made during the planning phase, it may not be appropriate for the monitoring phase. Similarly, confidence intervals for the distribution of monitored system parameters may be a plausible rationale for the monitoring phase, but not necessarily for the execution phase of a self-adaptive software system.

Performance of approaches to provide assurances. Finally, for an assurance approach to be applicable in real-world scenarios it is essential to evaluate its performance, which is the fourth and final proposed benchmark aspect. We consider a set of benchmarks, which directly follow from *efficiency* and *scalability* concerns discussed in Section 1.4, namely timeliness, computational overhead, and complexity analysis.

Timeliness. The time required to achieve the required evidence is a key performance benchmark criteria, which is obvious for techniques that are used at runtime.

Computational overhead. Related to timeliness is the computational overhead, i.e., the consumed resources (e.g., memory and CPU) for enacting the assurance approach.

Complexity analysis. As each assurance technique has an (implicit) complexity that may affect its applicability in terms of the analysis required for a problem at hand, we propose a complexity analysis benchmark. Assurance techniques may be benchmarked in terms of their scope of applicability across different types of problems.

The following table summarizes the benchmark aspects, and for each of them the different benchmark criteria.

Table 4 Summary benchmark aspects and criteria

Benchmark aspect	**Benchmark criteria**	
	Criteria	**Description**
	Variability	Capability of an approach to handle variations in requirements (adding, updating, deleting goals), and the system (adding, updating, deleting elements)
	Inaccuracy & incompleteness	Capability of an approach to handle inaccuracy and incompleteness of models of the system and context
Capabilities of approaches to provide assurances	Conflicting criteria	Capability of an approach to balance the tradeoffs between utility (e.g., coverage, quality) and cost (e.g., time, resources)
	User interaction	Capability of an approach to handle changes in user behavior (preferences, profile)
	Handling alternatives	Capability of an approach to handle changes in adaptation strategies (e.g., pre-emption)
Basis of assurance benchmarking	Historical data only	Capability of an approach to provide evidence over time based on historical data
	Projections in the future	Capability of an approach to provide evidence based on predictive models
	Combined approaches	Capability of an approach to provide evidence based on combining historical data with predictive models
	Human evidence	Capability of an approach to complement automatically gathered evidence by evidence provided by humans
Stringency of assurances	Assurance rational	Capability of the approach to provide the required rational of evidence for the purpose of the system and its users (e.g., completeness, precision)
Performance of approaches	Timeliness	The time an approach requires to achieve the required evidence.
	Computational overhead	The resources required by an approach (e.g., memory and CPU) for enacting the assurance approach.
	Complexity	The scope of applicability of an approach to different types of problems.

Some of the benchmark criteria from Table 4 directly reflect a specific requirement from Table 2. For example, the 'Timeliness' criterion is directly linked to requirement R8 ('Produce timely updates'). Other criteria relate to multiple requirements from Table 2. As an example, the 'Human evidence' criterion links to R5 ('Integrate new evidence'), R7 ('Provide evidence for human activities that realize R5'), and R10 ('Auditable arguments'). Finally, some arguments only link indirectly to the requirements from Table 2. This is the case for the `Handling alternatives' criterion, which relates to the solution for self-adaptation and this solution may provide different levels of support for the requirements of perpetual assurances.

2. Case study

We now present a case study that allows evaluating, comparing, and ranking approaches for perpetual assurances. The case study has been conceived to challenge the capability of approaches for self-adaptation to achieve *perpetual provisioning of assurances* for different requirements driven by the benchmark criteria discussed in the previous section. The particular ways in which different approaches solve the challenges proposed by the case study offers two distinct advantages. First it allows comparing prototypal research efforts, and second it acts as a testbed to demonstrate the effectiveness of the different techniques adopted by the self-adaptive solutions.

We start by briefly motivating the domain of the case study. We then introduce a set of general adaptation scenarios, linking them to the types of uncertainty occurring in self-adaptive systems and to the requirements for perpetual assurances, and proposing general metrics for benchmarking. Next we describe the concrete case study. Finally, we give concrete instances of the generic adaptation scenarios and explain benchmark criteria.

2.1. Domain and general adaptation scenarios

The case study comes from the domain of *service-based systems*, in which software services offered by third-party providers are dynamically composed at run time to deliver complex functionality. Service-based systems are widely used in e-commerce, online banking, e-health and many other types of applications. They increasingly rely on self-adaptation to cope with the uncertainties that are often associated with third-party services [8][18][41][44], as the loose coupling of service-oriented architectures makes online reconfiguration feasible. Hence, the case study is a prototypical application.

Figure 1 shows an overview of the architecture of a service-based system.

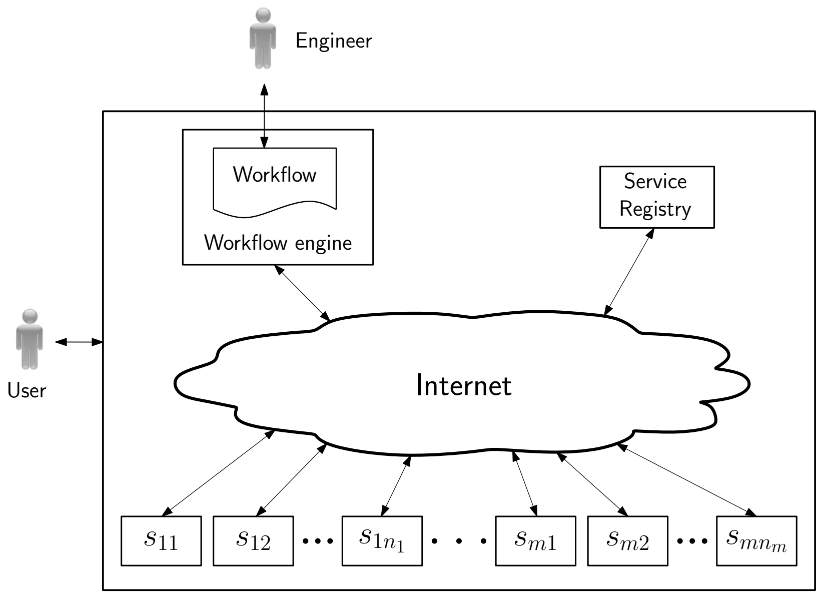

Figure 1. Architecture of a service-based system

A typical service-based system consists of a composition of web services that are accessed remotely through a software application called workflow engine. Several providers may offer services that provide the same functionality, often with different levels of performance, reliability, costs, etc. To capture the different quality characteristics, the diagram shows $m \geq 1$ sets of concrete services: the set $S_i = \{s_{i1}, s_{i2}, ..s_{ini}\}$, $1 \leq i \leq m$, is comprised of $n_i \geq 1$ concrete services that provide the same abstract service from a functional point of view. Service providers can register concrete services at the service registry. The workflow of a composite service finds addresses (end points) of concrete services via the service registry. The way in which the workflow engine employs concrete services in order to provide the functionality required by the user is specified in the workflow that the engine is executing.

Table 5 shows a list of generic adaptation scenarios for service-based systems. These scenarios provide different and increasing challenges to self-adaptation in general and the provision of perpetual assurances in particular. They are not meant to be exhaustive, but cover typical types of adaptation problems with different types of uncertainties.

Table 5 Generic adaptation scenarios for TAS

Scenario	Type(s) of uncertainty	Type(s) of requirements	Observable properties	Types of adaptations
S1	Individual service failure	Reliability	Success or failure of each service invocation (Boolean – true=success, false=failure)	E.g, select equivalent service, invoke idempotent services in parallel, enact alternative service
S2	Variation of response time of a service over time	Performance	Response time for each service invocation (ms)	E.g, select equivalent service, invoke idempotent services in parallel, enact alternative service
S3	New alternative service becomes available	Reliability, performance, cost	Available concrete services for abstract service (Set of currently available services)	E.g, select new concrete service, enact new service
S4	New type of service becomes available, requirement for new functionality	Add new service	Request to add new service (String), available concrete services including services for the new functionality (Set of currently available services)	E.g, adapt workflow, enact adapted workflow, select new concrete service, enact new service

Adaptation scenarios differ based on the following characteristics:

1. **Types of uncertainties that the system needs to handle:** different types of uncertainty present different challenges to adaptation solutions and to the approaches uses for perpetual assurances; for example monitoring or quantifying uncertainty (R1 and R2 in Table 2) handling the failures of an individual service may be less challenging than integrating a new type of service that becomes available.
2. **Types of requirements that the self-adaptive system must meet:** self-adaptation can be used to achieve different types of system requirements.[2] Scenarios with a single requirement are typically less challenging for self-adaptation solutions and assurance approaches than scenarios where multiple conflicting requirements need to be balanced. Different combinations of requirements pose different challenges to perpetual assurances; e.g., monitoring, quantifying, and integrating new evidence (R1, R2, and R5) may be more demanding for requirements that require real-time tracking, while managing overlapping uncertainty resources (R3) and combining new evidence (R6) may be particularly challenging for collecting evidence of interdependent requirements.
3. **Observable attributes/properties:** the observations of the system and its execution context (R1 to R7) that are required depend on scenario, and introduce different challenges for adaptation solutions and approaches for perpetual assurances. For example monitoring the failures of a service is less demanding than observing the emergence of a new type of service (which may not have been anticipated upfront) and handling the request for integrating it into the system.
4. **Types of adaptations that are possible:** different scenarios require different types of adaptations. As an example, handling service failures may involve trying equivalent services until an invocation completes successfully, while introducing a new type of service requires an adaptation of the service workflow. On the other hand, approaches may be available that offer different adaptation strategies; e.g., an alternative strategy for reducing the likelihood of an operation failing may be executing multiple idempotent services in parallel (and using the result returned by the successful execution that completes successfully first).

[2] It is important to distinguish the requirements that the self-adaptive system needs to realize (reliability, performance, etc.) and the requirements for approaches of perpetual assurances (summarized in Table 2)

Different approaches to providing perpetual assurance should be evaluated and compared based on:

1. The benchmark aspects and criteria from Table 4.
2. Their ability to handle the scenarios in Table 5 (or a subset of them).
3. The quality of the adaptation for which they provide assurance, which can be evaluated as described in the existing literature, e.g., [53].

2.2. Tele Assistance System

We now present the case study. The concrete application is a Tele Assistance System (TAS) that offers health support to patients using home devices. The case study is based on the example introduced in [44] and later used in [8][36]. The TAS is a composite service that uses the following services:

- Alarm Service, which provides the operation sendAlarm
- Medical Analysis Service, which provides the operation analyzeData
- Drug Service, which provides the operations changeDoses and changeDrug

Multiple providers offer concrete services for the TAS with different characteristics, e.g. for reliability and cost. The TAS workflow (Figure 2) will use a particular strategy to select concrete services, for example, based on the minimum response time.

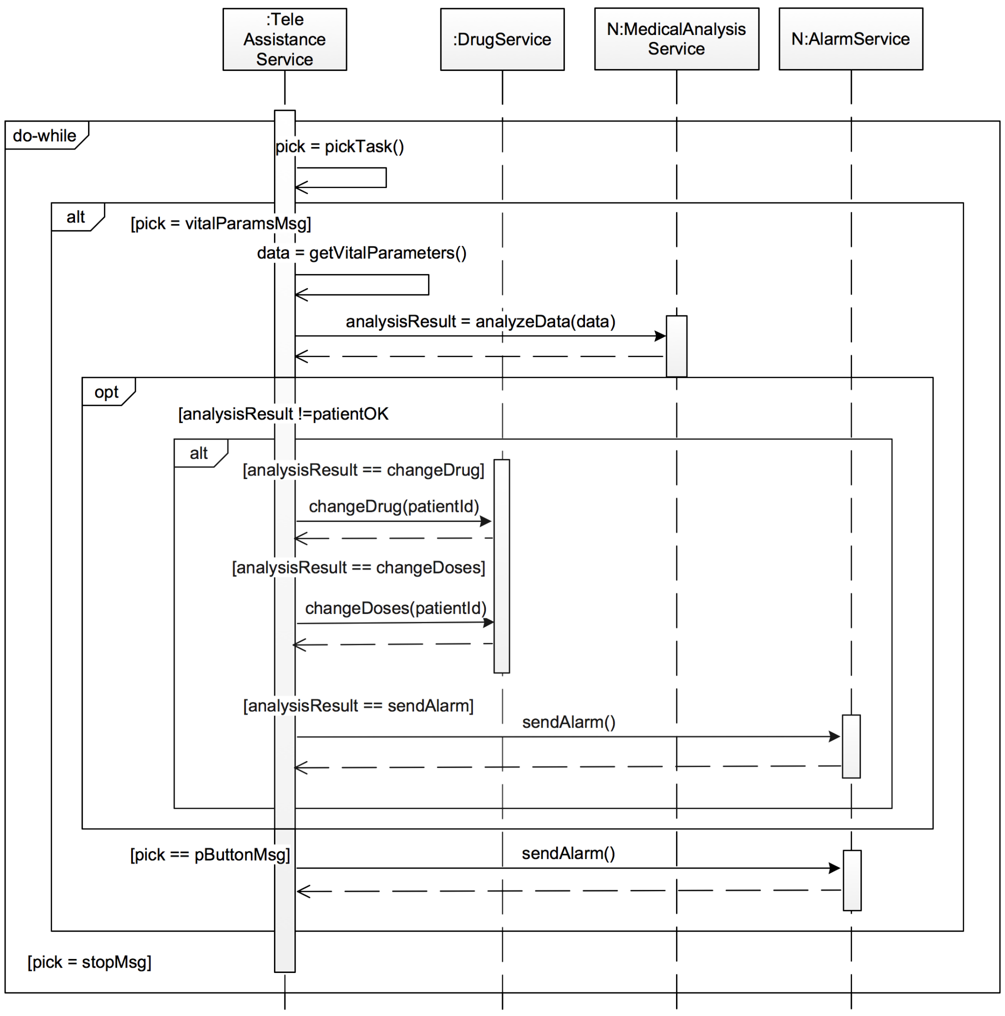

Figure 2. Workflow of the TAS

2.2. Adaptation scenarios and benchmark criteria

We now provide instances of the generic adaptation scenarios.

Setting: The TAS can use one of several concrete services for each abstract service it requires. Concrete services differ in failure rate, response time, and cost. To select concrete services the workflow uses a service registry with service descriptions that is refreshed from time to time. A service description lists the maximum failure rate and maximum response time promised by the provider of the concrete service, and the cost per service invocation.

S1 - Individual service failure

Without adaptation: Concrete service i that provides the sendAlarm operation is selected such that:

(i) $fr_i = \min_{1 \leq j \leq n}(fr_j)$, where fr_x is the promised maximum failure rate of the x-th concrete service Sx from the set of concrete alarm services,

(ii) $cost_i$ is the minimum cost of the concrete Alarm Service options that satisfy (i).

Uncertainty: The concrete services that provide the Alarm Service fail from time to time (we assume that there are at least two concrete alarm services).

Adaptation requirement: The failure rate of workflow executions that consist of an invocation of the Alarm Service does not exceed X.

With adaptation: The system observes each successful and failed invocation of an Alarm Service and dynamically selects the lowest-cost concrete service that meets the requirement.

S2 - Variation of failure rate of services over time

Without adaptation: A pair of concrete services that provide the Medical Analysis Service and Alarm Service is selected such that:

(i) $fr_1 + fr_2 <= X$, where fr_1 and fr_2 are the promised maximum failure rates of the two selected services, and X is a pre-specified maximum failure rate of workflow executions comprising an invocation of the Medical Analysis Service followed by an invocation of the Alarm Service.

(ii) cost_1 + cost_2 is minimal cost across all combinations of concrete Medical Analysis Service and Alarm Service options that satisfy (i).

Uncertainty: The failure rates of the concrete services that provide the Alarm Service change significantly over time.

Adaptation requirement: The system requirement of a maximum failure rate of workflow executions comprising an invocation of the Medical Analysis Service followed by an invocation of the Alarm Service lower than or equal to X with minimal cost of service selections is maintained, regardless of the changing failure rates of concrete services over time.

With adaptation: The system observes each successful and failed invocation of a Medical Analysis Service and Alarm Service and dynamically selects the lowest-cost pair of concrete services that meets the requirement.

S3 - New service becomes available

Without adaptation: A concrete service that provides the Alarm Service is selected as in Scenario S1. The registry with service descriptions is periodically refreshed with a period T.

Uncertainty: A new concrete service that provides the Alarm Service becomes available at some time t.

Adaptation requirement: The failure rate of workflow executions that consist of an invocation of the Alarm Service does not exceed X.

With adaptation: The system observes each successful and failed invocation of an Alarm Service and dynamically selects the lowest-cost concrete service from the set of concrete alarm services, including, after a time $\Delta t << T$, the new concrete service.

S4 - New type of service becomes available

Without adaptation: A concrete service that provides the Alarm Service is selected as in Scenario S1.

Uncertainty: a new type of service called Inform Relatives becomes available at time t; the Inform Relatives service informs relatives of the patient in case of invocations of the Alarm Service; concrete services of the Inform Relatives service are characterized by a cost.

Adaptation requirement: After a time Δt a concrete service that provides the Inform Relatives service is selected; the lowest cost concrete Inform Relatives service is invoked after every selection of an Alarm Service.

With adaptation: The system discovers new Inform Relatives services, it integrates the corresponding abstract service in the workflow, caches the concrete instances of the new service, and dynamically selects the lowest-cost concrete service from the set of Inform Relatives services.

For each of these scenarios, different benchmark criteria can be applied. For example, for Scenario S2, we may apply the benchmark *variability* (the capability of an approach to handle variations in requirements) by comparing the percentage of time solutions assure compliance of the requirement that the maximal failure rate of workflow executions is not violated when different distributions of changing failure rates of services are imposed. In Scenario S4, we may benchmark *incompleteness* by comparing the capability of an approach to provide partial assurances that a new service for contacting relatives is correctly integrated in the workflow. We may apply benchmark *performance of approaches* for different scenarios by comparing the cost of different approaches to assure compliance with the requirement at runtime, in terms of overhead in time, CPU, and memory.

3. Conclusions

Providing assurances for self-adaptive systems is hard. The primary underlying problem is uncertainty that may stem from a variety of different sources, ranging from incomplete knowledge to noise in sensing and uncertain behavior of humans in the loop. Providing assurances for the ability of the self-adaptive system to achieve its requirements calls for an enduring process spanning the whole life time of the system. In this process, humans and the system jointly derive and integrate new evidence and arguments, which we coined perpetual assurances for self-adaptive systems. However, realizing perpetual assurances poses severe challenges. We summarize the key challenges below.

First, we need a better understanding of the nature of uncertainty for software systems. This paper provides an initial classification of sources of uncertainties. However, more research is required to obtain a comprehensive view on sources of uncertainty and their specific characteristics.

Second, we need a deeper understanding of how we can handle uncertainty in self-adaptive systems, and in particular, how can we monitor and quantify uncertainty. Currently, there is a growing understanding of how to handle uncertainty regarding parameters of the system, its goals, and the environment. However, how to handle uncertainty regarding parts of the system and the environment that may not be completely know upfront, or handling uncertainty regarding new goals remains to a large extent an open problem.

Third, deriving, integrating and combining new evidence pose additional hard challenges. A variety of techniques for obtaining evidence for requirements compliance of software systems exist. However, most of these techniques are conceived for offline use. Perpetual assurance requires continuously deriving, integrating and combining new evidence, while the system is operating. We have presented decomposition and model-based mechanisms as potentially interesting paths to go forward. However, making these mechanisms effective is particularly challenging and requires a radical revision of many of the existing techniques.

Fourth, to drive research on assurances for self-adaptive systems forward, we need good exemplars. Exemplars enable comparison of different solution, pinpoint the critical challenges, and demonstrate the effectiveness of the different mechanisms adopted by the self-adaptive solutions. The case study in this paper aims to contribute with such an exemplar.

Bibliography

[1] Inverardi, P.: Software of the Future Is the Future of Software? Trustworthy Global Computing. Lecture Notes in Computer Science, vol. 4661, Springer 2007

[2] Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and runtime. FSE/SDP workshop on Future of software engineering research, ACM 2010

[3] Andersson, L. Baresi, N. Bencomo, R. de Lemos, A. Gorla, P. Inverardi and T. Vogel, Software Engineering Processes for Self-Adaptive Systems, Software Engineering for Self- Adaptive Systems II Lecture Notes in Computer Science vol. 7475, Springer 2013

[4] Cheng B. et al. Software engineering for self-adaptive systems: A research roadmap. In Software Engineering for Self-Adaptive Systems, Lecture Notes in Computer Science vol. 5525. 2009.

[5] de Lemos R. et al. Software engineering for self-adaptive systems: A second research roadmap. In Software Engineering for Self-Adaptive Systems II, Lecture Notes in Computer Science vol. 7475, Springer, 2013.

[6] Zhang J. and Cheng B.: Model-based development of dynamically adaptive software. In 28th International Conference on Software Engineering, ICSE 2006.

[7] Filieri A., Ghezzi C., and Tamburrelli G.: Runtime efficient probabilistic model checking. International Conference on Software Engineering, ICSE 2011

[8] Calinescu R., Grunske L., Kwiatkowska M., Mirandola R., and Tamburrelli G.: Dynamic QoS management and optimization in service-based systems. IEEE Transactions on Software Engineering, 37(3):387–409, May 2011

[9] Iftikhar U. and Weyns D., ActivFORMS: Active Formal Models for Self-Adaptation, Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014

[10] Funtowicz S., and Ravetz J.: Uncertainty and Quality in Science for Policy. Springer, 1990.

[11] Walker W., Harremos P., Romans J., van der Sluus J., van Asselt M., Janssen P., and Krauss M.: Defining uncertainty. a conceptual basis for uncertainty management in model-based decision support. Integrated Assessment, 4(1):5–17, 2003.

[12] Perez-Palacin and Mirandola R.. Uncertainties in the modeling of self-adaptive systems: A taxonomy and an example of availability evaluation. In Proc. of the 5th ACM/SPEC Int. Conf. on Performance Engineering, New York, NY, USA, 2014. ACM.

[13] Refsgaard J.C., van der Sluijs J.P., Højberg A.L., and Vanrolleghem P.A.: Uncertainty in the environmental modeling process - a framework and guidance. Environ. Model. Softw., 22(11):1543–1556, Nov. 2007.

[14] Garlan D.: Software engineering in an uncertain world. In Future of Software Engineering Research workshop, FoSER, New York, NY, USA, 2010

[15] Esfahani N. and S. Malek S.: Uncertainty in self-adaptive software systems. In Software Engineering for Self-Adaptive Systems II, volume 7475 of LNCS, pages 214–238. Springer Berlin Heidelberg, 2013.

[16] Filieri A., Ghezzi C., and Tamburrelli G.: A formal approach to adaptive software: continuous assurance of non-functional requirements. Formal Asp. Comput. 24(2): 163-186, 2012.

[17] Ghezzi, C., et al. On requirements verification for model refinements. Requirements Engineering Conference (RE), 2013 21st IEEE International. IEEE, 2013.

[18] Cardellini V., Casalicchio E., Grassi V., Iannucci S., Lo Presti F., and Mirandola R.: MOSES: A Framework for QoS Driven Runtime Adaptation of Service-Oriented Systems. IEEE Trans. Softw. Eng. 38, 5, 1138-1159, 2012.

[19] Hahn, E. M.; Hermanns, H.; Wachter, B. and Zhang, L. PARAM: A Model Checker for Parametric Markov Models. In CAV, pages 660-664, 2010.

[20] Kwiatkowska M., Norman G. and Parker D.: PRISM 4.0: Verification of Probabilistic Real-time Systems. In Proc. 23rd International Conference on Computer Aided Verification (CAV'11), volume 6806 of LNCS, pages 585-591, Springer, 2011.

[21] Legay A., Delahaye B., Bensalem S.: Statistical Model Checking: An Overview. In Proceedings of the First International Conference on Runtime Verification (RV 2010), volume 6418 of LNCS, pages 122-135 , Springer, 2010.

[22] Younes H. L. S. and Simmons R. G.: Statistical probabilistic model checking with a focus on time-bounded properties. Information and Computation, 204(9):1368-1409, 2006

[23] Grunske L., Zhang P.: Monitoring probabilistic properties. In Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE 2009

[24] Younes H.L.S., Kwiatkowska M.Z., Norman G., and Parker D.: Numerical vs. statistical probabilistic model checking. STTT, 8(3):216-228, 2006.

[25] Kwiatkowska, M., Parker, D., Qu, H.: Incremental quantitative verification for Markov decision processes. In: Proceedings 2011 IEEE/IFIP International Conference Dependable Systems and Networks, 2011.

[26] Johnson K., Calinescu R., and Kikuchi S.: An incremental verification framework for component-based software systems. In CBSE'13. Pages 33-42, 2013.

[27] Calinescu R., Kikuchi S., and Johnson K.: Compositional reverification of probabilistic safety properties for large-scale complex IT systems. In Large-Scale Complex IT Systems, volume 7539 of LNCS, pages 303-329. Springer, 2012.

[28] Kwiatkowska M., Norman G., Parker D., and Qu H.: Assume-guarantee verification for probabilistic systems. In TACAS'10, volume 6105 of LNCS, pages 23-37, 2010.

[29] Weyns D., Iftikhar M. U., de la Iglesia D. G., and Ahmad T.: A survey of formal methods in self-adaptive systems. Fifth International C* Conference on Computer Science and Software Engineering, C3S2E 2012

[30] Vogel T. and Giese H.: Model-Driven Engineering of Self-Adaptive Software with EUREMA. ACM Trans. Auton. Adapt. Syst. 8, 4, Article 18 (January 2014)

[31] Pavese E., Braberman V., and Uchitel S.: Probabilistic environments in the quantitative analysis of (non-probabilistic) behaviour models. In Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE 2009

[32] Sykes D., Corapi D., Magee J., Kramer J., Russo A., and Inoue K.: Learning revised models for planning in adaptive systems. In Proceedings of the 2013 International Conference on Software Engineering, ICSE 2013

[33] Giese H., Bencomo N., Pasquale L., Ramirez A.J., Inverardi P., Watzoldt S., and Clarke S., Living with Uncertainty in the Age of Runtime Models, in Bencomo, France, Betty Cheng and Assmann (Editors), LNCS 8378, pp. 47-100, Springer 2014

[34] Ramirez A., Jensen A., and Cheng B.: A taxonomy of uncertainty for dynamically adaptive systems, Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2012

[35] Cheng B.H.C., Eder K.I., Gogolla M., Grunske L., Litoiu M, Müller H.A., Pelliccione P., Perini A., Qureshi N.A., Rumpe B., Schneider D., Trollmann F., Villegas N.M.: Using Models at Runtime to Address Assurance for Self-Adaptive Systems, LNCS 8378, pp. 47-100, Springer, 2014.

[36] Epifani I., Ghezzi C., Mirandola R., and Tamburrelli G.: Model evolution by runtime parameter adaptation. In Proceedings of the 31st International Conference on Software Engineering IEEE Computer Society, ICSE 2009

[37] Calinescu R., Johnson K., and Rafiq Y.: Using observation ageing to improve markovian model learning in QoS engineering. 2nd ACM/SPEC International Conference on Performance engineering ACM, New York, NY, USA, 2011.

[38] Epifani I., Ghezzi C., and Tamburrelli G.: Change-point detection for black-box services. In Proceedings of the eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2010

[39] Calinescu R., Rafiq Y., Johnson K., and Bakır M.E.: Adaptive model learning for continual verification of non-functional properties. In Proceedings of the 5th ACM/SPEC international conference on Performance engineering, ICPE 2014

[40] Calinescu R. and Kwiatkowska M.: Using quantitative analysis to implement autonomic IT systems. In Proceedings of the 31st International Conference on Software Engineering (ICSE '09). IEEE Computer Society, Washington, DC, USA, 100-110.

[41] Calinescu R., Johnson K., and Rafiq Y.: Developing self-verifying service-based systems, IEEE/ACM 28th International Conference on Automated Software Engineering, ASE 2013

[42] Calinescu R., Ghezzi C., Kwiatkowska M., and Mirandla R.: Self-adaptive software needs quantitative verification at runtime. Commun. ACM 55, 9 2012,

[43] Beugnard A., Jezequel J., Plouzeau N., and Watkins D. : Making components contract aware. In IEEE Computer 32 (7), 38-45

[44] Baresi L., Bianculli D., Ghezzi C., Guinea S., and Spoletini P.: Validation of Web Service Compositions, IET Software, vol. 1, no. 6, pp. 219-232, Dec. 2007.

[45] Zhang J. and Cheng B., Using temporal logic to specify adaptive program semantics, The Journal of Systems and Software 79,1361–1369, 2006

[46] Ye C., Cheung S.C. and Chan W.K. Process Evolution with Atomicity Consistency, Software Engineering of Adaptive and Self-Managing Systems, SEAMS 2007

[47] Uttamchandani S., Yin L., Alvarez G., Palmer J. and Agha G.: Chameleon: a self-evolving, fully-adaptive resource arbitrator for storage systems, USENIX Technical Conference 2005

[48] de la Iglesia D.G. and Weyns D.: Guaranteeing Robustness in a Mobile Learning Application Using Formally Verified MAPE Loops, Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2013

[49] Cámara J., de Lemos R., Laranjeiro N., Ventura R., and Vieira M.: Testing the robustness of controllers for self-adaptive systems, Journal of the Brazilian Computer Society, 20:1, 2014

[50] Fredericks E.M., Ramirez A.J., and Cheng B.H.C.: Towards runtime testing of dynamic adaptive systems. 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2013

[51] Stoller S., Bartocci E., Seyster J., Grosu R., Havelund K., Smolka S., Zadok E.: Runtime Verification with State Estimation, Runtime Verification, Lecture Notes in Computer Science Volume 7186, 2012

[52] Inverardi P. and Mori P. : Model checking requirements at runtime in adaptive systems, Workshop on Assurances for Self-Adaptive Systems, ASAS, 2011

[53] Villegas N., Müller H., Tamura G., Duchien L., and Casallas R.: A Framework for Evaluating Quality-Driven Self-Adaptive Software Systems. Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011.