Winter January, 2017

Synthesis and Verification of Self-aware Computing Systems

Radu Constantin Calinescu
Marco Autili
Javier Cámara
Antinisca Di Marco,
Simos Gerasimou, et al.

Synthesis and verification of self-aware systems

Radu Calinescu, Marco Autili, Javier Cámara, Antinisca Di Marco,
Simos Gerasimou, Paola Inverardi, Alexander Perucci, Nils Jansen,
Joost-Pieter Katoen, Marta Kwiatkowska, Ole J Mengshoel, Romina Spalazzese
and Massimo Tivoli

Abstract Self-aware computing systems are envisaged to exploit the knowledge of their own software architecture, hardware infrastructure and environment in order to follow high-level goals through proactively adapting as their environment evolves. This chapter describes two classes of key enabling techniques for self-adaptive systems: automated synthesis and formal verification. The ability to dynamically synthesise component connectors and compositions underpins the proactive adaptation of the architecture of self-aware systems. Deciding when adaptation is needed and selecting valid new architectures or parameters for self-aware systems often requires formal verification. We present the state of the art in the use of the two techniques for the development of self-aware computing systems, and summarise the main research challenges associated with their adoption in practice.

Radu Calinescu
University of York, UK, e-mail: radu.calinescu@york.ac.uk

Marco Autili
University of L'Aquila, Italy, e-mail: marco.autili@univaq.it

Javier Cámara
Carnegie Mellon University, US, e-mail: jcmoreno@cs.cmu.edu

Antinisca Di Marco
University of L'Aquila, Italy, e-mail: antinisca.dimarco@univaq.it

Simos Gerasimou
University of York, UK, e-mail: simos@cs.york.ac.uk

Paola Inverardi
University of L'Aquila, Italy, e-mail: paola.inverardi@univaq.it

Alexander Perucci
University of L'Aquila, Italy, e-mail: alexander.perucci@graduate.univaq.it

Nils Jansen
University of Texas at Austin, US, e-mail: njansen@utexas.edu

Joost-Pieter Katoen
RWTH Aachen University, Germany, e-mail: katoen@cs.rwth-aachen.de

Marta Kwiatkowska
University of Oxford, UK, e-mail: marta.kwiatkowska@cs.ox.ac.uk

Ole J Mengshoel
Carnegie Mellon University, US e-mail: ole.mengshoel@sv.cmu.edu

Romina Spalazzese
Malmö University, Sweden, e-mail: romina.spalazzese@mah.se

Massimo Tivoli
University of L'Aquila, Italy, e-mail: massimo.tivoli@univaq.it

1 Introduction

Self-aware computing systems address a rapidly growing need for automation in the management of large, complex computer applications. Made possible by recent advances in the numbers and types of sensors embedded in computing systems, and in the ability to dynamically change system parameters and architectures, self-aware systems are envisaged to be self-reflective, self-predictive and self-adaptive. As such, self-aware systems need to learn models that encode knowledge about themselves and their environment, and to reason using these models in ways that allow them to self-adjust in response to evolving environments and goals [46].

This chapter presents two classes of key enabling techniques for self-aware systems. First, Sections 2 and 3 describe techniques for the *automated synthesis* of self-aware service compositions and self-adaptive connectors, respectively. These synthesis techniques support the runtime modification of software architectures that underpins the adaptation of an increasing number of self-aware computing systems.

Second, Sections 4–6 present *formal verification* techniques that support model-based reasoning within self-aware systems. Quantitative verification at runtime, the technique summarised in Section 4, supports the runtime detection of quality-of-service (QoS) requirement violations in self-aware systems and the selection of new configurations that reinstate system compliance with these requirements. Section 5 introduces a technique called *parametric verification*, which is particularly suited for use in a runtime context, to establish the QoS properties of self-aware computing systems. Finally, Section 6 discusses *runtime verification* based on probabilistic graphical models such as Bayesian networks.

The two classes of techniques—verification and synthesis—are brought together in Section 7. This section describes how modelling a system and its environment as a stochastic multiplayer game supports the analysis of their interplay, and can be used to drive runtime strategy synthesis for self-aware computing systems.

2 From design-time to runtime synthesis of self-aware choreographies of software services

The automated synthesis of the distributed coordination logic that is required to compose software services achieves correctness by construction of the composed system with respect to specified business goals. State-of-the-art approaches (see [35] and references there in) are static and are poorly suited to the synthesis of service compositions that are able to dynamically evolve in response to changes, e.g., goal changes, QoS degradation and security policies changes. Focussing on goal changes, this section overviews a novel approach where, together with self-awareness, the integration of design-time and runtime synthesis enables the dynamic evolution of service compositions so as to adapt to possible goal changes. Automatic support is required at design time to synthesize the initial overall logic to exogenously coordinate (in a fully distributed way) the involved services. At runtime, automatic support is then required to achieve self-adaptation, through self-awareness, when re-synthesizing on the fly the portion of the coordination logic affected by the change. For a reasonably complete treatment of alternative approaches in the state of the art, we refer to related work described in [7, 5].

2.1 Setting the context

The Future Internet [30] promotes a distributed-computing environment that will be increasingly inhabited by a virtually infinite number of software services. Software systems will be more and more built by composing together software services distributed over the Internet.

Today's service composition mechanisms are based mostly on service orchestration, a centralized approach to the composition of multiple services into a larger application. Orchestration works well in static environments with predefined services and minimal environment changes. These assumptions are inadequate in the Future Internet vision, in which many diverse service providers and consumers keep changing and cannot be coordinated through a centralized approach. In contrast, service choreography is a form of decentralized composition that models the external interaction of the participant services by specifying peer-to-peer message exchanges from a global perspective [7, 5].

The need for service choreography was recognized in BPMN2 (Business Process Model and Notation Version 2.0[1]), the de facto standard for specifying choreographies, which introduced choreography-modeling constructs. BPMN2 *Choreography Diagrams* model peer-to-peer communication by defining a multiparty protocol that, when put in place by the cooperating parties, allows reaching the overall choreography objectives in a fully distributed way. In this sense, service choreographies differ significantly from service orchestrations, in which one stakeholder centrally determines how to reach an objective through cooperation with other services. Future software systems will not be realized by orchestration only; they will also require choreographies. Indeed, services will be increasingly active entities that, communicating peer-to-peer, proactively make decisions and autonomously perform tasks according to their own imminent needs and the emergent global collaboration.

2.2 The need for self-adaptation

When third-party participants are involved, usually black-box services to be reused, a key enabler for the actual realization of choreographies is the ability to automatically compose services, and to dynamically perform exogenous coordination of their interaction. However, in a distributed setting, obtaining the coordination logic required to realize a choreography is nontrivial and error prone. Accordingly, automatic support for realizing choreographies is needed.

[1] www.omg.org/spec/BPMN/2.0

Choreography-based software systems may be in operation for a long time, and it is impractical (if not infeasible) to replace or retry the whole choreography process whenever a change occurs. Instead, choreographies will continuously self-adapt to new and modified goals, e.g., to meet new business requirements, as well as changing execution environment, e.g., to support new technologies. Indeed, in general, many facets can be considered when dealing with software systems that are capable to evolve by adapting their behavior at runtime. The study presented in [2] identifies the multiple facets of (self-)adaptation and classifies them into four groups. These modeling dimensions and their classification help engineers to precisely characterize the types of change that a given system can deal with, and how the system can evolve to face them.

To address the above challenges, this section describes a method for the automatic synthesis of self-adaptable choreographies [5]. The *synthesis processor* used by this method takes as input a BPMN2 choreography diagram together with a set of services discovered as possible candidates to play the choreography roles, and automatically generates a set of coordination software entities. When interposed among the services according to a predefined architectural style, these software entities proxify the participant services to coordinate their interaction, when needed. Specifically, coordination entities (called *Coordination Delegates* - CDs) enforce the collaboration specified by the choreography diagram through distributed protocol coordination [7].

The synthesis steps performed by the synthesis processor are described in Section 2.3. The synthesis processor has been fully-implemented within the context of the EU FP7 CHOReOS project as a set of REST services whose open source code is available at `www.choreos.eu`. Moreover, a set of Eclipse plugins that allow for interacting with the REST services according to a predefined development process model is available at `choreos.disim.univaq.it`, and in [3] a tool demo is described on a real CHOReOS case study in the marketing and sales domain. In Section 2.4, we report on recent enhancements of the CHOReOS synthesis processor that are being developed within the context of the EU H2020 CHOReVOLUTION project (`www.chorevolution.eu`), a CHOReOS follow-up project. The main aim of these enhancements is to cope with the automated synthesis of adaptive/evolvable choreographies. An explanatory example showing the enhanced synthesis processor at work is given in Section 2.5.

Following key principles of autonomic computing and related architecture concepts [38], it is widely recognized [26] that, in order to effectively design adaptive systems, feedback loops enabling adaptiveness must become first-class entities. Furthermore, the system engineering process must be rethought in order to break the traditional division among development phases by moving some activities from design-time to deployment- and runtime, hence asking for the exploitation of models at run time [26].

Being inspired by this valuable work in the literature, our enhanced CDs are first-class coordination entities. CDs represent external controllers that realize *multiple interacting feedback loops* enabling choreography self-adaptation at the level of both the supervised participants (local adaptation) and the emergent collaboration among them (global adaptation). In this sense, the choreography-based systems we target are self-aware systems where individual autonomic elements, i.e., CDs manage their internal status and behavior and their relationships with the other autonomic elements in accordance with the choreography specification.

2.3 Method for the synthesis of self-adaptable choreographies

The method presented in this section advances our previous work in [5, 6, 7] by enhancing the synthesis method to also deal with the dynamic evolution of the coordination logic implied by the choreography in response to goal changes.

The synthesis processor takes as input a BPMN2 choreography diagram together with a set of services discovered as possible candidates for the choreography roles, and automatically generates a set of coordination entities (i.e., CDs). Figure 1 pictorially describes the main steps of the synthesis processor.

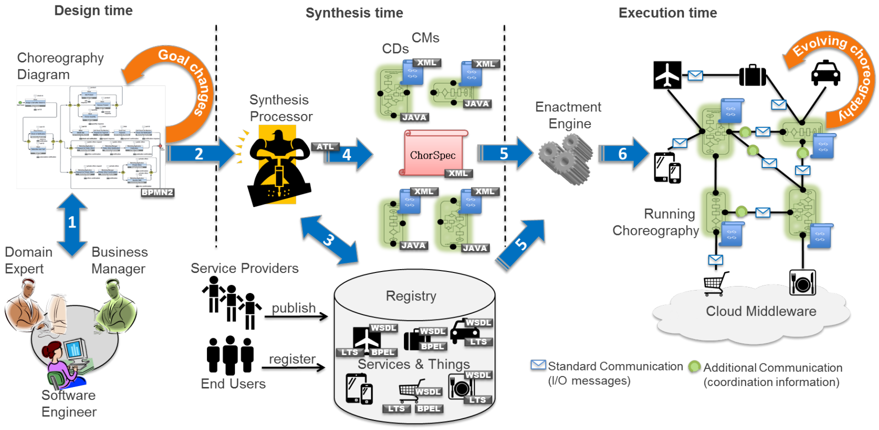


Fig. 1 From choreography design to execution and evolution, through automatic synthesis

Step 1. Software producers cooperate with, e.g, domain experts and business managers to: (i) set the business goal (for example, assist travellers from arrival, to staying, to departure); (ii) identify the tasks and participants required to achieve the goal (for example, reserving a taxi from the local taxi company, purchasing digital tickets at the train station, and performing transactions through services based on near-field communication in a shop), and (iii) specify how participants must collaborate through a BPMN2 choreography diagram.

Step 2. The synthesis processor takes as input the BPMN2 choreography diagram.

Step 3. The synthesis processor queries the registry to discover services suitable for playing the choreography's roles. The registry contains services published by providers (for example, transportation companies and airport retailers) that have identified business opportunities in the domain of interest. As service interfaces description, the synthesis processor assumes WSDL[2]. To describe service interaction behavior, the synthesis processor assumes an automata-based specification, i.e., a Labeled Transition System – LTS, or a BPEL[3] specification.

Step 4. Starting from the choreography diagram and the set of services, the synthesis processor generates the set of CDs through model transformation. The processor also generate the so called `ChorSpec`, a specification of service dependencies to be used by the Enactment Engine (EE) component for deploying and enacting the choreography.

Step 5. The generated CDs, together with the description of the services and their dependencies, serve as input to the EE for deployment and enactment. The description of the EE is outside the scope of this chapter.

Step 6. Following the dependencies, CDs are then interposed among the participant services needing coordination.

When interposed among the services according to a predefined architectural style (an instance of which is shown in Figure 2), CDs proxify the participant services to coordinate their interaction, when needed.

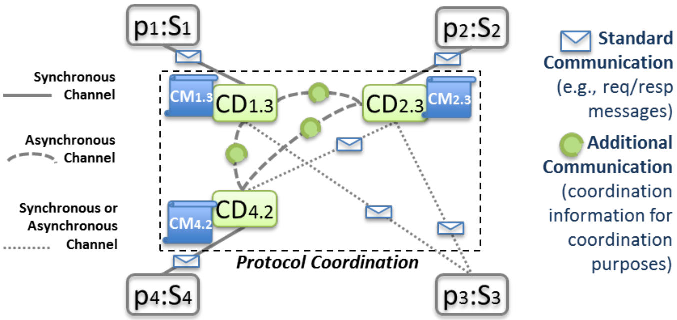

p1:S1
p2:S2
Synchronous Channel
Asynchronous Channel
Synchronous or Asynchronous Channel
CM1.3
CD1.3
CD2.3
CM2.3
CM4.2
CD4.2
Protocol Coordination
p4:S4
p3:S3
Standard Communication (e.g., req/resp messages)
Additional Communication (coordination information for coordination purposes)

Fig. 2 Instance of the predefined architectural style

CDs guarantee the collaboration specified by the choreography specification through distributed protocol coordination [7]. CDs perform pure coordination of the service interactions (i.e., *standard communication* in the diagram) to ensure that the resulting collaboration realizes the specified choreography. For this purpose, the coordination logic is extracted from the BPMN2 choreography diagram and is distributed among a set of *Coordination Models* (CMs) that codify coordination information. Then, at run time, the CDs manage their CMs and exchange this coordination information (i.e., *additional communication*) to prevent possible *undesired interactions* [4, 7], and dynamically self-adapt to possible changes in the specified choreography diagram.

[2] www.w3.org/TR/wsdl

[3] https://www.oasis-open.org/committees/wsbpel

2.4 Dealing with choreography self-adaptation

For choreography-based systems, the choreography diagram represents the concrete specification of the system goal. The source of goal changes is the choreography modeler (i.e., software producers, domain experts, and business managers in Figure 1) who modifies the current specification of the choreography at run time, e.g., by adding/removing choreography tasks to meet new customer needs.

Following the philosophy of models at run time [26], CDs realize multiple interacting feedback loops while managing their own CMs as follows.

CDs implement a MAPE loop [38], i.e., an abstraction of a feedback loop where the dynamic behavior of the managed system is controlled using an autonomic manager. Thus, by following the architectural blueprint for autonomic computing, CDs implement four phases: Monitor (M), Analyzer (A), Planner (P) and Executor (E). Moreover, in order to enable choreography self-adaptation without incurring in disruptive interruptions, the CDs' MAPE loop makes use of the notion of choreography *quiescent* state. For a formal definition of the seminal general notion of quiescent state we refer to [47]. In our context, a choreography is in a quiescent state with respect to given goal changes if the CDs affected by the change are in a quiescent state. Roughly speaking, a CD is in a quiescent state if: (i) the portion of its CM, which is affected by the change, does not involve the current execution state, and (ii) it has completed all service-CD and CD-CD interactions required to perform a task and it has not yet started interactions required for a new task.

After the choreography modelers change the current choreography specification (see goal changes in Figure 1), the synthesis processor re-synthesizes (only) the CMs that are affected by the change, and redistributes them to the interested CDs. Moreover, the processor may also synthesize a new `ChorSpec` due to CDs addition, removal, or substitution of their CMs. The CDs MAPE loop is then realized as follows:

1. **Monitors** – each interested monitor pre-processes the new CM and `ChorSpec`, gathers differences with the previous ones, and informs the analyzer.
2. **Analyzers** – by reasoning on the gathered differences, each interested analyzer establishes the nature of the change, i.e., CD addition or removal, or CM substitution.
3. **Planners** – each interested planner selects suitable actions to enable choreography evolution according to the supported adaptation rules, related mechanisms, and the results of the analyzer. Before executing the required adaptations, a choreography quiescent state has to be reached, and kept all throughout the adaptation process. For this purpose, interested planners communicate with each other to check if their CDs are all in a quiescent state. If it is the case, the planner activates the executor by providing it with the adaptation plan, e.g., the substitution of the CM and, according to a new `ChorSpec`, the substitution of one of the two controlled services. Otherwise, the planner waits for the quiescent state to be reached (if possible). Note that the way CD planners are realized is in line with

the distributed nature of the overall MAPE loop, which we realize by means of multiple interacting feedback loops.

4. **Executors** – after each interested executor is activated by its planner, the executor is in charge of keeping the quiescent state of its CD for the time needed to realize the received adaptation plan. For this purpose, the executor first informs the instance of the distributed coordination algorithm run by its CD (see Section 2.3) to buffer possible incoming service requests. Secondly, the executor interacts with the EE to reconfigure the current architectural, e.g., by deploying/undeploying services and (re-)establishing the new dependencies. Finally, pending service requests are handled once the adaptation process terminates and after the affected portion of the choreography is re-enacted.

2.5 Case study

We applied our method in a case study from the military domain concerning an instance of an Emergency Deployment System (EDS) inspired by the scenario in [2, 50], which is representative of a large number of modern distributed software applications. We implemented a simulation of this system and used it to validate the approach described in the previous sections. Briefly, starting from a BPMN2 Choreography Diagram specification and following the steps shown in Figure 1, the case study shows that our method is able to automatically synthesize the needed CDs and let them dynamically evolve in response of goal changes without re-synthesizing the whole choreography. A detailed description of the case study, including a report on our findings from this work, detailed results, and related implementation, are available at `choreos.disim.univaq.it/downloads`.

3 Synthesis of Self-adaptive Connectors meeting Behavioral and Quality Requirements

Today's networked environment is increasingly characterized by a wide variety of heterogeneous Networked Systems (NSs), including for instance tablets and smartphones, that dynamically decide to achieve goals through interoperation with other systems. Some independently developed heterogeneous applications running on NSs could interact since they use similar interaction protocols implementing compatible functionalities. However, they can exhibit mismatches in their interaction protocols (e.g., different ordering of messages and/or input/ouptut data, or different formats and granularity of input/output data) preventing them to interact seamlessly. Moreover, applications should obey quality requirements and this may undermine their ability to seamlessly interoperate. Achieving interoperability requires solving such mismatches and meeting the quality requirements. This asks for the adaptation of the applications through the use of *connectors* (or *mediators*), that are *the only locus where we can act to make the NSs adaptable*.

Within the self-adaptation area [26], many works can be found in literature tackling several issues by exploiting different techniques. In this section, we concentrate on run-time application interoperability by means of synthesized self-adaptive connectors taking into account both functional and quality concerns. Out of scope of this section are: program synthesis, feedback generation in programming, code generation, or frameworks that automatically manipulate specific files.

In the following, we report on the literature about the synthesis of connectors that take into account behavioral and quality constraints (Section 3.1), and how such techniques can be used to synthesize self-adaptive connectors (Section 3.2).

3.1 QB-Synthesis: Quality and Behavioral Connector Synthesis

The research community has devoted significant effort to the synthesis of connectors since when the seminal paper [66], proposed an adaptor theory to characterize and solve the interoperability problem of augmented interfaces of applications.

Relating to the automated synthesis of behavioral connectors/mediators, within the Web Service area a huge effort have been recently devoted to the description and the automated generation of adaptors, e.g., [54] that is very related to our work.

Works [39, 40] describe an approach for the automated synthesis of *functional* (i.e., behavioral) connectors to reconcile application protocol diversities from a behavioral viewpoint. The approach considers NSs as black boxes which expose within their interface: the application interaction behavior protocol and an ontological description of exchanged messages and data. By reasoning on the ontology, the approach automatically synthesizes a connector between the considered application protocols allowing interoperability.

Concerning combined approaches taking into account both behavioral and quality issues, we can mention papers, [63], and [65]. The first proposes a theoretical framework to identify composite services satisfying both functional and non functional properties. [63] presents an approach to formally specify connector wrappers as protocol transformations, modularizing them, and reasoning about their properties, with the aim to resolve component mismatches. A wrapper is new code interposed between component interfaces and communication mechanisms and its intended effect is to moderate the behavior of the component in a way that is transparent to the component or the interaction mechanism. The paper [65] proposes an approach to automatically derive adaptors in order to assemble correct by construction real-time systems from COTS. The approach takes into account interaction protocols, timing information, and QoS constraints to prevent deadlocks and unbounded buffers. A synthesized adaptor is then a component that mediates the interaction between the components it supervises, in order to harmonize their communication.

Even though the above described approaches take into account both behavioral and quality issues, they do not take into account self-adaptation, that is considered a challenge for modern systems [26].

To the best of our knowledge, three works contributed to the automated synthesis of self-adaptive connectors meeting both functional and quality requirements,

[54] more at middleware level and [28] and [53] at application level. This latter, presents an automated connector synthesis approach for the interoperability at application layer, taking into account both behavioral and quality interoperability at both pre-deployment time and runtime. Functional interoperability refers to applications' behavior and aims at allowing systems to communicate and correctly coordinate. Non functional interoperability is about the required non functional properties (i.e., performance and dependability attributes) that must be guaranteed during systems interoperation. The focus of such work is on issues arising from the execution environment that can be related to the initial uncertainties about the environment and its unpredictable evolution. Such issues can cause violations of the non functional properties. The synthesized connector is self-adaptive: in a first place, it is synthesized to be able to tolerate environment variations and changes within certain ranges. When the environment diverges from the considered assumptions, an adaptation cycle needs to be run according to a provided approach at the cost of re-analysis and synthesis.

3.2 QB-Synthesis of Self-adaptive Connector

In this section we describe a general approach of self-adaptive connectors synthesis (see Figure 3). It is a three steps approach that assumes as input the NS *Applications*, and the QoS properties and requirements of the connected system. A case study that used this approach is described in [28], where the general approach has been tailored on performance properties and applied to a system coming from the e-commerce domain used in CONNECT EU project [4]. For sake of space we here present only the general approach and we refer to [28] for details on its application.

As first step, the approach synthesizes (*Functional Synthesis*) a *Functional* connector (satisfying the functional requirements) that allows the input applications to interoperate on the behavioral side. In this step, by choosing among different settings, one could used different state of the art approaches to synthesizes connectors, e.g. [40]. Then, the approach carries out the *QoS Analysis* on the connected system (i.e., applications and connector) to ensure that it meets the QoS requirement during interaction. In the literature, one can find many analysis approaches that can be used in this step, such as [24, 9]. This step involves analyzing different connected system's configuration alternatives obtained by applying some adaptation strategies. These strategies might include connector behavior slicing, tuning the upper bound on the number of cycle iterations, and choosing the most convenient deployment configuration (these strategies are better described later in this section). Next, the *Analysis Results* are used to instrument the functional connector so that each possible mediation path is decorated with quantitative information (*Decorated* connector) and is executed only if the current QoS requirement is satisfied. The set of enabled paths associated with a QoS requirement is called *mediator configuration*.

[4] https://www.connect-forever.eu/

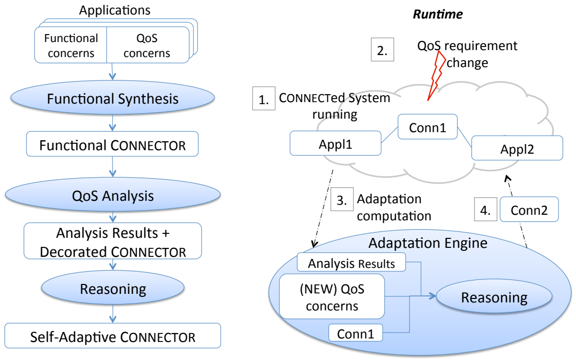


Fig. 3 Connector synthesis overview

The Analysis Results are thus exploited for two different reasons: the first is to decide (*Reasoning*) what is the current connector configuration to deploy (*Self-Adaptive* connector). The second reason is to select at runtime a new mediator configuration after a QoS requirement change thus obtaining a new adapted connector. The runtime adaptation is illustrated in the right part of Figure 3 and is described in the following. The approach synthesizes a self-adaptive connector between the applications and the (1.) the connected system is running. If the QoS requirements change for an application (2.), the adaptation process (3.) is triggered. The connector, as done in [11], is instrumented with monitoring probes that capture the event and triggers our *Adaptation Engine*. The latter takes as input the decorated connector, the analysis results and the new QoS requirement. By reasoning on them, the adaptation engine selects another connector configuration (4.) to be properly deployed and run at the end of the ongoing system transaction.

An example of the realisation of this process, targeting performance concerns, is proposed in Figure 4 [28]. The process combines: (i) the mediator synthesis taking into account the behavioral concerns (upper part of the figure), and (ii) the QoS analysis-based reasoning acting on the intermediary mediator to meet the connected system's performance requirements (bottom part of Figure 4).

The considered approach for the automated synthesis of mediators overcoming interoperability problems between heterogeneous applications, originally introduced in [39, 40], takes as input systems with compatible affordances, i.e., compatible high level capabilities in the same domain and comprises three steps: identification of the common language, behavioral matching, and mediator synthesis. The *Identification of the Common Language*, or *Abstraction* (❶ in Figure 4), among the actions and data of the various applications takes as input the as applications protocol and the subset of the domain ontology they refer to, and identifies the applications common language by classifying their respective (sub-)ontologies into the application domain ontology through an ontology reasoner revealing the correspondences. The common language makes applications behavior comparable to reason on them. The *Behavioral Matching* or *Matching* (❷ in Figure 4) checks the applica-

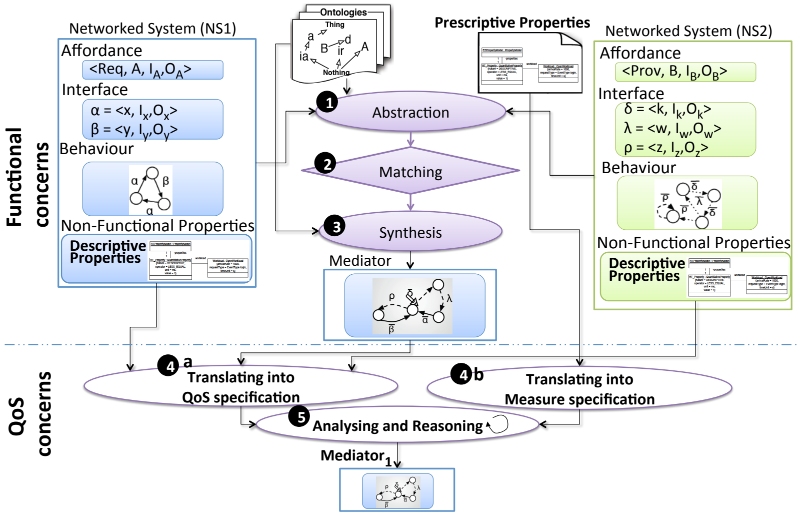


Fig. 4 Overview of the connector synthesis

tions behavioral compatibility, i.e., that the two applications can synchronize at least on one trace reaching one of their respective final states by properly reconciling possible mismatches through a mediator. An extensive description of the identified and managed mismatches is given in [62]. Finally, the *Synthesis* (❸ in Figure 4) automatically produces a (intermediary) mediator that addresses the identified mismatches thus enabling the functional interoperation of the two applications.

The QoS analysis-based reasoning takes into consideration QoS concerns during the synthesis. This is done by acting on the intermediary mediator produced before. This reasoning is composed by two steps detailed in the following: Generation of a QoS Model and QoS analysis-based reasoning. The *Generation of a QoS Model* is composed by two activities: the first one, Figure 4 **❹a**, takes as input the LTS-based specification of the NSs and of the synthesized mediator, and the *descriptive properties* to generate the QoS Model. To execute QoS analysis, a *measure specification* that defines the QoS indices of interest is required. The second activity, Figure 4 **❹b**, translates the *prescriptive property* defining the QoS requirements on the final system, into the required *measure specification*. In the *QoS analysis-based reasoning*, from the QoS specification and the measure specification, the QoS analysis is executed. If the composed system satisfies the specified requirements then the mediator is not modified, otherwise it undergoes a reasoning step that tries to obtain a mediator showing better QoS (see Figure 4 ❺). In [28], the performance analysis-based reasoning leverages the Æmilia ADL [10]. As mentioned before, to improve the composed system QoS we can act on the mediator to slice alternative behaviors, limit the number of execution of cycles, and find out the best deployment. At the end of this step two scenarios are possible: i) we obtain a mediator that allows the composed system to meet the QoS requirements; ii) all the refined mediators obtained by applying the identified strategies do not allow the composed system to meet the

QoS requirements. In this case the approach does not produce a mediator and asks to relax the QoS requirement in order to provide a suitable system.

Adaptation Strategies. The proposed approach tries to act on the mediator to satisfy the quality requirement. In [28], three possible *strategies* are identified and can be applied singly or in combination: i) *connector behaviors slicing* that can be applied if at least one of the NSs protocols has alternative paths to achieve communication. In this case the connector behavior is sliced and it mediates only a subset of the NSs communication paths; ii) *tuning the upperbound on the number of cycle iterations*. If the protocols contain cycles, several bounds are considered in the analysis and only the ones that allow the satisfaction of the QoS requirements are considered in the final synthesized connector. Finally iii) *choosing the most convenient deployment configuration* among three possibilities: *all remote* where the mediator and the applications are deployed on separate machines, and *local to NS1* or *local to NS2* where the mediator is deployed on the same machine where either one application or the other is running, NS1 or NS2 respectively. It is worth noticing that the synthesis process generates the *most general* connector that satisfies the functional and non-functional requirements. Most general means that, with respect to the functional mediation, the connector protocol prevents the execution only of the paths that do not satisfy non-functional requirements.

Due to context evolutions, and in particular to runtime changes on the QoS requirements, it might also happen that at runtime the connected system does not satisfy the new requirements. To address this issue, the proposed approach performs a runtime adaptation of the connector based on the three strategies identified above thus obtaining a new connector.

Adaptation Reasoning. Based on the applicable strategies, the reasoning identifies a number of different mediator configurations (# mediator configurations = # possible slicing * # different upper bounds on cycles * # different deployment). For instance, the combination of the strategies in a case study in [28] gave us 54 final configurations to consider and hence 54 corresponding experiments to conduct.

The initial QoS requirement, from the analysis results performed before during the synthesis, can be satisfied by several configurations. Among them, it is selected the best configuration based on one *policy* or more, e.g., in [28] it is selected the admissible configuration that maximizes the user interactions (no mediator slicing and highest upper bound on the number of the number of cycles in the protocol).

This *runtime change of QoS requirement triggers an adaptation*, since the new requirement is not satisfied by the previous mediator configuration. The adaptation is realized by selecting, among all the admissible mediators configurations, the one that better suits the new requirement taking into account the policy (e.g., maximize the user interactions).

3.3 Open Issues

Nowadays it is possible to enable interoperability between heterogeneous protocols leveraging automatically synthesized self-adaptive mediators that let them interact seamlessly. In this setting, many works in literature proposed a solution to automated mediator synthesis that addresses the functional facet of the problem. Additionally, few approaches have been recently proposed to also deal with non-functional aspects and with adaptations of the connector due, for instance, to runtime QoS requirement changes.

However, the behavioral connector synthesis and the quality analysis approaches suffer from the usual limitations of model-based (stochastic) analysis and synthesis (e.g., state explosion). Hence, more efficient methods are needed to make behavioral synthesis and quality analysis widely applicable. Toward this direction, a study on how to optimize the designed approaches should be made. This would improve the runtime applicability and scalability of the overall approach for the synthesis of self-adaptive connector that considers both behavioral and quality concerns.

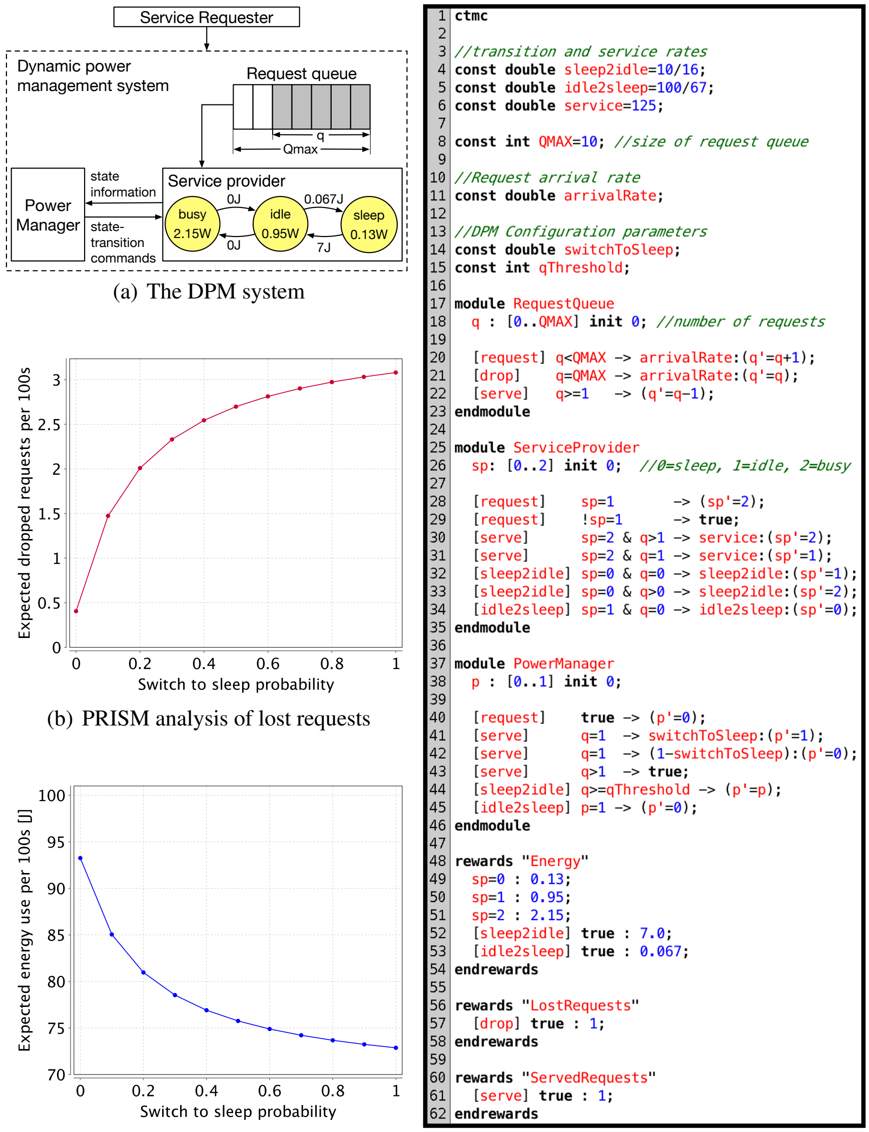
4 Quantitative verification at runtime

The integration of formal verification techniques into the reconfiguration of self-aware systems aims to guarantee that these systems continue to meet their requirements as they change over time. The formal verification paradigm overviewed in this section extends the applicability of *quantitative verification* to self-aware systems.

Quantitative verification [48] is a mathematically based technique for analysing the correctness, reliability, performance and other QoS properties of systems characterised by stochastic behaviour. The technique carries out its analysis on finite state-transition models comprising states that correspond to different system configurations, and edges associated with the transitions that are possible between these states. Depending on the analysed QoS properties, the edges are annotated with transition probabilities or transition rates; additionally, the model states and transitions may be labelled with *rewards*. The types of models with probability-annotated transitions include discrete-time Markov chains (DTMCs) and Markov decision processes (MDPs), while the edges of continuous-time Markov chains (CTMCs) are annotated with transition rates.

Given one of these models and a QoS property specified formally in a variant of temporal logic extended with probabilities and rewards, the technique analyses the model exhaustively to evaluate the property. Examples of properties that can be established using the technique include the probability that a system component operates correctly, the expected reliability of a composite service, and the expected energy consumed by an embedded system. Quantitative verification is typically performed automatically by tools termed *probabilistic model checkers*.

We will illustrate the use of quantitative verification for a dynamic power management (DPM) system with the structure in Fig. 5(a). This system comprises a *service provider* responsible for handling requests generated by a *service requester* and stored in a finite request queue. The service provider can operate in three different modes—*busy*, *idle* and *sleep*—that correspond to different power usages and operation rates. Fig. 5(a) depicts the power usage of each mode (in watts), the pos-



(a) The DPM system

Expected dropped requests per 100s
0
0.5
1
1.5
2
2.5
3
0
0.2
0.4
0.6
0.8
1
Switch to sleep probability

(b) PRISM analysis of lost requests

(c) PRISM analysis of power usage

```
ctmc

//transition and service rates
const double sleep2idle=10/16;
const double idle2sleep=100/67;
const double service=125;

const int QMAX=10; //size of request queue

//Request arrival rate
const double arrivalRate;

//DPM Configuration parameters
const double switchToSleep;
const int qThreshold;

module RequestQueue
  q : [0..QMAX] init 0; //number of requests

  [request] q<QMAX -> arrivalRate:(q'=q+1);
  [drop]    q=QMAX -> arrivalRate:(q'=q);
  [serve]   q>=1   -> (q'=q-1);
endmodule

module ServiceProvider
  sp: [0..2] init 0;  //0=sleep, 1=idle, 2=busy

  [request]    sp=1       -> (sp'=2);
  [request]    !sp=1      -> true;
  [serve]      sp=2 & q>1 -> service:(sp'=2);
  [serve]      sp=2 & q=1 -> service:(sp'=1);
  [sleep2idle] sp=0 & q=0 -> sleep2idle:(sp'=1);
  [sleep2idle] sp=0 & q>0 -> sleep2idle:(sp'=2);
  [idle2sleep] sp=1 & q=0 -> idle2sleep:(sp'=0);
endmodule

module PowerManager
  p : [0..1] init 0;

  [request]    true -> (p'=0);
  [serve]      q=1  -> switchToSleep:(p'=1);
  [serve]      q=1  -> (1-switchToSleep):(p'=0);
  [serve]      q>1  -> true;
  [sleep2idle] q>=qThreshold -> (p'=p);
  [idle2sleep] p=1 -> (p'=0);
endmodule

rewards "Energy"
  sp=0 : 0.13;
  sp=1 : 0.95;
  sp=2 : 2.15;
  [sleep2idle] true : 7.0;
  [idle2sleep] true : 0.067;
endrewards

rewards "LostRequests"
  [drop] true : 1;
endrewards

rewards "ServedRequests"
  [serve] true : 1;
endrewards
```

(d) PRISM code for the DPM system

Fig. 5 Modelling and quantitative verification of DPM system properties

sible transitions between modes, and the energy consumed by each transition (in joules). These values are taken from [56] and correspond to a Fujitsu disk drive.

The DPM system operates as follows. Upon receiving a new request, the service provider automatically transitions to the *busy* mode, if it is in the *idle* mode; otherwise (i.e., if in the *sleep* or *busy* mode), its mode remains unchanged. If the queue contains $q > 1$ requests and the service provider is in the *busy* mode, one of these requests is processed. Upon processing the last request, $q = 0$, the service provider automatically transitions to the *idle* mode. A software power manager controls the transitions between the *sleep* and *idle* service provider modes. In doing so, the power manager considers the state of the system and aims to reduce power use while maintaining an acceptable service level.

Fig. 5(d) shows a CTMC model of the DPM system. This model is adapted from [56], and is defined in the high-level modelling language of the probabilistic model checker PRISM [49]. The *RequestQueue*, *ServiceProvider* and *PowerManager* modules within this model correspond to the similarly named components of the DPM system. The local variables from the RequestQueue and ServiceProvider modules represent the number of requests within the queue (q) and the service provider mode (sp). Synchronisation between all modules occurs through the *request* and *serve* actions, which denote requests arriving into the queue and being processed, respectively. The service provider manages the transitions between the *busy* and *idle* operating modes (lines 28–31). In contrast, the transitions between *idle* and *sleep* are controlled by the power manager through the synchronisation of ServiceProvider and PowerManager commands using common actions (lines 32–34 and 44–45). For instance, when the request queue becomes empty, the power manager performs an *idle* to *sleep* transition with probability *switchToSleep* (lines 41–42). Similarly, when the request queue grows to $q \geq qThreshold$ requests, the power manager performs a *sleep* to *idle* transition to ensure that the service provider starts handling requests (line 44).

Finally, CTMC states and transitions are associated with **rewards...endrewards** structures (lines 48–62). The *"Energy"* reward structure encodes the energy consumed by the service provider in each mode and during transitions between modes. The *"DroppedRequests"* and *"ServedRequests"* reward structures associate a reward of 1 with the transitions where requests are dropped due to a full queue and served, respectively. Using these reward structures and a reward-augmented variant of continuous stochastic logic (CSL), it is possible to specify and analyse QoS properties of the DPM system. For example, Fig. 5(b) and 5(c) depict the expected number of dropped requests and the expected energy use over 100s of system operation, as a function of the *switchToSleep* probability, when $arrivalRate = 0.8\text{s}^{-1}$ and $qThreshold = 8$. These results were obtained by analysing the CSL reward properties $R_{=?}^{\text{“}DroppedRequests\text{”}}[C^{\leq 100}]$ and $R_{=?}^{\text{“}Energy\text{”}}[C^{\leq 100}]$, respectively.

4.1 Application to self-aware systems

Extending the applicability of quantitative verification to self-aware systems requires the continual use of the technique at run time [15]. To this end, *quantitative verification at runtime* integrates this formal verification technique into the MAPE (Monitor-Analyse-Plan-Execute) closed control loop of self-aware systems.

Quantitative verification at runtime requires the monitoring of self-adaptive systems and their environment, in order to identify relevant changes and quantify them using fast on-line learning techniques. These observations are used to continually

Table 1 QoS requirements for the self-aware DPM system

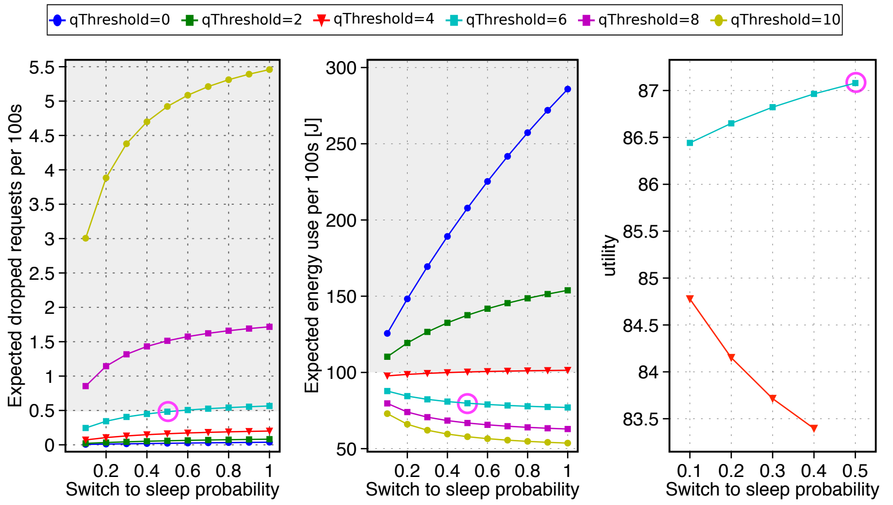
ID	Informal description	Formal specification
R1	*Performance*: “The expected number of dropped request per 100s of running time should not exceed 0.5.”	$R_{\leq 0.5}^{\text{“}DroppedRequests\text{''}}[C^{\leq 100}]$
R2	*Energy use*: “The service provider should consume at most 100 Joules per 100s of running time.”	$R_{\leq 100}^{\text{“}Energy\text{''}}[C^{\leq 100}]$
R3	*Utility*: “Subject to R1 and R2 being met, the DPM system must use a configuration that maximises $$utility(switchToSleep, qThreshold) = w_1 S + w_2/E,$$ where S is the number of requests served given by evaluating the property $R_{=?}^{\text{“}ServedRequests\text{”}}[C^{\leq 100}]$ and E is the energy used by the service provider per 100s of system running time. The weights $w_1, w_2 > 0$ express the tradeoff between system throughput and battery usage.	find $\mathsf{argmax}\ utility\ (switchToSleep, qThreshold)$ such that **R1** and **R2** are satisfied

update a probabilistic model of the system, starting from an initial model provided by the system developers. For example, fast on-line learning techniques are used in [16, 17, 29] to monitor the changing probabilities of successful service invocation for service-based systems, and thus to update DTMC models of these systems. Probabilistic model checking performed at runtime is then used to re-verify the compliance of these updated models with QoS requirements related to the system response time, reliability, cost, etc. If QoS requirement violations are identified or (when the functionality associated with the unsatisfied requirements has not been exercised) predicted, the results of the analysis support the synthesis of a reconfiguration plan. Executing this plan ensures that the self-adaptive system will continue to satisfy its QoS requirements despite the changes identified during monitoring.

To illustrate the use of the technique, suppose that the DPM system from the previous section must comply with the QoS requirements from Table 1 in the presence of changes in the request arrival rate. To achieve this compliance, the power manager uses quantitative verification at runtime. Request arrivals are monitored to establish the current request arrival rate. This information is used to continually update the CTMC model from Fig. 5(d). The updated CTMC model is then verified to identify values for

- the probability *switchToSleep* for setting the service provider to *sleep* mode, and
- the parameter *qThreshold* for switching the service provider to *idle* mode

which ensure that the DPM system meets its QoS requirements for the current request arrival rate. As an example, Figs. 6(a) and 6(b) show the results of the quantitative verification carried out at runtime when $arrivalRate = 0.65\text{s}^{-1}$. These results establish the expected number of dropped requests and the energy usage for a range of possible *switchToSleep* and *qThreshold* values. The shaded areas from Fig. 6(a) and 6(b) correspond to parameter values (i.e., to power manager *configurations*) that violate requirements **R1** and **R2**, respectively. These configurations are discarded.



(a) (b) (c)

Fig. 6 Verification results for (a) requirement **R1** and (b) requirement **R2** from Table 1, and (c) *utility* of the valid configurations of the DPM system

Next, the *utility* of the remaining, feasible configurations is computed. Fig. 6(c) depicts the utility values calculated for the feasible configurations from Figs. 6(a) and 6(b), with $w_1 = 1$ and $w_2 = 2000$ in the utility formula from Table 1. The configuration that maximises the system utility is circled in Fig. 6(a)–(c). This configuration is adopted by the power manager when $arrivalRate = 0.65\text{s}^{-1}$.

4.2 Research challenges

Quantitative verification at runtime is a new area of research. Although successful applications have emerged in QoS optimisation for service-based systems [16, 29], reconfiguration of cloud computing infrastructure [42], and adaptive resource management in embedded systems [13], several research challenges remain unsolved.

In particular, the state explosion problem that affects model checking is an even greater challenge for quantitative verification at runtime. In the case of self-aware systems, the probabilistic model checking of an updated model needs to be performed not only with acceptable overheads for the verified system, but also fast enough to support timely self-adaptation, and before the model is rendered obsolete by the next update. Recent research has proposed preliminary approaches to tackling this challenge through *parametric verification* (described in the next section), *compositional* and *incremental* [42] *verification*, and, more recently, *decentralised quantitative verification* [13]. Nevertheless, further research is needed to extend these approaches and to enable the application of the technique to large, rapidly changing and distributed systems.

Even when quantitative verification at runtime can be performed very fast for the model associated with a configuration of a self-aware system, the extremely large configuration spaces of many such systems pose a major challenge. The recently proposed search-based software engineering approach to *probabilistic model synthesis* [34] may have the potential to help address this challenge, as it operates without an exhaustive exploration of the configuration space. Assessing the applicability of these solutions to self-aware systems remains an area of future research.

Finally, another key challenge is to ensure that the probabilistic models verified at runtime are accurate. Despite the development of effective techniques for learning the transition probabilities of Markov chains from runtime observations [16, 17, 29], the current use of point estimates for these probabilities may introduce unquantified estimation errors. Quantitative verification with confidence intervals has been introduced recently to address this problem [14], although its use in self-aware systems is yet to be explored.

5 Parametric verification

Many systems that are subject to verification are inherently stochastic. Examples include randomized distributed algorithms (where randomization breaks the symmetry between processes), security (e.g., key generation at encryption), systems biology (where species randomly react depending on their concentration), embedded systems (interacting with unknown and varying environments), and so forth. A well-known example is the *crowds* protocol, which employs random routing to ensure anonymity. Nodes randomly choose to deliver a packet or to route it to another randomly picked node. In the presence of "bad" nodes that eavesdrop, we could be interested in analyzing probabilistic safety properties such as "the probability of a bad node identifying the sender's identity is less than 5%".

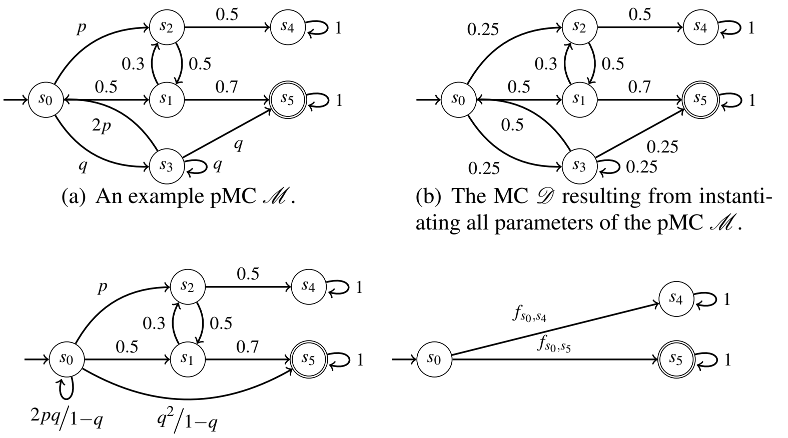
In the recent past, different automata- and tableau-based *probabilistic model-checking* techniques to prove model properties specified by, e.g., probabilistic ω-regular languages or probabilistic branching-time logicssuch as pCTL and pCTL* have been developed [37]. Probabilistic model checking is applicable to a plethora of probabilistic models, ranging from discrete-time Markov chains to continuous-time Markov decision processes and probabilistic timed automata, possibly extended with notions of resource consumption (such as memory footprint and energy usage) using rewards (or prices). For instance, PRISM [49] or MRMC [44] are mature probabilistic model checkers and have been applied successfully to a wide range of benchmarks.

A major practical obstacle is that these quantitative verification techniques and tools work under the assumption that *all probabilities in models are a priori known*. However, at early development stages, certain system quantities such as faultiness, reliability, reaction rates and packet loss ratios are often not (or at the best partially) known. In such cases, *parametric* probabilistic models can be used for specification, where transition probabilities are specified as arithmetic expressions using real-valued parameters. In addition to checking instantiated models for fixed parameter values, the important problem of parameter synthesis arises, posing the question which parameter values lead to the satisfaction of certain properties of interest. Parametric models are very natural in adaptive and self-aware software where "continuous" verification frequently amends system models during deployment as well as in model repair, where probabilities are dynamically tuned so as to satisfy a desired property. This section will describe the state of the art in parametric quantitative verification and its usage in model repair.

5.1 Parametric Markov Chains

We will first introduce the model at hand. A *parametric discrete-time Markov chain* (pMC) is defined as a usual discrete-time Markov chain (MC), with probabilities given by *rational functions* (fractions of polynomials) over a given set of parameters.

Consider the example pMC given in Figure 7(a). From state s_0 three transitions emerge, one transition carrying probability 0.5 and two parametric transitions with the parameters p and q. In order for these three transitions to form a suitable *probability distribution*, one needs to make sure that the parameters are only instantiated in a way such that all probabilities emanating from s_0 sum up to one. We call such an instantiation a *valuation* of parameters, which is a function mapping from the set of parameters to the real numbers. A valuation inducing probability distributions is called *well-defined*. For instance, the valuation v with $v(p) = v(q) = 0.25$ induces a well-formed MC which is depicted in Figure 7(b).



(a) An example pMC $\mathscr{M}$.

(b) The MC $\mathscr{D}$ resulting from instantiating all parameters of the pMC $\mathscr{M}$.

(c) The pMC $\mathscr{M}$ resulting from eliminating state s_3.

(d) The pMC $\mathscr{M}$ resulting from eliminating all states but s_0, s_4, and s_5.

Fig. 7 Parametric Markov chains

The properties we investigate are so-called *reachability properties*, i. e. to compute the probability of reaching a dedicated set of target states. For instance, for the MC in Figure 7(b) the probability to reach state s_5 from the initial state s_0 is 0.72. For pMCs, this can be solved by computing a rational function over the occurring parameters describing the reachability probability. This means, if the parameters of the function are instantiated by a valuation that is well-defined for the original pMC, this will evaluate to exactly the reachability probability of the corresponding instantiated MC. In our running example, the functions describing reachability probabilities from s_0 to s_4 and s_5 are given by:

$$f_{s_0,s_4} = \frac{40p^2 + 20pq + 6p + 3q}{68p^2 + 34pq + 34q^2 + 34p + 17q}$$

$$f_{s_0,s_5} = \frac{28p^2 + 14pq + 34q^2 + 28p + 14q}{68p^2 + 34pq + 34q^2 + 34p + 17q}$$

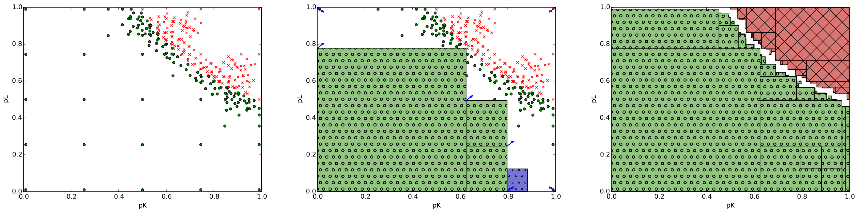
This example shows that already for very simple benchmarks rather complicated functions might occur.

5.2 State of the art

In 2004, Daws [25] first proposed to represent reachability probabilities in pMCs by means of rational functions, which are obtained by state elimination (as for obtaining a regular expression from automata). The basic idea is to "bypass" a state s by removing it from the model and increasing the probabilities $P(s_1, s_2)$ of the transitions from each predecessors s_1 to each successors s_2 by the probability of moving from s_1 to s_2 via s, possibly i cluding a self-loop on s.

Consider again the pMC in Figure 7(a). Assume, state s_3 is to be eliminated. The states that are relevant for this procedure are the only predecessor s_0 and the successors s_0 and s_5. Applying state elimination yields the model in Figure 7(c). Moreover, if all intermediate states are eliminated, only transitions directed from initial states to absorbing states remain. The result is depicted in Figure 7(d). This is what we call the model checking result for parametric probabilistic verification.

This technique has been improved by Hahn *et al.* [36] by directly computing and simplifying intermediate functions, as a major drawback of the state elimination technique is the rapid growth of rational functions. The simplification involves the addition of rational functions where the costly operation of computing the greatest common divisor (gcd) needs to be performed. Jansen *et al.* [41] further improved the state elimination technique by combining it with SCC decomposition, and a dedicated gcd-computation operating on partial factorizations of polynomials. State elimination is the core of the tool PARAM [36] and has recently been adopted in PRISM [49]. The new tool PROPhESY [27] also employs variants of state-elimination. These are—to the best of our knowledge—the only available tools for computing reachability probabilities (and expected rewards) of pMCs. For the



(a) Sampling (b) Coarse region partition (c) Fine region partition

Fig. 8 Sampling and region analysis

common benchmarks available at the PRISM website [49], PROPhESY performs best on nearly all benchmark instances both for reachability probabilities and expected rewards. In general, for systems with two parameters instances having up to 10 million states can be handled within reasonable time.

With the exception of PROPhESY, the available tools just output the rational function sometimes accompanied by constraints ensuring well-definedness. The problem of parameter synthesis is thereby not addressed directly. Other works consider parameter synthesis of timed reachability in parametric CTMCs [19].

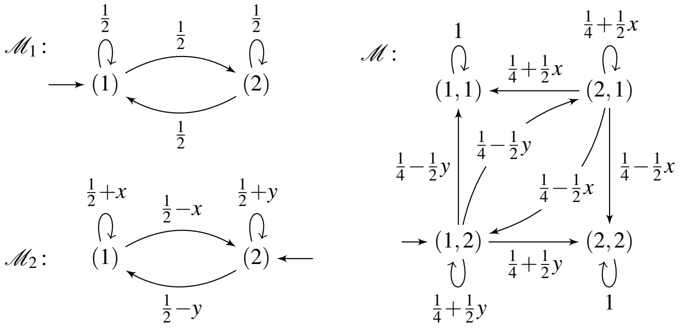
In model repair [8], models refuting a given property are amended so as to satisfy this property. In this setting, parametric MCs are used as underlying model.

5.3 Parameter synthesis

In our setting, a *requirement* is given as an upper bound on reachability probabilities. For instance, for the function f_{s_0,s_5} as in Figure 7(d) one might give a value $\lambda \in [0,1]$ such that $f_{s_0,s_5} < \lambda$. Now, to determine whether the requirement is met, one has to consider *all possible parameter valuations* for p and q. These *parameter synthesis* problems are challenging and substantially more complex than verifying standard MCs—just checking whether a pMC is realizable (having a parameter evaluation inducing a well-defined MC) is exponential in the number of parameters.

The tool PROPhESY addresses this problem as follows. To give the user a feasible and usable approach, an (approximate) partitioning of the parameter space into *safe* and *unsafe regions* is computed. Each parameter instantiation within a safe region satisfies the requirement under consideration, while inside unsafe regions, no instantiation meets the requirement.

This is done in an incremental fashion: After the rational function is computed, the first step is to *sample* the rational function up to a user-adjustable degree. This amounts to instantiating parameter values (determined by dedicated heuristics) over the entire parameter space. This yields a coarse approximation of parts of the solution space that are safe or unsafe and can be viewed as an abstraction of the true partitioning into safe or unsafe parts. A typical sampling result with respect to two



(a) Environment $\mathscr{M}_1$ and strategy $\mathscr{M}_2$ (b) Parallel composition $\mathscr{M}$

Fig. 9 Parametric Markov chain for model repair

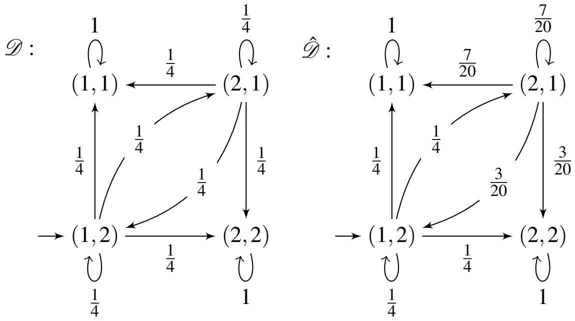
parameters can be seen in Figure 8(a). Points (or rather parameter instantiations) that satisfy the requirement are drawn green, the other are red.

The goal is now to divide the parameter space into regions which are *certified* to be safe or unsafe. This is done in an iterative CEGAR-like fashion. First, a region candidate assumed to be safe or unsafe is automatically generated. An *SMT solver* like Z3 [43] is then used to verify the assumption. In case it was wrong, a *counterexample* in the form of a contradicting sample point is provided along which the abstraction/sampling is *refined*, giving a finer abstraction of the solution space. Using this, new region candidates are generated. A very coarse partition into such regions is shown in Figure 8(b), a fine partition covering over 90% of the parameter space is shown in Figure 8(c). For the used benchmarks, a covering of over 95% can be achieved within seconds. After that it is increasingly costly to determine the rest.

5.4 Model repair

A still open and important problem is how to *automatically and efficiently repair* a MC model that does not meet a certain requirement. This application of parametric models is highly relevant for several applications such as in the area of robotics. A first approach was presented by Bartocci *et al.* [8], based on defining a *non-linear optimization problem* encoding that the system is changed with minimal cost such that desired property is satisfied. The main practical obstacle of using non-linear optimization, be it using a dedicated optimization algorithm or using an SMT-solver for non-linear real algebra [43] coupled with a binary search towards the optimal solution, is *scalability*. As the optimization involves costly computations of greatest common divisors of polynomials, approaches like [36, 41] are inherently restricted to small pMCs with just a few parameters.

Recently, Pathak *et al.* [55] proposed a heuristic approach to model repairs motivated by a robotics scenario. The method starts from an initial parameter assignment and iteratively changes the parameter values by *local repair steps*. To illustrate the



(a) Initial MC $\mathscr{D}$ (b) Repaired MC $\hat{\mathscr{D}}$

Fig. 10 Model repair

basic idea, assume a model in which the probability to reach some "unsafe" states is above an allowed bound. Using model checking we know for each state the probability to reach "unsafe" states. The higher this probability, the more dangerous it is to visit this state. To repair the model, we iteratively consider single probability distributions in isolation, and modify the parameter values such that we decrease the probability to move to more dangerous successor states. The approach is shown to be *sound* in the sense that each local repair step improves the reachability probability towards a desired bound for a repairable pMC.

To illustrate the robotics scenario and the proposed technique, consider a toy example from [55] given in Figure 9(a). An object moves between two places (1) and (2) according to the MC $\mathscr{M}_1$. To catch the object, a robot moves between the places according to a strategy modelled by the pMC $\mathscr{M}_2$. With certain parameter domains, this defines the degree of freedom one has to *perturb* the strategy. For these two systems, the synchronous parallel composition is depicted as the pMC $\mathscr{M}$ in Figure 9(b). Intuitively, the probability to actually catch the ball is the probability to finally reach the states $(1,1)$ and $(2,2)$.

Assume now that for some reason it is dangerous to catch the ball at (2). We therefore want to decrease the probability of reaching state $(2,2)$ in $\mathscr{M}$, say it should be smaller than 0.5. Assume furthermore, an initial valuation of parameters leads to the MC $\mathscr{D}$ depicted in Figure 10(a) by basically instantiating all parameters with zero. In $\mathscr{D}$, the probability to reach $(2,2)$ is 0.5, i. e., the requirement is not met. Locally changing the probability distributions towards the requirement means, that the reachability probability needs to be decreased. A possible result is the MC $\hat{\mathscr{D}}$ depicted Figure 10(b), where the probability to reach $(2,2)$ is now only $4/9$, which means that the requirement is met. We call $\hat{\mathscr{D}}$ a *repair* of $\mathscr{D}$. Experiments show that this approach is feasible for systems with millions of states.

6 Runtime verification and probabilistic models

The goal of system health management is similar to that of a healthcare system, which aims to keep people healthy. System health management [23], in contrast, focuses on artificial or engineered systems, including aerospace [59, 60], electrical power [51, 57], infrastructure, mobile, and software systems [52, 58], and aims to keep them (and consequently also their users and operators) healthy or safe. Key system health management processes are detection, isolation, prognosis, and mitigation. Detection amounts to asking "Is something wrong or faulty in the system?" Isolation asks "Which components, if any, are faulty?" Prognosis is forecasting, and asks the question "Is the system or any components going to fail soon?" Finally, mitigation focuses on the question "What can I do in order to work around failures?"

The methods used to address each of these four questions, along with their relative importance, vary from application area to application area. In the following we will focus on general methods—namely probabilistic graphical models [45] and runtime verification [61]—and illustrate them by means of aerospace applications. Unmanned aerial systems (UASs), some of which are also known as drones, will be emphasized.

In general, there are two systems involved [23]. One system is the system being monitored; this can be a mechanical system, an electrical power system, a mobile phone, or a software system. Second, there is the monitor. In our case, the monitor is called a system health monitor (SHM). The SHM is typically a computer running adjacent to the system being monitored, and is taking as input sensor readings from the monitored (or target) system. The concept of sensor reading should here be interpreted broadly, and can be vibration measurements for a mechanical system, voltage measurements for an electrical power system, or log file entries for a software application. What these sensor readings have in common is that they reflect the target's health status, and are used by the system health monitor to compute estimates of the target's current and future health status. Often, the SHM relies on the use of one or several models, along with algorithms operating on them, and a key questions system health management centers around how these models are expressed, how they are developed, what they are used for, and how computation with them is performed.

We are in particular focused on models that are represented as probabilistic graphical models, and specifically using Bayesian networks and arithmetic circuits. Bayesian networks have received substantial attention both from researchers and practitioners, and have been successfully employed for target systems such as electrical power systems and software systems. Sometimes, not only the monitored system is being modeled, but also its sensors. This enables the one to distinguish between system failure and sensor failure in the SHM model and thus by the SHM monitor.

Bayesian network models for system health management can be developed manually by developers in collaboration with subject matter experts as well as in a completely data-driven fashion [64]. There are also hybrid development methods, in which the structure of the Bayesian network is constructed manually, while its parameters are estimated from data, perhaps informed by a Bayesian prior. Once a probabilistic graphical model for SHM has been developed, there is the question of its V&V as well as deployment. A Bayesian network model can be used directly by an SHM monitor for online system health management, or it can be compiled to a secondary model which is then used by SHM monitor. Data structures for such secondary models include junction trees and arithmetic circuits. Benefits of using a secondary data structure include faster computation, simpler SHM algorithms, more predictable compute time and memory consumption, and lower power consumption. In addition, different computational platforms can come into play, including not only CPUs but also graphics processing units (GPUs) [67, 68] and field programmable gate arrays (FPGAs).

Regardless of whether a Bayesian network model or a compiled model is used for SHM monitoring, a posterior distribution $P(H \mid e)$ is computed by means of the probabilistic graphical model. Here, e is the input to the model (computed from the target or monitored system), while H is a set of random variables that reflect the health status of the target system. One random variable among this set can, for example, represent the health status of one component or one subsystem in the target system. Each component in the target system is typically represented by a handful of random variables in the SHM model. The graph structure of the model will reflect dependencies and independencies of the target system, and will also reflect the structure of the problem that we are trying to solve.

There are at least three key design requirements for SHM in aerospace [60]: unobtrusiveness, responsiveness, and realizability. *Unobtrusiveness* means that the SHM framework must not change in any way the properties of the vehicle.[5] *Responsiveness* means that the SHM framework must operate in real time. *Realizability* means that the SHM framework must easily plug into the existing hardware and software stack of the UAS. One cannot expect that this stack can be easily changed; an SHM component needs to blend into an existing ecosystem without too much "pain."

The above requirements have been met in the novel rt-R2U2 framework [60]. The rt-R2U2 framework is a hybrid or modular one, in which different models are, to a large extent, developed and implemented separately but with well-defined interfaces. Typical building blocks of rt-R2U2 are the following: signal processing blocks, temporal logic blocks, Bayesian reasoning blocks, and miscellaneous computing blocks. These blocks can be connected in a block diagram fashion, thus a temporal logic block can take as input results from a Bayesian reasoning block and vice versa. Bayesian reasoning blocks contain arithmetic circuits compiled from Bayesian networks. Bayesian reasoning blocks are typically used to estimate component and sensor state-of-health from noisy and uncertain sensor data. Tempo-

[5] For example, for manned flight there is a stringent certification requirement. For both unmanned and manned flight, there is typically stringent resource limitations. These resource requirements may vary from aerospace vehicle to aerospace vehicle, but are often concerned with electrical power consumption, weight, and computational needs. These requirements are different from what is often seen in other applications of verification or runtime verification, and impact the techniques developed to meet the requirements.

ral logic blocks, on the other hand, contain temporal logic formulas that express aerospace safety requirements. Linear Temporal Logic (LTL) and Metric Temporal Logic (MTL) are often used to formalize safety requirements in aerospace. The key benefit of the rt-R2U2 framework is that such temporal formulas can be used for runtime verification, even on a UAS under the three quite strict design requirements discussed above (unobtrusiveness, responsiveness, and realizability). The rt-R2U2 framework has been implemented on an FPGA, and overall this approach meets the three design requirements identified above, namely unobtrusiveness, responsiveness, and realizability.

The rt-R2U2 framework has been successfully implemented and validated using data from the Swift UAS, an unmanned all-electric aircraft at NASA [60]. In the Swift UAS, there is a read-only interface from the bus attached to the main flight computer. This read-only interface is used by the SHM monitor, which is implemented by means of an FPGA. The FPGA is used to compute with an arithmetic circuit (AC) model compiled from a Bayesian network model. In addition to a Bayesian network model, runtime verification methods are being used. These runtime verification techniques are evaluating temporal logic formulas on the FPGA. These temporal logic formulas express flight rules, operational limitations of the Swift UAS, and so forth. Inputs to an AC model can be sensor readings, filtered sensor readings, or outputs of another Bayesian reasoning or temporal logic block. This enables a powerful and flexible system health management capability for a UAS, and we believe it could be foundation for similar SHM systems outside of aerospace.

There is a close relationship between the techniques discussed in this section and those discussed in Section 4 and Section 5. However, there are some additional requirements and constraints in place, due to the aerospace or more broadly transportation setting that is being focusing on here. For such aerospace (runtime) verification techniques, there are requirements and concerns beyond the concern for the potential of a state space explosion found in some traditional verification and model checking techniques.

7 Analysis and synthesis for self-adaptation exploiting environment assumptions

During the construction of a self-aware computing system, developers face many decisions about the system's architecture and design. For example, questions such as whether to employ centralized or decentralized decision-making, or whether adaptation should be executed reactively or proactively must be carefully considered.

The answer to these questions can be informed by past experience with similar existing systems, or by prototyping and simulation activities that can provide a good estimation of the behavior of the system at run time. Nevertheless, experience with similar systems may not always be available, and prototyping or simulating (potentially many) system design variants is not cost-effective. Moreover, prototyping and simulation are good at providing estimations of the behavior of a system in the "normal" case, but do not systematically support system analysis in the context of an unpredictable environment (e.g., worst-case scenario).

In this section, we overview a technique to analyze self-adaptive systems that explicitly considers the uncertainty in their operating environment. The approach enables developers to approximate the behavioral envelope of a self-adaptive system by analyzing best- and worst-case scenarios of alternative designs for self-adaptation mechanisms, given some assumptions about the behavior of the operating environment. The formal underpinnings of the approach are based on model checking of stochastic multiplayer games (SMGs) [20], a technique appropriate to analyze the interplay between a self-adaptive system and its environment. SMG models are expressive enough to capture: (i) the inherent uncertainty and variability of the environment; and (ii) the competitive behavior between a system and its environment (reflecting the fact that environment changes cannot be controlled by the system).

The underlying idea behind the approach is modeling a system and its environment as players in a SMG, which can either cooperate to achieve a common goal (best-case scenario analysis), or compete against each other (worst-case scenario analysis). Being purely declarative, this technique does not require the availability of specific adaptation algorithms or infrastructure.

This technique is intended to endow developers with a preliminary understanding of the behavior resulting from architecting a self-adaptive system based on a set of coarse-grained design decisions, helping them to narrow down the solution space. Once the coarse-grained architecture of the system has been laid out, it can be refined into more detailed specifications that can be employed to develop prototypes or simulations that require the availability of specific adaptation algorithms and demand more effort to develop, compared to a declarative approach.

The remainder of this section first provides an overview of the technique for model checking SMGs that serves as foundation for analyzing self-adaptation. Next, we describe how model checking of SMGs is applied to compare alternative approaches to proactive adaptation in the context of Znn.com [21], a benchmark system that has been employed to assess different works on self-adaptation. Finally, we discuss the challenges in shifting the technique from design-time analysis to run-time synthesis of adaptation behavior.

7.1 Model checking stochastic games

Automatic verification techniques for probabilistic systems have been successfully applied in a variety of application domains that range from power management or wireless communication protocols, to biological systems. In particular, techniques such as probabilistic model checking provide a means to model and analyze systems that exhibit stochastic behavior, enabling reasoning quantitatively about probability and reward-based properties (e.g., about the system's use of resources, or time).

Competitive behavior may also appear in systems when some component cannot be controlled, and could behave according to different or even conflicting goals with respect to other system components. In such situations, a natural fit is modeling a system as a game between different players, adopting a game-theoretic perspective.

The approach that we describe in this section builds upon a recent technique for modeling and analyzing SMGs [20]. In this technique, systems are modeled as turn-based SMGs, meaning that in each state of the model, only one player can choose between several actions, the outcome of which can be probabilistic. Players can cooperate to achieve the same goal, or compete to achieve their own goals.

The approach includes a logic called rPATL for expressing quantitative properties of stochastic multiplayer games and reasoning about the ability of a set of players to collectively achieve a particular goal. Properties written in rPATL can state that a coalition of players has a strategy which can ensure that either the probability of an event's occurrence or an expected reward measure meets some threshold. rPATL is a CTL-style branching-time temporal logic that incorporates the coalition operator $\langle\langle C \rangle\rangle$ of ATL [1], combining it with the probabilistic operator $\mathsf{P}_{\bowtie \mathsf{q}}$ and path formulae from PCTL [12]. Moreover, rPATL includes a generalization of the reward operator $\mathsf{R}^{\mathsf{r}}_{\bowtie \mathsf{x}}$ from [32] to reason about goals related to rewards. An extended version of the rPATL reward operator $\langle\langle C \rangle\rangle \mathsf{R}^{\mathsf{r}}_{\mathsf{max}=?}[\mathsf{F}\ \phi]$ enables the quantification of the maximum accrued reward r along paths that lead to states satisfying state formula ϕ that can be guaranteed by players in coalition C, independently of the strategies followed by the other players. An example of typical usage combining the coalition and reward maximization operators is $\langle\langle \mathsf{sys} \rangle\rangle \mathsf{R}^{\mathsf{utility}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{end}]$, meaning "value of the maximum $\mathsf{utility}$ reward accumulated along paths leading to an end state that a player sys can guarantee, regardless of the strategies of other players."

Reasoning about strategies is a fundamental aspect of model checking SMGs, which enables the synthesis of a strategy that is able to optimize an objective expressed as a property including an extended version of the rPATL reward operator. An SMG strategy resolves the choices in each state, selecting actions for a player based on the current state and a set of memory elements (please refer to [20] for more details on SMG strategy synthesis).

7.2 Reasoning about self-adaptation using stochastic games

The underlying idea behind the approach to analyze self-adaptive systems consists in modeling both the self-adaptive system and its environment as two players of a SMG. The system's player objective is optimizing an objective function encoded in a rPATL specification (e.g., minimizing the probability of violating a safety property, or maximizing accrued utility - encoded as a reward structure on the game). In contrast, the environment can either be considered as adversarial to the system (enabling worst-case scenario analysis), or as a cooperative player that helps the system to optimize its objective function (enabling best-case scenario analysis).

We illustrate the technique in the context of Znn.com [21], a benchmark case study employed to assess different works on self-adaptation.

The main objective for Znn.com is to provide content to customers within a reasonable response time, while keeping the cost of the server pool within a certain operating budget. It is considered that from time to time, due to highly popular events, Znn.com experiences spikes in requests that it cannot serve adequately, even at maximum pool size. To prevent losing customers, the system can maintain functionality at a reduced level of fidelity by setting servers to return only textual content during such peak times, instead of not providing service to some of its customers. There are two main quality objectives for the self-adaptation of the system: (i) performance, which depends on request response time, server load, and network bandwidth, and (ii) cost, which depends on the number of active servers.

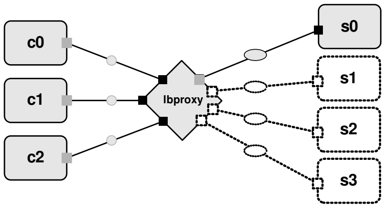


Fig. 11 Znn.com system architecture

When response time becomes too high, the system can execute adaptation tactics to: (i) increment its server pool size if it is within budget to improve performance; or (ii) switch servers to textual mode if the cost is near to budget limit. For simplicity, we consider a simple version of Znn.com that adapts only by adjusting server pool size. Note that different adaptation tactics take different amounts of time until their effects are produced (i.e., they have different *latency*). For example, adapting the system to serve results in textual mode may be achieved quickly if it can be done by changing a simple setting in a component, whereas booting up a server to share the load may take some time. When planning how to adapt, self-adaptive approaches tend to make simplifying assumptions about the properties of adaptation, such as ignoring the time it takes for an adaptation tactic to cause its intended effect.

In the following, we show how SMG analysis can be instantiated to quantify the benefits of employing adaptation latency information in decision-making, comparing latency-aware with latency-agnostic adaptation in Znn.com.

The approach consists of two phases: (i) model specification, consisting of building the game model that describes the possible interactions between the self-adaptive system and its environment, and (ii) strategy synthesis, in which a game strategy that optimizes the objective function of the system player is built, enabling developers to quantify the outcome of adaptation in boundary cases.

7.2.1 Model specification

We consider a game for Znn.com played in turns by two players who are in control of the behavior of the environment and the system, respectively:

- *Environment Player.* The environment is in control of the evolution of time and other variables of the execution context that are out of the system's control. During its turn, the environment sets the number of request arrivals for the current time period and updates the values of other environment variables (e.g., increasing the variable that keeps track of execution time).
- *System Player.* During its turn, the system can either: (i) trigger the activation of a new server, which will become effective only after the latency period of the tactic

expires; (ii) discharge a server (with no latency associated); or (iii) yield the turn to the environment player without executing any actions. In addition, the system updates during its turn the value of the response time according to the request arrivals placed by the environment player during the current time period and the number of active servers (computed using an M/M/c queuing model [22]).

The objective of the system player is maximizing the accrued utility during execution. To represent utility, we employ a reward structure that maps game states to a utility value computed according to a set of utility functions and preferences (i.e., weights). We consider two functions $\mathsf{U_R}$ and $\mathsf{U_C}$ that map the response time and the number of active servers in the system to performance and cost utility, respectively.

In latency-aware adaptation, a reward structure rIU encoded in the game employs the value of response time $MMc(s,a)$ computed according to the request arrivals a and the number of active servers s during the tactic latency period to compute the value of instantaneous utility as $\mathsf{rIU} = w_R \cdot \mathsf{U_R}(MMc(s,a)) + w_C \cdot \mathsf{U_C}(cps \cdot s)$, where cps is the cost of operating a server.

However, in non-latency-aware adaptation, the instantaneous utility expected by the algorithm during the latency period for activating a server does not match the real utility extracted for the system, since the new server has not yet impacted the performance (although there is impact on cost, since the server is already active while booting up). In this case, we add to the model a second reward structure rEIU in which the utility for performance during the latency period is based on the response time that the system would have if the new server had completed its activation: $\mathsf{rEIU} = w_R \cdot \mathsf{U_R}(MMc(s+1,a)) + w_C \cdot \mathsf{U_C}(cps \cdot s)$.

7.2.2 Strategy synthesis

To illustrate how SMG analysis can be employed to compare latency-aware with latency-agnostic adaptation, we describe rPATL specifications that enable quantifying the maximum accrued utility that adaptation can guarantee, independently of the behavior of the environment (worst-case scenario analysis).

- *Latency-Aware Adaptation.* We define the *real guaranteed accrued utility* (U_{rga}) as the maximum real instantaneous utility reward accumulated throughout execution that the system player is able to guarantee, independently of the behavior of the environment player:

  $$U_{rga} \triangleq \langle\langle \mathsf{sys} \rangle\rangle \mathsf{R}^{\mathsf{rIU}}_{\mathsf{max}=?}[\mathsf{F^c}\ \mathsf{t} = \mathsf{MAX\_TIME}]$$

  This expression quantifies the utility that an optimal self-adaptation algorithm would be able to extract from the system, given the most adverse possible conditions of the environment.
- *Latency-Agnostic Adaptation.* In latency-agnostic adaptation, real utility does not coincide with the expected utility that a self-adaptation algorithm would employ for decision-making. Hence, the analysis is performed in two steps:

  1. Compute the strategy that the adaptation algorithm would follow based on the information it employs about expected utility. That strategy is computed based

on an rPATL specification that obtains the expected guaranteed accrued utility (U_{ega}) for the system player:

$$U_{ega} \triangleq \langle\langle \mathsf{sys} \rangle\rangle \mathsf{R}^{\mathsf{rEIU}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$$

For the specification of this property we use the expected utility reward rEIU instead of the real utility reward rIU (in latency-aware adaptation $U_{ega} = U_{rga}$).

2. Verify the U_{rga} under the generated strategy. This is done by building a product of the existing game model and the strategy synthesized in the previous step, obtaining a new game under which further properties can be verified. In our case, we quantify the reward for real utility in the new game in which the system player strategy for maximizing expected utility has already been fixed.

Model checking different variants of Znn.com's SMG shows that latency-aware adaptation outperforms in all cases its latency-agnostic counterpart (more details in [18]). Concretely, latency-aware adaptation is able to guarantee an increment in utility extracted from the system, independently of the behavior of the environment (ΔU_{rga}) that increases progressively with higher tactic latencies.

7.3 From design-time analysis to runtime synthesis

Although the approach presented in this section is oriented towards obtaining a preliminary understanding of adaptation behavior at design-time, its underlying principles can also be tailored to decide how to adapt at run-time by synthesizing system strategies while considering explicitly the behavior of the environment.

The main advantage of using probabilistic model checking at run-time instead of a specific adaptation algorithm is that the adaptation decision is optimal (e.g., over a time horizon), since the model checker selects a strategy that optimizes the system's goal through exhaustive search. However, the current approach has limited scalability, since the interleaving of environment and system transitions can easily lead to an explosion of the state-space. This limitation can be mitigated by carefully choosing the level of abstraction of the model, or representing the behavior of the environment only over a relevant time horizon, for instance. Moreover, we expect that the maturation of this technology will result in the development of efficient run-time synthesis techniques, following the lead of recent advances in efficient quantitative verification at run-time [33, 31].

8 Summary

We presented two classes of techniques that support key processes associated with the operation of self-aware computing systems—verification and synthesis. Using probabilistic models ranging from Markov chains and Bayesian networks to stochastic multiplayer games, verification enables self-aware systems to reason about their reliability, performance, cost and other QoS properties. When changes in the environment or goals render the actual or predicted values of these properties inadequate, the synthesis of connectors and service compositions supports the adaptation of the software architecture of self-aware systems.

Acknowledgments

The work concerning the synthesis method described in Section 2 has been supported by the European Union's H2020 Programme under grant agreement number 644178 (project CHOReVOLUTION - Automated Synthesis of Dynamic and Secured Choreographies for the Future Internet), and by the Ministry of Economy and Finance, Cipe resolution n. 135/2012 (project INCIPICT - INnovating CIty Planning through Information and Communication Technologies).

References

1. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.
2. J. Andersson, R. de Lemos, S. Malek, and D. Weyns. Modeling dimensions of self-adaptive software systems. In *SEfSAS*, pages 27–47. 2009.
3. M. Autili, D. Di Ruscio, A. Di Salle, and A. Perucci. CHOReOSynt: Enforcing choreography realizability in the future internet. In *FSE'14*, pages 723–726, 2014.
4. M. Autili, A. Di Salle, and M. Tivoli. Synthesis of resilient choreographies. In *Software Engineering for Resilient Systems*, pages 94–108. 2013.
5. M. Autili, P. Inverardi, and M. Tivoli. Automated synthesis of service choreographies. *IEEE Software*, 32(1):50–57, 2015.
6. M. Autili, L. Mostarda, A. Navarra, and M. Tivoli. Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems. *Journal of Systems and Software*, 81(12):2210–2236, 2008.
7. M. Autili and M. Tivoli. Distributed enforcement of service choreographies. In *FOCLASA'14*, pages 18–35, 2014.
8. E. Bartocci, R. Grosu, P. Katsaros, et al. Model repair for probabilistic systems. In *TACAS'11*, pages 326–340. 2011.
9. S. Bernardi, J. Merseguer, and D. C. Petriu. *Model-Driven Dependability Assessment of Software Systems*. Springer, 2013.
10. M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. *ACM TOSEM*, 11:386–426, 2002.
11. A. Bertolino, A. Calabrò, F. Di Giandomenico, et al. On-the-fly dependable mediation between heterogeneous networked systems. In *ICSOFT'11*, pages 20–37, 2012.
12. A. Bianco and L. de Alfaro. Model checking of probabalistic and nondeterministic systems. In *FSTTCS*, pages 499–513, 1995.
13. R. Calinescu, S. Gerasimou, and A. Banks. Self-adaptive software with decentralised control loops. In *FASE'15*, pages 235–251. 2015.
14. R. Calinescu, C. Ghezzi, K. Johnson, et al. Formal verification with confidence intervals to establish quality of service properties of software systems. *IEEE Transactions on Reliability*, pages 1–16, 2015.
15. R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM*, 55(9):69–77, 2012.
16. R. Calinescu, K. Johnson, and Y. Rafiq. Developing self-verifying service-based systems. In *ASE'13*, pages 734–737, 2013.
17. R. Calinescu, Y. Rafiq, K. Johnson, and M. E. Bakir. Adaptive model learning for continual verification of non-functional properties. In *ICPE'14*, pages 87–98, 2014.

18. J. Cámara, G. A. Moreno, and D. Garlan. Stochastic game analysis and latency awareness for proactive self-adaptation. In *SEAMS'14*, pages 155–164, 2014.
19. M. Ceska, F. Dannenberg, M. Z. Kwiatkowska, and N. Paoletti. Precise parameter synthesis for stochastic biochemical systems. In *CMSB'14*, pages 86–98, 2014.
20. T. Chen, V. Forejt, M. Z. Kwiatkowska, et al. Automatic verification of competitive stochastic systems. *Formal Methods in System Design*, 43(1):61–92, 2013.
21. S. Cheng, D. Garlan, and B. R. Schmerl. Evaluating the effectiveness of the rainbow self-adaptive system. In *SEAMS'09*, pages 132–141, 2009.
22. R. Chiulli. *Quantitative Analysis: An Introduction*. Automation and production systems. 1999.
23. A. Choi, A. Darwiche, L. Zheng, and O. J. Mengshoel. A tutorial on Bayesian networks for system health management. In *Data Mining in Systems Health Management: Detection, Diagnostics, and Prognostics*. 2011.
24. V. Cortellessa, A. Di Marco, and P. Inverardi. *Model-Based Software Performance Analysis*. Springer, 2011.
25. C. Daws. Symbolic and parametric model checking of discrete-time Markov chains. In *ICTAC'04*, pages 280–294, 2004.
26. R. de Lemos, H. Giese, H. A. Müller, et al. Software engineering for self-adaptive systems: A second research roadmap. In *SEfSAS II*, pages 1–32. 2013.
27. C. Dehnert, S. Junges, N. Jansen, et al. PROPhESY: A probabilistic parameter synthesis tool. In *CAV'15*, pages 214–231, 2015.
28. A. Di Marco, P. Inverardi, and R. Spalazzese. Synthesizing self-adaptive connectors meeting functional and performance concerns. In *SEAMS'13*, pages 133–142, 2013.
29. I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *ICSE'09*, pages 111–121, 2009.
30. European Commission. Digital Agenda for Europe - Future Internet Research and Experimentation (FIRE) initiative, 2015.
31. A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *ICSE'11*, pages 341–350, 2011.
32. V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker. Automated verification techniques for probabilistic systems. In *SFM'11*, pages 53–113, 2011.
33. S. Gerasimou, R. Calinescu, and A. Banks. Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration. In *SEAMS'14*, pages 115–124, 2014.
34. S. Gerasimou, G. Tamburrelli, and R. Calinescu. Search-based synthesis of probabilistic models for quality-of-service software engineering. In *ASE'15*, pages 319 – 330, 2015.
35. M. Güdemann, G. Salaün, and M. Ouederni. Counterexample guided synthesis of monitors for realizability enforcement. In *ATVA'12*, pages 238–253. 2012.
36. E. M. Hahn, H. Hermanns, and L. Zhang. Probabilistic reachability for parametric Markov models. *Software Tools for Technology Transfer*, 13(1):3–19, 2010.
37. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1194.
38. M. C. Huebscher and J. A. McCann. A survey of autonomic computing – degrees, models, and applications. *ACM Comput. Surv.*, 40(3):1–28, 2008.
39. P. Inverardi, V. Issarny, and R. Spalazzese. A Theory of Mediators for Eternal CONNECTors. In *ISoLA'10*, pages 236–250, 2010.
40. P. Inverardi, R. Spalazzese, and M. Tivoli. Application-Layer Connector Synthesis. In *SFM'11*, pages 148–190, 2011.
41. N. Jansen, F. Corzilius, M. Volk, et al. Accelerating parametric probabilistic verification. In *QEST'11*, pages 404–420, 2014.
42. K. Johnson, R. Calinescu, and S. Kikuchi. An incremental verification framework for component-based software systems. In *CBSE'13*, pages 33–42, 2013.
43. D. Jovanovic and L. M. de Moura. Solving non-linear arithmetic. In *IJCAR*, pages 339–354, 2012.
44. J.-P. Katoen, I. S. Zapreev, E. M. Hahn, et al. The ins and outs of the probabilistic model checker MRMC. *Performance Evaluation*, 68(2):90–104, 2011.

45. D. Koller and N. Friedman. *Probabilistic Graphical Methods: Principles and Techniques*. MIT Press, 2009.
46. S. Kounev, X. Zhu, J. O. Kephart, and M. Kwiatkowska. Model-driven Algorithms and Architectures for Self-Aware Computing Systems (Dagstuhl Seminar 15041). *Dagstuhl Reports*, 5(1):164–196, 2015.
47. J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.
48. M. Kwiatkowska. Quantitative verification: models, techniques and tools. In *ESEC/FSE'07*, pages 449–458, 2007.
49. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV'11*, pages 585–591, 2011.
50. S. Malek, N. Beckman, M. Mikic-Rakic, and N. Medvidovic. A framework for ensuring and improving dependability in highly distributed systems. In *Architecting Dependable Systems III*, pages 173–193. 2004.
51. O. J. Mengshoel, M. Chavira, K. Cascio, et al. Probabilistic model-based diagnosis: An electrical power system case study. *Systems, Man and Cybernetics*, 40(5):874–885, 2010.
52. O. J. Mengshoel and J. M. Schumann. Software health management with bayesian networks. In *2nd Intl. Workshop On Software Health Management*, 2011.
53. N. Nostro, R. Spalazzese, F. Di Giandomenico, and P. Inverardi. Achieving functional and non functional interoperability through synthesized connectors. *Journal of Systems and Software*, pages 185–199, 2016.
54. J. L. Pastrana, E. Pimentel, and M. Katrib. QoS-enabled and self-adaptive connectors for web services composition and coordination. *Comput. Lang. Syst. Struct.*, 37(1):2–23, 2011.
55. S. Pathak, E. Ábrahám, N. Jansen, et al. A greedy approach for the efficient repair of stochastic models. In *NFM'15*, pages 295–309, 2015.
56. Q. Qiu, Q. Wu, and M. Pedram. Stochastic modeling of a power-managed system: construction and optimization. In *Intl. Symp. on Low Power Electronics and Design*, pages 194–199, 1999.
57. B. Ricks and O. J. Mengshoel. Diagnosis for uncertain, dynamic and hybrid domains using bayesian networks and arithmetic circuits. *Intl. Journal of Approximate Reasoning*, 55(5):1207–1234, 2014.
58. J. Schumann, T. Mbaya, and O. J. Mengshoel. Bayesian software health management for aircraft guidance, navigation, and control. In *Prognostics and Health Management Society*, 2011.
59. J. Schumann, O. J. Mengshoel, and T. Mbaya. Integrated software and sensor health management for small spacecraft. In *Intl. Conf. on Space Mission Challenges for Information Technology*, pages 77–84, 2011.
60. J. Schumann, K. Y. Rozier, T. Reinbacher, et al. Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. *Intl. Journal of Prognostics and Health Management*, 6, 2015.
61. J. Schumann, A. N. Srivastava, and O. J. Mengshoel. Who guards the guardians? toward V&V of health management software. In *RV'10*, pages 399–404, 2010.
62. R. Spalazzese and P. Inverardi. Mediating connector patterns for components interoperability. In *ECSA'10*, pages 335–343, 2010.
63. B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *ICSE'03*, pages 374–384, 2003.
64. A. Srivastava and J. Han, editors. *Data Mining in Systems Health Management: Detection, Diagnostics, and Prognostics*. Chapman and Hall/CRC Press, 2011.
65. M. Tivoli, P. Fradet, A. Girault, and G. Gößler. Adaptor synthesis for real-time components. In *TACAS'07*, pages 185–200, 2007.
66. D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19, 1997.
67. L. Zheng and O. J. Mengshoel. Exploring multiple dimensions of parallelism in junction tree message passing. In *UAI Application Workshops*, 2013.
68. L. Zheng and O. J. Mengshoel. Optimizing parallel belief propagation in junction trees using regression. In *KDD'13*, pages 757–765, 2013.