Autom Softw Eng (2018) 25:785–831
https://doi.org/10.1007/s10515-018-0235-8

Synthesis of probabilistic models for quality-of-service software engineering

Simos Gerasimou[1] · **Radu Calinescu**[1] · **Giordano Tamburrelli**[2]

Received: 20 December 2017 / Accepted: 25 April 2018 / Published online: 17 May 2018

Abstract An increasingly used method for the engineering of software systems with strict quality-of-service (QoS) requirements involves the synthesis and verification of probabilistic models for many alternative architectures and instantiations of system parameters. Using manual trial-and-error or simple heuristics for this task often produces suboptimal models, while the exhaustive synthesis of all possible models is typically intractable. The EvoChecker search-based software engineering approach presented in our paper addresses these limitations by employing evolutionary algorithms to automate the model synthesis process and to significantly improve its outcome. EvoChecker can be used to synthesise the Pareto-optimal set of probabilistic models associated with the QoS requirements of a system under design, and to support the selection of a suitable system architecture and configuration. EvoChecker can also be used at runtime, to drive the efficient reconfiguration of a self-adaptive software system. We evaluate EvoChecker on several variants of three systems from different application domains, and show its effectiveness and applicability.

Keywords Search-based software engineering · Probabilistic model checking · Evolutionary algorithms · QoS requirements

✉ Simos Gerasimou
simos.gerasimou@york.ac.uk

Radu Calinescu
radu.calinescu@york.ac.uk

Giordano Tamburrelli
tambug@gmail.com

[1] Department of Computer Science, University of York, York, UK

[2] lastminute.com, Chiasso, Switzerland

1 Introduction

Software systems used in application domains including healthcare, finance and manufacturing must comply with strict reliability, performance and other quality-of-service (QoS) requirements. The software engineers developing these systems must use rigorous techniques and processes at all stages of the software development life cycle. In this way, the engineers can continually assess the correctness of a system under development (SUD) and confirm its compliance with the required levels of reliability and performance.

Probabilistic model checking (PMC) is a formal verification technique that can assist in establishing the compliance of a SUD with QoS requirements through mathematical reasoning and rigorous analysis (Baier and Katoen 2008; Clarke et al. 1999). PMC supports the analysis of reliability, timeliness, performance and other QoS requirements of systems exhibiting stochastic behaviour, e.g. due to unreliable components or uncertainties in the environment (Kwiatkowska 2007). The technique has been successfully applied to the engineering of software for critical systems (Alur et al. 2015; Woodcock et al. 2009). In PMC, the behaviour of a SUD is defined formally as a finite state-transition model whose transitions are annotated with information about the likelihood or timing of events taking place. Examples of probabilistic models that PMC operates with include discrete and continuous-time Markov chains, and Markov decision processes (Kwiatkowska et al. 2007). QoS requirements are expressed formally using probabilistic variants of temporal logic, e.g., probabilistic computation tree logic and continuous stochastic logic (Kwiatkowska et al. 2007). Through automated exhaustive analysis of the underlying low-level model, PMC proves or disproves compliance of the probabilistic model of the system with the formally specified QoS requirements.

Recent advances in PMC reinforced its applicability to the cost-effective engineering of software both at design time (Baier et al. 2010; Calinescu et al. 2016; Kwiatkowska et al. 2010) and at runtime (Calinescu et al. 2012; Draeger et al. 2014). The design-time use of the technique involves the verification of alternative designs of a SUD. The objectives are to identify designs whose quality attributes comply with system QoS requirements and also to eliminate early in the design process errors that could be hugely expensive to fix later (Damm and Lundberg 2007). Designs that meet these objectives can then be used as a basis for the implementation of the system. Alternatively, software engineers can construct probabilistic models of existing systems and employ PMC to assess their QoS attributes. Within the last decade, PMC has also been used to drive the reconfiguration of self-adaptive systems (Baresi and Ghezzi 2010; Calinescu and Kwiatkowska 2009; Epifani et al. 2009) by supporting the "analyse" and "plan" stages of the monitor-analyse-plan-execute control loop (Calinescu et al. 2017d; Salehie and Tahvildari 2009) of these systems. In this runtime use, PMC provides formal guarantees that the reconfiguration plan adopted by the self-adaptive system meets the QoS requirements (Calinescu et al. 2017a, 2012). We discuss related research on using PMC at runtime, including our recent work from Calinescu et al. (2015), Gerasimou et al. (2014), in Sect. 8.

Notwithstanding the successful applications of PMC at both design time and runtime, the synthesis and verification of probabilistic models that satisfy the QoS requirements of a system remains a very challenging task. The complexity of this task increases significantly when the search space is large and/or the QoS requirements ask for the optimisation of conflicting QoS attributes of the system (e.g. the maximisation of reliability and minimisation of cost). Existing approaches such as exhaustive search and simple heuristics like manual trial-and-error and automated hill climbing can only tackle this challenge for small systems. Exhaustively searching the solution space for an optimal probabilistic model is intractable for most real-world systems. On the other hand, trial-and-error requires manual verification of numerous alternative instantiations of the system parameters, while simple heuristics do not generalise well and are often biased towards a particular area of the problem landscape (e.g. through getting stuck at local optima).

The EvoChecker search-based software engineering approach presented in our paper addresses these limitations of existing approaches by automating the synthesis of probabilistic models and by considerably improving the outcome of the synthesis process. EvoChecker achieves these improvements by using evolutionary algorithms (EAs) to guide the search towards areas of the search space more likely to comprise probabilistic models that meet a predefined set of QoS requirements. These requirements can include both *constraints*, which specify bounds for QoS attributes of the system (e.g. “Workflow executions must complete successfully with probability at least 0.98”), and *optimisation objectives* (e.g. “The workflow response time should be minimised”).

Given this set of QoS requirements and a *probabilistic model template* that encodes the configuration parameters (e.g., alternative architectures, parameter ranges) of the software system, EvoChecker supports both the identification of suitable architectures and configurations for a software system under design, and the runtime reconfiguration of a self-adaptive software system.

When used at design time, EvoChecker employs multi-objective EAs to synthesise (i) a set of probabilistic models that closely approximates the Pareto-optimal model set associated with the QoS requirements of a software system; and (ii) the corresponding approximate Pareto front of QoS attribute values. Given this information, software designers can inspect the generated solutions to assess the tradeoffs between multiple QoS requirements and make informed decisions about the architecture and parameters of the SUD.

When used at runtime, EvoChecker drives the reconfiguration of a self-adaptive software system by synthesising probabilistic models that correspond to configurations which meet the QoS requirements of the system. To speed up this runtime search, we use *incremental probabilistic model synthesis*. This novel technique involves maintaining an *archive* of specific probabilistic models synthesised during recent reconfiguration steps, and using these models as the starting point for each new search. As reconfiguration steps are triggered by events such as changes to the environment that the system operates in, the EvoChecker archive accumulates “solutions” to past events that often resemble new events experienced by the system. Therefore, starting new searches from the archived “solutions” can achieve significant performance improvement for the model synthesis process.

The main contributions of our paper are:

- The EvoChecker approach for the search-based synthesis of probabilistic models for QoS software engineering, and its application to the synthesis of models that meet QoS requirements both at design time and at runtime.
- The EvoChecker high-level modelling language, which extends the modelling language used by established probabilistic model checkers such as PRISM (Kwiatkowska et al. 2011) and Storm (Dehnert et al. 2017).
- The definition of the probabilistic model synthesis problem.
- An incremental probabilistic model synthesis technique for the efficient runtime generation of probabilistic models that satisfy the QoS requirements of a self-adaptive system.
- An extensive evaluation of EvoChecker in three case studies drawn from different application domains.
- A prototype open-source EvoChecker tool and a repository of case studies, both of which are freely available from our project webpage at http://www-users.cs.york.ac.uk/simos/EvoChecker.

These contributions significantly extend the preliminary results from our conference paper on search-based synthesis of probabilistic models (Gerasimou et al. 2015) in several ways. First, we introduce an incremental probabilistic model synthesis technique that extends the applicability of EvoChecker to self-adaptive software systems. Second, we devise and evaluate different strategies for selecting the archived solutions used by successive EvoChecker synthesis tasks at runtime. Third, we extend the presentation of the EvoChecker approach with additional technical details and examples. Fourth, we use EvoChecker to develop two self-adaptive systems from different application domains (service-based systems and unmanned underwater vehicles). Finally, we use the systems and models from our experiments to assemble a repository of case studies available on our project website.

The rest of the paper is organised as follows. Section 2 presents the software system used as a running example. Section 3 introduces the EvoChecker modelling language, and Sect. 4 presents the specification of EvoChecker constraints and optimisation objectives using the QoS requirements of a software system. Section 5 describes the use of EvoChecker to synthesise probabilistic models at design time and at runtime. Section 6 summarises the implementation of the EvoChecker prototype tool. Section 7 presents the empirical evaluation carried out to assess the effectiveness of EvoChecker, and an analysis of our findings. Finally, Sects. 8 and 9 discuss related work and conclude the paper, respectively.

2 Running example

We will illustrate the EvoChecker approach for synthesising probabilistic models using a real-world service-based system from the domain of foreign exchange trading. The system, which for confidentiality reasons we anonymise as FX, is used by a European foreign exchange brokerage company and implements the workflow in Fig. 1.

An FX trader can use the system to carry out trades in *expert* or *normal* mode. In the *expert* mode, the trader can provide her objectives or action strategy. FX periodically analyses exchange rates and other market activity, and automatically executes

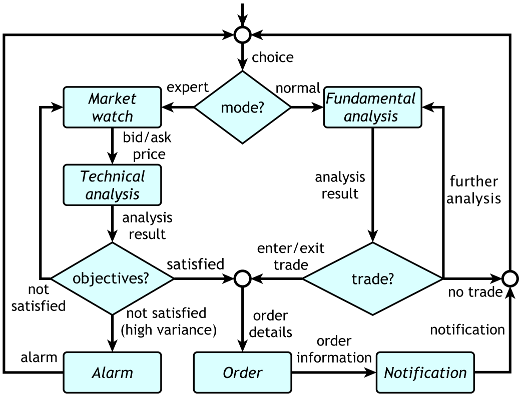

Fig. 1 Workflow of the FX system. (modified with the permission from Gerasimou et al. (2015))

Table 1 QoS requirements for the FX system

ID	Informal description
R1	"Workflow executions must complete successfully with probability at least 98%"
R2	"The total service response time per workflow execution should be minimised"
R3	"The probability of a service failure during a workflow execution should be minimised"
R4	"The total cost of the third-party services used by a workflow execution should be minimised"

a trade once the trader's objectives are satisfied. In particular, a *Market watch* service retrieves real-time exchange rates of selected currency pairs. A *Technical analysis* service receives this data, identifies patterns of interest and predicts future variation in exchange rates. Based on this prediction and if the trader's objectives are *"satisfied"*, an *Order* service is invoked to carry out a trade; if they are "unsatisfied", execution control returns to the *Market watch* service; and if they are "unsatisfied with high variance", an *Alarm* service is invoked to notify the trader about opportunities not captured by the trading objectives. In the *normal* mode, FX assesses the economic outlook of a country using a *Fundamental analysis* service that collects, analyses and evaluates information such as news reports, economic data and political events, and provides an assessment of the country's currency. If the trader is satisfied with this assessment, she can sell/buy currency by invoking the *Order* service, which in turn triggers a *Notification* service to confirm the successful completion of a trade.

The FX system uses $m_i \geq 1$ functionally equivalent implementations of the ith service. For any service i, the jth implementation, $1 \leq j \leq m_i$ is characterised by its reliability $r_{ij} \in [0, 1]$ (i.e., probability of successful invocation), invocation cost $c_{ij} \in \mathbb{R}^+$ and response time $t_{ij} \in \mathbb{R}^+$.

FX is required to satisfy the QoS requirements from Table 1. For each service, FX must select one of two invocation strategies by means of a configuration parameter $str_i \in$ {PROB, SEQ}, where

- if str_i = PROB, FX uses a *probabilistic* strategy to randomly select one of the service implementations based on an FX-specified discrete probability distribution $p_{i1}, p_{i2}, \ldots, p_{im_i}$; and
- if str_i = SEQ, FX uses a *sequential* strategy that employs an execution order to invoke one after the other all *enabled* service implementations until a successful response is obtained or all invocations fail.

For the SEQ strategy, a parameter $ex_i \in \{1, 2, \ldots, m_i!\}$ establishes which of the $m_i!$ permutations of the m_i implementations should be used, and a configuration parameter $x_{ij} \in \{0, 1\}$ indicates if implementation j is enabled ($x_{ij} = 1$) or not ($x_{ij} = 0$).

3 EvoChecker modelling language

EvoChecker uses an extension of the modelling language that leading model checkers such as PRISM (Kwiatkowska et al. 2011) and Storm (Dehnert et al. 2017) use to define probabilistic models. This language is based on the Reactive Modules formalism (Alur and Henzinger 1999), which models a system as the parallel composition of a set of *modules*. The state of a *module* is defined by a set of finite-range local variables, and its state transitions are specified by probabilistic guarded commands that modify these variables:

$$[action]\ guard \ \text{−>}\ e_1 : update_1 + \cdots + e_n : update_n; \qquad (1)$$

where *guard* is a boolean expression over all model variables. If the *guard* is *true*, the arithmetic expression e_i, $1 \leq i \leq n$, gives the probability (for discrete-time models) or rate (for continuous-time models) with which the $update_i$ change of the module variables occur. The *action* is an optional element of type 'string'; when used, all modules comprising commands with the same *action* must synchronise by performing one of these commands simultaneously. For a complete description of the modelling language, we refer the reader to the PRISM manual at www.prismmodelchecker.org/manual.

EvoChecker extends this language with the following three constructs that support the specification of the possible configurations of a system.

1. Evolvable parameters EvoChecker uses the syntax

$$\begin{array}{l}\textsf{evolve int}\ param\ [min..max];\\ \textsf{evolve double}\ param\ [min..max];\end{array} \qquad (2)$$

to define model parameters of type 'int' and 'double', respectively, and acceptable ranges for them. These parameters can be used in any field of command (1) other than $action$.

2. Evolvable probability distributions The syntax

$$\textsf{evolve distribution}\ dist\ [min_1..max_1] \ldots [min_n..max_n]; \qquad (3)$$

where $[min_i, max_i] \subseteq [0, 1]$ for all $1 \leq i \leq n$, is used to define an n-element discrete probability distribution, and ranges for the n probabilities of the distribution. The name of the distribution with $1, 2, \ldots, n$ appended as a suffix (i.e., $dist_1$, $dist_2$, etc.) can then be used instead of expressions e_1, e_2, …, e_n from an n-element command (1).

3. Evolvable modules EvoChecker uses the syntax

$$\begin{array}{l}\textsf{evolve module}\ \mathit{modName}\ \mathit{implementation}_1\ \textsf{endmodule}\\ \ldots\\ \textsf{evolve module}\ \mathit{modName}\ \mathit{implementation}_n\ \textsf{endmodule}\end{array} \quad (4)$$

to define $n \geq 2$ alternative implementations of a module *modName*.

The interpretation of the three EvoChecker constructs within a probabilistic model template is described by the following definitions.

Definition 1 (*Probabilistic model template*) A valid probabilistic model augmented with EvoChecker evolvable parameters (2), probability distributions (3) and modules (4) is called a *probabilistic model template*.

Definition 2 (*Valid probabilistic model*) A probabilistic model M is an *instance* of a probabilistic model template $\mathcal{T}$ if and only if it can be obtained from $\mathcal{T}$ using the following transformations:

- Each evolvable parameter (2) is replaced by a ‘const int *param* = *val;*’ or ‘const double *param* = *val;*’ declaration (depending on the type of the parameter), where $val \in \{min, \ldots, max\}$ or $val \in [min..max]$, respectively.
- Each evolvable probability distribution (3) is removed, and the n occurrences of its name instead of expressions e_1, …, e_n of a command (1) are replaced with values from the ranges $[min_1..max_1]$, …, $[min_n..max_n]$, respectively. For a discrete-time model, the sum of the n values is 1.0.
- Each set of evolvable modules with the same name is replaced with a single element from the set, from which the keyword ‘evolve’ was removed.

As the EvoChecker modelling language is based on the modelling language of established probabilistic model checkers such as PRISM and Storm, our approach can handle templates of all types of probabilistic models supported by these model checkers. Table 2 shows these types of probabilistic models, and the probabilistic temporal logics available to specify the QoS requirements of the modelled software systems.

Example 1 Figure 2 presents the discrete-time Markov chain (DTMC) probabilistic model template of the FX system introduced in Sect. 2. The template comprises a WorkflowFX module modelling the FX workflow, and two modules modelling the alternative implementations of each service. These two service modules correspond to the probabilistic invocation strategy and the sequential invocation strategy, respectively. Due to space constrains, Fig. 2 shows in full only the MarketWatch module for the probabilistic strategy of the *Market watch* service; the complete FX probabilistic model template is available on our project webpage.

The local variable state from the WorkflowFX module (line 5 in Fig. 2) encodes the state of the system, i.e. the service being invoked, the success or failure of that

Table 2 Types of probabilistic models supported by EvoChecker

Type of probabilistic model	QoS requirement specification logic
Discrete-time Markov chains	PCTL[a], LTL[b], PCTL*[c]
Continuous-time Markov chains	CSL[d]
Markov decision processes	PCTL[a], LTL[b], PCTL*[c]
Probabilistic automata	PCTL[a], LTL[b], PCTL*[c]
Probabilistic timed automata	PCTL[a]

[a]Probabilistic Computation Tree Logic (Bianco and Alfaro 1995; Hansson and Jonsson 1994), [b]Linear Temporal Logic (Pnueli 1985) , [c]PCTL* is a superset of PCTL and LTL, [d]Continuous Stochastic Logic (Aziz et al. 2000; Baier et al. 1999)

service invocation, etc. The local variable mw from the MarketWatch implementations (line 39) records the internal state of the *Market watch* service invocation.The WorkflowFX module synchronises with the service modules through 'start'-, 'failed'- and 'succ'-prefixed actions, which are associated with the invocation, failed execution, and successful execution of a service, respectively. For instance, the synchronisation with the MarketWatch module occurs through the actions startMW, failedMW and succMW (lines 9–11).

Each service module models a specific invocation strategy, e.g. a probabilistic selection is made between three available *Market watch* service implementations in line 41 of the first MarketWatch module. Then, the selected service implementation is invoked (lines 43–45) and either completes successfully (line 47) or fails (line 48). The FX system continues with the rest of its workflow (lines 12–31 from WorkflowFX) if the service executed successfully, or terminates (line 32) otherwise.

All three EvoChecker constructs (2)–(4) are used by the FX probabilistic model template:

- four evolvable parameters specify the enabled *Market watch* service implementations and their execution order (lines 50–53) associated with the sequential invocation strategy;
- an evolvable distribution specifies the discrete probability distribution for the probabilistic invocation strategy of the first MarketWatch module (line 36);
- two alternative implementations of the MarketWatch module are provided in lines 37–49 and 54–57, respectively.

4 EvoChecker specification of QoS requirements

4.1 Quality-of-service attributes

Given the probabilistic model template $\mathcal{T}$ of a system, QoS attributes such as the response time, throughput and reliability of the system can be expressed in the probabilistic temporal logics from Table 2, and can be evaluated by applying probabilistic model checking to relevant instances of $\mathcal{T}$. Formally, given the probabilistic temporal logic formula Φ for a QoS attribute *attr* and an instance M of $\mathcal{T}$ (i.e. a probabilistic

```
dtmc

const double pExpert=0.66; const double pSat = 0.61; const double pNotSat = 0.28;
const double pTrade = 0.27; const double pRetry = 0.20;

module WorkflowFX
  state: [0..15] init 0; // FX state

  // Start: expert or normal mode
  [fxStart] state=0 −> pExpert :(state'=1) + (1-pExpert):(state'=9);

  //Service #1: Market Watch
  [startMW] state=1 −> 1.0:(state'=2);
  [failedMW] state=2 −> 1.0:(state'=5);
  [succMW] state=2 −> 1.0:(state'=3);

  //Service #2: Technical Analysis
  [startTA] state=3 −> 1.0:(state'=4);
  [failedTA] state=4 −> 1.0:(state'=5);
  [succTA] state=4 −> pSat : (state'=1) + pNotSat : (state'=11) + (1-pSat-pNotSat) : (state'=7);

  //Service #3: Alarm
  [startAL] state=7 −> 1.0:(state'=8);
  [failedAL] state=8 −> 1.0:(state'=5);
  [succAL] state=8 −> 1.0:(state'=13);

  //Service #4: Fundamental Analysis
  [startFA] state=9 −> 1.0:(state'=10);
  [failedFA] state=10 −> 1.0:(state'=5);
  [succFA] state=10 −> pTrade : (state'=11) + pRetry : (state'=9) + (1-pTrade-pRetry) : (state'=0);

  //Service #5: Order
  [startOR] state=11 −> 1.0:(state'=12);
  [failedOR] state=12 −> 1.0:(state'=5);
  [succOR] state=12 −> 1.0:(state'=13);

  //Service #6: Notification
  [startNOT] state=13 −> 1.0:(state'=4);
  [failedNOT] state=14 −> 1.0:(state'=5);
  [succNOT] state=14 −> 1.0:(state'=15);

  [failedFX] state=5 −> 1.0:(state'=5);
  [succFX] state=15 −> 1.0:(state'=15);
endmodule

const double r11 = 0.998; const double r12 = 0.995; const double r13 = 0.996;

evolve distribution p1[0.1..0.3][0.3..0.5][0.2..0.6];

// Probabilistic invocation strategy for Service #1: Market Watch
evolve module MarketWatch
  mw: [0..5] init 0; // MW state
  // Probabilistic service selection
  [startMW] mw=0 −> p11 : (mw'=1) + p12 : (mw'=2) + p13 : (mw'=3);

  // Run services
  [runMW1] mw=1 −> r11 : (mw'=4) + (1-r11): (mw'=5);
  [runMW2] mw=2 −> r12 : (mw'=4) + (1-r12): (mw'=5);
  [runMW3] mw=3 −> r13 : (mw'=4) + (1-r13): (mw'=5);

  // End Market Watch service
  [succMW] mw=4 −> 1.0:(mw'=0);
  [failedMW] mw=5 −> 1.0:(mw'=0);
endmodule

evolve int x11[0..1];
evolve int x12[0..1];
evolve int x13[0..1];
evolve int ex1[1..6];

// Sequential invocation strategy for Service #1: Market Watch
evolve module MarketWatch
  ...
endmodule
...
```

Fig. 2 DTMC probabilistic model template for the FX system

Table 3 QoS attributes for the FX system

QoS attribute	Informal description	Formula Φ_i
$attr_1$	Workflow reliability	$P_{=?}[F\ state = 15]$
$attr_2$	Workflow response time	$R^{\mathsf{time}}_{=?}[F\ state = 15 \vee state = 5]$
$attr_3$	Workflow invocation cost	$R^{\mathsf{invocationCost}}_{=?}[F\ state = 15 \vee state = 5]$

model corresponding to a system configuration being examined), the value of the QoS attribute is

$$attr = PMC(M, \Phi), \tag{5}$$

where *PMC* is the probabilistic model checking "function" implemented by tools such as PRISM and Storm.

Example 2 The QoS requirements of the FX system from our running example (shown in Table 1) are based on three QoS attributes. Requirements **R1** and **R3** refer to the probability of successful completion (i.e. the reliability) of FX workflow executions. This QoS attribute corresponds to the probabilistic computation tree logic (PCTL) formula $P_{=?}[F\ state = 15]$ from the first row of Table 3. This PCTL formula expresses the probability that the probabilistic model template from Fig. 2 reaches its success state.

The QoS attributes for the other two requirements can be specified using *rewards* PCTL formulae (Andova et al. 2004; Katoen et al. 2005; Kwiatkowska 2007). For this purpose, positive values are associated with specific states and transitions of the model template from Fig. 2 by adding the following two **rewards ...endrewards** structures to the template:

```
rewards "time"
  [runMW1] true : $t_{11}$;
  [runMW2] true : $t_{12}$;
  [runMW3] true : $t_{13}$;
  ...
endrewards
```

```
rewards "invocationCost"
  [runMW1] true : $c_{11}$;
  [runMW1] true : $c_{12}$;
  [runMW1] true : $c_{13}$;
  ...
endrewards
```

These structures support the computation of the total service response time for requirement **R2** and of the workflow invocation cost for requirement **R4**. To this end, the two structures associate the mean response time t_{ij} and the invocation cost c_{ij} of the jth implementation of FX service i with the transition that models the execution of this service implementation. The corresponding PCTL formulae, shown in the last two rows of Table 3, represent the reward (i.e. the response time and cost, respectively) "accumulated" before reaching a state where the workflow execution terminates. In these formulae, $state = 15$ denotes a successful termination, and $state = 5$ an unsuccessful one.

Before describing the formalisation of QoS requirements in EvoChecker, we note that a software system has two types of parameters:

1. *Configuration parameters,* which software engineers can modify to select between alternative system architectures and configurations. The EvoChecker constructs (2)–(4) are used to define these parameters and their acceptable values. The set of all possible combinations of configuration parameter values forms the *configuration space Cfg* of the system.
2. *Environment parameters*, which specify relevant characteristics of the environment in which the system will operate or is operating. These parameters cannot be modified, and need to be estimated based on domain knowledge or observations of the actual system. The set of all possible combinations of environment parameter values forms the *environment space Env* of the system.

A probabilistic model template $\mathcal{T}$ of a system with configuration space *Cfg* and environment space *Env* corresponds to a specific combination of environment parameter values $e \in Env$ and to the entire configuration space *Cfg*. Furthermore, each instance M of $\mathcal{T}$ is associated with the same environment state e and with a specific combination of configuration parameter values $c \in Cfg$. We will use the notation $M(e, c)$ to refer to this specific instance of $\mathcal{T}$, and the notation $attr(e, c)$ for the value of a QoS attribute (5) computed for this instance.

Example 3 The environment parameters of the FX system comprise:

- the probabilities *pExpert*, *pSat*, *pNotSat*, *pTrade* and *pRetry* from module WorkflowFX in Fig. 2;
- the success probabilities r_{ij}, response times t_{ij} and costs c_{ij} of the FX service implementations.

The FX configuration parameters defined by the EvoChecker constructs from Fig. 2 are:

- the invocation strategies str_i used for the ith FX service;
- the probabilities p_{ij} of invoking the jth implementation of service i when the probabilistic invocation strategy is used;
- the x_{ij} and ex_i parameters specifying which implementations of service i are used by the sequential invocation strategy and their execution order.

4.2 Quality-of-service requirements

EvoChecker supports the engineering of software systems that need to satisfy two types of QoS requirements:

1. *Constraints*, i.e. requirements that specify bounds for the acceptable values of QoS attributes such as response time, throughput, reliability and cost.
2. *Optimisation objectives*, i.e. requirements which specify QoS attributes that should be minimised or maximised.

Formally, EvoChecker considers systems with $n_1 \geq 0$ constraints $R_1^C, R_2^C, \ldots, R_{n_1}^C$, and $n_2 \geq 1$ optimisation objectives $R_1^O, R_2^O, \ldots, R_{n_2}^O$. The ith constraint, R_i^C, has the form

$$attr_i \bowtie_i bound_i \tag{6}$$

Table 4 Formal specification of QoS requirements for the FX system

ID	Formal description	Informal description	Requirement type
R1	$attr_1 \geq 0.98$	Workflow reliability greater than 98%	Constraint
R2	$\mathsf{minimise} \quad attr_2$	Minimise workflow reponse time	Optimisation objective
R3	$\mathsf{minimise} \quad 1 - attr_1$	Minimise worklfow reliability	Optimisation objective
R4	$\mathsf{minimise} \quad attr_3$	Minimise workflow invocation cost	Optimisation objective

and, assuming that all optimisation objectives require the minimisation of QoS attributes,[1] the jth optimisation objective, R^O_j, has the form

$$\mathsf{minimise}\ attr_{n_1+j}, \tag{7}$$

where $\bowtie_i \in \{<, \leq, \geq, >, =\}$ is a relational operator, $bound_i \in \mathbb{R}$, $0 \leq i \leq n_1$, $1 \leq j \leq n_2$, and $attr_1, attr_2, \ldots, attr_{n_1+n_2}$ represent $n_1 + n_2$ QoS attributes (5) of the software system.

Example 4 The QoS requirements of the FX system (Table 1) comprise one constraint (**R1**) and three optimisation objectives (**R2**–**R4**). Table 4 shows the formalisation of these requirements for the design time use of EvoChecker using the FX QoS attributes from Table 3.

5 EvoChecker probabilistic model synthesis

EvoChecker supports both the selection of a suitable architecture and configuration for a software system under design, and the runtime reconfiguration of a self-adaptive software system. There are three key differences between these two uses of EvoChecker .

First, the use of EvoChecker during system design requires the specification of the (fixed) environment state that the system will operate in by a domain expert, while for its runtime use the environment state is continually updated based on monitoring information.

Second, the EvoChecker use at design time can handle multiple optimisation objectives and yields multiple Pareto-optimal solutions (i.e. probabilistic models). In contrast, EvoChecker at runtime yields a single solution (as synthesising multiple solutions is not useful without a software engineer to examine them) by operating with a single optimisation objective (i.e. a "loss" function).

Finally, the use of EvoChecker for the design of a system is a one-off activity, whereas for a self-adaptive system the approach is used to select new system configurations on frequent intervals or after each environment change. The latter use involves the *incremental* synthesis of probabilistic models by generating each new configuration efficiently based on previously synthesised ones.

[1] This assumption simplifies the presentation of EvoChecker without loss of generality.

Given these differences between the design time and runtime EvoChecker uses, we present them separately in Sects. 5.1 and 5.2, respectively.

5.1 Using EvoChecker at design time

5.1.1 Probabilistic model synthesis problem

Consider a SUD with environment space *Env*, an environment state $e \in Env$ provided by a domain expert, and a probabilistic model template $\mathcal{T}$ associated with the configuration space *Cfg* of the system. Given $n_1 \geq 0$ constraints (6) and $n_2 \geq 1$ optimisation objectives (7), the *probabilistic model synthesis problem* involves finding the *Pareto-optimal set PS* of configurations from *Cfg* that satisfy the n_1 constraints and are *non-dominated* with respect to the n_2 optimisation objectives:

$$PS = \Big\{c \in Cfg \mid \nexists\, c' \in Cfg \bullet (\forall 0 \leq i \leq n_1 \bullet attr_i(e,c) \bowtie_i bound_i \wedge \\ attr_i(e,c') \bowtie_i bound_i\Big) \wedge c' \prec c\Big\} \tag{8}$$

with the *dominance relation* $\prec : Cfg \times Cfg \rightarrow \mathbb{B}$ defined by

$$\forall c, c' \in Cfg \bullet c \prec c' \equiv \quad \forall\, n_1 + 1 \leq i \leq n_1 + n_2 \bullet attr_i(e,c) \leq attr_i(e,c') \;\wedge \\ \exists\, n_1 + 1 \leq i \leq n_1 + n_2 \bullet attr_i(e,c) < attr_i(e,c').$$

Finally, given the Pareto-optimal set *PS*, the *Pareto front* PF is defined by

$$PF = \{(a_{n_1+1}, a_{n_1+2}, \ldots, a_{n_1+n_2}) \in \mathbb{R}^{n_2} \mid \\ \exists c \in PS \bullet \forall n_1 + 1 \leq i \leq n_1 + n_2 \bullet a_i = attr_i(e,c)\}, \tag{9}$$

because the system designers need this information in order to choose between the configurations from the set PS.

5.1.2 Probabilistic model synthesis approach

Obtaining the Pareto-optimal set of a SUD, given by Eq. (8), is usually unfeasible, as the configuration space *Cfg* is typically extremely large or (when the probabilistic model template $\mathcal{T}$ includes evolvable distributions or evolvable parameters of type double) infinite. Therefore, EvoChecker synthesises a close approximation of the Pareto-optimal set by using standard *multiobjective evolutionary algorithms* such as the genetic algorithms NSGA-II (Deb et al. 2002), SPEA2 (Zitzler et al. 2001) and MOCell (Nebro et al. 2009).

Evolutionary algorithms (EAs) encode each possible solution of a search problem as a sequence of *genes*, i.e. binary representations of the problem variables. For EvoChecker, each use of an ‘evolvable’ construct (2)–(4) within the probabilistic model template $\mathcal{T}$ contributes to this sequence with the gene(s) specified by the encoding rules in Table 5. EvoChecker uses these rules to obtain the value ranges

Table 5 EvoChecker gene encoding rules

Evolvable feature of the probabilistic model template	EvoChecker gene(s)		
	Type	Cardinality	Value range V_i
evolve int $param[min..max]$;	int	1	$\{min,\ldots, max\}$
evolve double $param[min..max]$;	double	1	$[min..max]$
evolve distribution$dist[min_1..max_1]\ldots$ $\ldots[min_n..max_n]$;	double	n	$[min_1..max_1]\ \ldots[min_n..max_n]$

evolve module $mod\ implementation_1$
endmodule…
evolve module $modimplementation_m$
endmodule
	int	1	$\{1, 2, \ldots, m\}$

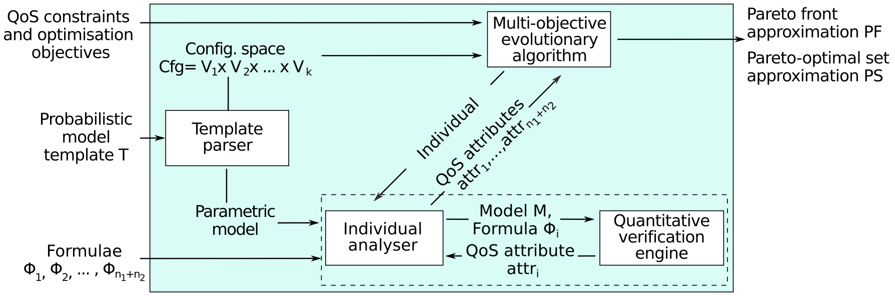
QoS constraints and optimisation objectives
Config. space Cfg= V1x V2x ... x Vk
Multi-objective evolutionary algorithm
Pareto front approximation PF
Pareto-optimal set approximation PS
Probabilistic model template T
Template parser
Individual
QoS attributes attr1,...,attrn1+n2
Parametric model
Individual analyser
Model M, Formula Φi
QoS attribute attri
Quantitative verification engine
Formulae Φ1, Φ2, ... , Φn1+n2

Fig. 3 High-level EvoChecker architecture

$V_1, V_2, \ldots, V_k$ for the $k \geq 1$ genes of $\mathcal{T}$, and to assemble the SUD configuration space $Cfg = V_1 \times V_2 \times \cdots \times V_k$.

The high-level architecture of EvoChecker is shown in Fig. 3. The probabilistic model template $\mathcal{T}$ of the SUD is processed by a *Template parser* component. The *Template parser* converts the template into an internal representation (i.e. a parametric probabilistic model) and extracts the configuration space *Cfg* as described above. The configuration space *Cfg* and the n_1 QoS constraints and n_2 optimisation objectives of the SUD are used to initialise the *Multi-objective evolutionary algorithm* component at the core of EvoChecker. This component creates a random initial *population* of *individuals* (i.e. a set of random gene sequences corresponding to different *Cfg* elements), and then iteratively evolves this population into populations containing “fitter” individuals by using the standard EA approach summarised next.

The EA approach involves evaluating different individuals (i.e., potential new system configurations) through invoking an *Individual analyser*. This component combines an individual and the parametric model created by the *Template parser* to produce a probabilistic model M in which all configuration parameters are fixed using values from the genes of the analysed individual. Next, the *Individual analyser* invokes a *Quantitative verification engine* that uses probabilistic model checking to determine the QoS attributes $attr_i$, $1 \leq i \leq n_1 + n_2$, of the analysed individual. To this end, the *Quantitative verification engine* analyses the model M and each of the probabilistic temporal logic formulae Φ_1, Φ_2, …, $\Phi_{n_1+n_2}$ corresponding to the $n_1 + n_2$ QoS attributes. These attributes are then used by the *Multi-objective evolutionary algorithm* to establish whether the individual satisfies the n_1 QoS constraints and to assess its "fitness" based on the QoS attribute values associated with the n_2 QoS optimisation objectives.

Once all individuals have been evaluated, the *Multi-objective evolutionary algorithm* performs an *assignment*, *reproduction* and *selection* step. During *assignment*, the algorithm establishes the fitness of each individual (e.g., its rank in the population). Fit individuals have higher probability to enter a "mating" pool and to be chosen for reproduction and selection. With *reproduction*, the algorithm creates new individuals from the mating pool by means of *crossover* and *mutation*. *Crossover* randomly selects two fit individuals and exchanges genes between them to produce offspring with potentially higher fitness values. *Mutation*, on the other hand, introduces variation in the population by selecting an individual from the pool and creating an offspring by randomly changing a subset of its genes. Finally, through *selection*, a subset of the individuals from the current population and offspring becomes the new population that will evolve in the next generation.

The *Multi-objective evolutionary algorithm* uses *elitism*, a strategy that directly propagates into the next population a subset of the fittest individuals from the current population. This strategy ensures the iterative improvement of the Pareto-optimal approximation set PS assembled by EvoChecker. Furthermore, the multi-objective EAs used by EvoChecker maintain diversity in the population and generate a Pareto-optimal approximation set spread as uniformly as possible across the search space. This uniform spread is achieved using algorithm-specific mechanisms for diversity preservation. One such mechanism involves combining the *nondomination level* of each evaluated individual and the population density in its area of the search space.[2]

The evolution of fitter individuals continues until one of the following termination criteria is met:

1. the allocated computation time is exhausted;
2. the maximum number of individual evaluations has been reached;
3. no improvement in the quality of the best individuals has been detected over a predetermined number of successive iterations.

Once the evolution terminates, the set of nondominated individuals from the final population is returned as the Pareto-optimal set approximation *PS*. The values of the QoS attributes associated with the n_2 optimisation objectives and with each individual from *PS* are used to assemble the Pareto front approximation *PF*. System designers can analyse the *PS* and *PF* sets to select the design to use for system implementation.

[2] For example, NSGA-II (Deb et al. 2002) associates a nondominance level of 1 to all nondominated individuals of a population, a level of 2 to the individuals that are not dominated when level-1 individuals are ignored etc. Individuals not satisfying problem constraints receive a default level of ∞. SPEA2 (Zitzler et al. 2001) evaluates population density as the inverse of the distance to the kth nearest neighbour of the individual.

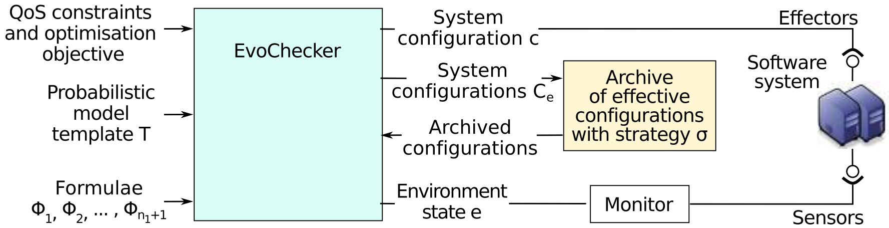
QoS constraints and optimisation objective
Probabilistic model template T
Formulae Φ1, Φ2, ... , Φn1+1
EvoChecker
System configuration c
System configurations Ce
Archived configurations
Environment state e
Archive of effective configurations with strategy σ
Effectors
Software system
Monitor
Sensors

Fig. 4 EvoChecker-driven reconfiguration of a self-adaptive software system

5.2 Using EvoChecker at runtime

5.2.1 EvoChecker-based self-adaptive systems

The use of EvoChecker to drive the runtime reconfiguration of self-adaptive software systems is illustrated in Fig. 4. The approach uses system *Sensors* to continually monitor the system and identify the parameters of the environment it operates in. Changes in the environment state *e* lead to updates of the probabilistic model template $\mathcal{T}$ used by EvoChecker and to the *incremental synthesis of a probabilistic model* specifying a new configuration *c* that enables the system to meet its QoS requirements in the changed environment. This configuration is applied using an *Effectors* interface of the self-adaptive system.

To speed up the search for a new configuration, the use of EvoChecker at runtime builds on the principles of incrementality and exploits the fact that changes in a self-adaptive system are typically localised (Ghezzi 2012). As reported in other domains (Helwig and Wanka 2008; Kazimipour et al. 2014), and also discussed in our related work section (Sect. 8), an effective initialisation of the EA search can speed up its convergence and can yield better-quality solutions. Accordingly, EvoChecker maintains an *Archive* (cf. Fig. 4) of "effective" configurations identified during recent reconfigurations of the self-adaptive system. This *Archive* is used to "seed" each new search with a subset of recent configurations that encode solutions to potentially similar environment states experienced in the past.

To fully automate the EvoChecker operation, its runtime use combines the QoS requirements that target the optimisation of QoS attributes into a composite single objective. Similar to other approaches for developing self-adaptive systems (Ramirez et al. 2011), this objective requires the minimisation of a generalised *loss* function given by

$$loss(e, c) = \sum_{j=n_1+1}^{n_1+n_2} w_j \cdot attr_j(e, c), \tag{10}$$

where $w_j \geq 0$ are weight coefficients and at least one of them is strictly positive.[3]

These weight coefficients express the desired trade-off between the *j* QoS attributes. Fonseca and Fleming (Fonseca and Fleming 1997) show that for any positive set

[3] Alternative *loss* functions include lexicographic ordering, criterion-based, ϵ-constrained and aggregation-based (e.g., linear and nonlinear) functions of the relevant QoS attributes (Coello et al. 2006).

of coefficient values, the identified solution is always Pareto optimal (compared to all other solutions generated during the search). Selecting appropriate values for the coefficients is a responsibility of system designers. To this end, they can use domain knowledge to determine the value range of the QoS attributes comprising the loss function and assign appropriate coefficient values that reflect their relative importance (Coello et al. 2006). Note that although more complex, $loss$ is just another QoS attribute which can still be specified in the latest version of the probabilistic temporal logic languages supported by model checkers like PRISM (Kwiatkowska et al. 2011), so it fits the definition of an attribute from Eq. (5).

Example 5 To use EvoChecker in a self-adaptive variant of the FX system from our running example, the QoS attributes from Table 3 need to be combined into a *loss* function (10) that the self-adaptive system should minimise, e.g.:

$$loss(e, c) = w_1 \cdot (attr_1(e, c))^{-1} + w_2 \cdot attr_2(e, c) + w_3 \cdot attr_3(e, c),$$

with the weights w_1, w_2 and w_3 chosen based on the value ranges and on the relative importance of the three attributes. Note that the first attribute from the *loss* function is actually the reciprocal of the reliability attribute $attr_1$ from Table 3, as we want decreases in reliability to lead to rapid increases in *loss*. Using the failure probability $1–attr_1$ as the first attribute is also an option, although this choice yields a *loss* function that increases only linearly with the failure probability.

5.2.2 Runtime probabilistic model synthesis

When EvoChecker is used at runtime, the synthesis of probabilistic models is performed *incrementally*, i.e. by exploiting previously generated solutions, to speed up the synthesis of new solutions. This incremental synthesis is enabled by the *Archive* component shown in Fig. 4.

The use of EvoChecker within a self-adaptive system starts with an empty *Archive*, which is updated at the end of each reconfiguration step using an *archive updating strategy*. This strategy selects individuals from the final EA population synthesised by EvoChecker in the current reconfiguration step. Several criteria are used to enable this selection:

1. an individual that meets all n_1 constraints is preferred over an individual that violates one or more constraints;
2. from two individuals that satisfy all constraints, the individual with the lowest *loss* is preferred;
3. from two individuals that both violate at least one constraint, the individual with the lowest overall "level of violation" is preferred.[4]

While EvoChecker is not prescriptive about the calculation of the level of violation from the last criterion, the current version of our tool uses the following definition.

[4] In this scenario, the system might switch to a degraded, failsafe mode of operation. For the FX system, a failsafe mode is to skip the Order service invocation so that the system does not execute any trade that might be based on incorrect or stale data.

Definition 3 (*Constraints violation*) For each combination of an environment state $e \in Env$ and a configuration $c \in Cfg$ of a self-adaptive system, the level of violation of the n_1 QoS constraints is given by

$$violation(e, c) = \sum_{\substack{1 \le i \le n_1 \\ \neg(attr_i \bowtie_i bound_i)}} \alpha_i \cdot |bounds_i - attr_i(e, c)|, \tag{11}$$

where $\alpha_i > 0$ is a *violation weight* associated with the ith attribute.

Note that according to this definition, $violation(e, c) = 0$ for all (e, c) combinations that violate none of the n_1 bounds.

Example 6 Consider the QoS requirements of the FX system from Table 1. The only QoS constraint, **R1**, requires that workflow executions are at least $bound_1 = 0.98$ reliable. Hence, for any $(e, c) \in Env \times Cfg$,

$$violation(e, c) = \begin{cases} \alpha_1 \cdot |0.98 - attr_1(e, c)|, & \text{if } attr_1(e, c) < 0.98 \\ 0, & \text{otherwise} \end{cases}$$

The value of α_1 (i.e., $\alpha_1 = 100$ in our experiments from Sect. 7.2) is provided to EvoChecker by simply annotating the constraint **R1** with this value. EvoChecker automatically parses all such annotations and constructs the violation function for the system.

EvoChecker employs a *preference relation* based on criteria 1–3 to select configurations for storing in its *archive*. This relation and the EvoChecker *archive updating strategy* are formally defined below.

Definition 4 (*Preference relation*) Let $e \in Env$ be an environment state of a self-adaptive system. Then, given two configurations $c, c' \in Cfg$, configuration c is *preferred* over configuration c' (written $c \prec c'$) iff

$$\begin{aligned}
&\big(\forall 1 \le i \le n_1 \bullet attr_i(e, c) \bowtie_i bounds_i \, \wedge \\
&\qquad\qquad \exists 1 \le i \le n_1 \bullet \neg(attr_i(e, c') \bowtie_i bounds_i)\big) \vee && ① \\
&\big(\forall 1 \le i \le n_1 \bullet attr_i(e, c) \bowtie_i bounds_i \, \wedge \, attr_i(e, c') \bowtie_i bounds_i \\
&\qquad\qquad \wedge \, loss(e, c) < loss(e, c')\big) \vee && ② \\
&\big(\exists 1 \le i, j \le n_1 \bullet \neg(attr_i(e, c) \bowtie_i bounds_i) \wedge \neg(attr_j(e, c') \bowtie_j bounds_j) \\
&\qquad\qquad \wedge \, violation(e, c) < violation(e, c')\big) && ③
\end{aligned}$$

Definition 5 (*Archive updating strategy*) Let $C_e \subseteq Cfg$ be the set of configurations synthesised for the new environment state $e \in Env$ and $Arch$ be the archive before the change. Then an *archive updating strategy* is a function $\sigma : Cfg \to \mathbb{B}$ such that the updated archive at the end of the reconfiguration step is given by

$$Arch' = \{c \in Arch \cup C_e \mid \sigma(c)\} \tag{12}$$

We formally define four archive updating strategies that we will use to evaluate EvoChecker in Sect. 7:

1. The *prohibitive strategy* retains no configurations in the archive:

$$\sigma(c) = \mathit{false}, \forall c \in \mathit{Arch} \cup C_e \tag{13}$$

2. The *complete recent strategy* uses the entire population from the current adaptation step and removes the previous configurations from the archive:

$$\sigma(c) = \begin{cases} \mathit{true}, & \text{if } c \in C_e \\ \mathit{false}, & \text{otherwise} \end{cases} \tag{14}$$

3. The *limited recent strategy* keeps the x, $0 < x < \#C_e$, best configurations from the current adaptation step and removes the previous configurations from the archive:

$$\sigma(c) = \begin{cases} \mathit{true}, & \text{if } c \in C_e \text{ and } \mathit{position}(c) \leq x \\ \mathit{false}, & \text{otherwise} \end{cases}, \tag{15}$$

   where $\mathit{position} : C_e \rightarrow \{1, 2, \ldots, \#C_e\}$ is a function that gives the position of a configuration $c \in C_e$, i.e. $\mathit{position}(c) = \#\{c' \in C_e \setminus \{c\} \mid c' \prec c\} + 1$.
4. The *limited deep strategy* accumulates the x, $0 \leq x \leq \#C_e$ best configurations from all previous adaptation steps, given by

$$\sigma(c) = \begin{cases} \mathit{true}, & \text{if } c \in C_e \text{ and } \mathit{position}(c) \leq x \\ \mathit{true}, & \text{if } c \in \mathit{Arch} \\ \mathit{false}, & \text{otherwise} \end{cases} \tag{16}$$

As the *limited deep strategy* yields archives that grow in size after each reconfiguration step, some of the archive elements must be evicted when the archive size exceeds the size of the EA population. Possible eviction methods include: (i) discarding the "oldest" individuals (e.g. by implementing the archive as a circular buffer of size equal to that of the EA population); and (ii) performing a random selection.

Using the archive *Arch* to create the initial EA population is carried out by importing configurations from the archive into the population (cf. Fig. 4). If a complete population cannot be created in this way (e.g. because *Arch* is empty at the beginning of the first reconfiguration step and may not contain sufficient individuals for a few more steps), additional individuals are generated randomly to form a complete initial population.

The *assignment*, *reproduction* and *selection* operations applied during the iterative evolution of the population, and the EA termination criteria are similar to those from the design-time use of EvoChecker. However, a standard *single-objective (generational) evolutionary algorithm* is used instead of the multi-objective evolutionary algorithm, since there is only one optimisation objective (10).

6 Implementation

To ease the evaluation and adoption of the EvoChecker approach, we have implemented a tool that automates its use at both design time and runtime. Our EvoChecker tool uses the leading probabilistic model checker PRISM (Kwiatkowska et al. 2011) as its *Quantitative verification engine*, and the established Java-based framework for multi-objective optimization with metaheuristics jMetal (Durillo and Nebro 2011) for its *(Multi-objective) Evolutionary algorithm* component. We developed the remaining EvoChecker components in Java, using the Antlr[5] parser generator to build the *Template parser*, and implementing dedicated versions of the *Individual analyser*, *Monitor*, *Sensor* and *Effector* components.

The open-source code of EvoChecker, the full experimental results summarised in the following section, additional information about EvoChecker and the case studies used for its evaluation are available at http://www-users.cs.york.ac.uk/simos/EvoChecker.

7 Evaluation

We performed a wide range of experiments to evaluate the effectiveness of EvoChecker at both design time and runtime. The design-time use of Evo-Checker employs multi-objective genetic algorithms (MOGAs), while the runtime use of EvoChecker is based on a single-objective (generational) Genetic algorithm (GA). Experimenting with other types of evolutionary algorithms (e.g. evolution strategies, differential evolution) is part of our future work (Sect. 9). In Sects. 7.1 and 7.2, we describe the evaluation procedure and the results obtained for the design-time and runtime use of EvoChecker, respectively. For each use, we introduce the research questions that guided the experimental process, we describe the experimental setup, we summarise the methodology followed for obtaining and analysing the results, and finally, we present and discuss our findings. We conclude the evaluation with a review of threats to validity (Sect. 7.3).

7.1 Evaluating EvoChecker at design time

7.1.1 Research questions

The aim of our evaluation was to answer the following research questions.

RQ1 **(Validation): How does the design-time use of EvoChecker perform compared to random search?** We used this research question to establish if the application of EvoChecker at design time "comfortably outperforms a random search" (Harman et al. 2012a), as expected of effective search-based software engineering solutions.

[5] http://www.antlr.org.

Table 6 Analysed system variants for EvoChecker at design time

Variant	Details	Size	$T_{run}[s]$
FX_Small	$m_1 = \cdots = m_4 = 3, m_5 = m_6 = 1$	4.98E+31	0.0858
FX_Medium	$m_1 = \cdots = m_6 = 4$	1.39E+65	0.1695
FX_Large	$m_1 = \cdots = m_8 = 4$	7.22E+86	0.4162
DPM_Small	$Qmax_{H,L} \in \{1, \ldots, 10\}, m=2$	2E+14	0.1050
DPM_Medium	$Qmax_{H,L} \in \{1, \ldots, 15\}, m=2$	4.5E+14	0.2118
DPM_Large	$Qmax_{H,L} \in \{1, \ldots, 20\}, m=2$	8E+14	0.3796

RQ2 **(Comparison): How do EvoChecker instances using different MOGAs perform compared to each other?** Since we devised EvoChecker to work with any MOGA, we examined the results produced by EvoChecker instances using three established such algorithms [i.e., NSGA-II (Deb et al. 2002), SPEA2 (Zitzler et al. 2001), MOCell (Nebro et al. 2009)].

RQ3 **(Insights): Can EvoChecker provide insights into the tradeoffs between the QoS attributes of alternative software architectures and instantiations of system parameters?** To support system experts in their decision making, EvoChecker must provide insights into the tradeoffs between multiple QoS objectives. To address this question, we identified a range of decisions suggested by the EvoChecker results for the software systems considered in our evaluation.

7.1.2 Experimental setup

The experimental evaluation comprised multiple scenarios associated with two software systems from different application domains. The first is the foreign exchange (FX) service-based system described in Sect. 2. The second is a software-controlled dynamic power management (DPM) system adapted from Qiu et al. (2001), Sesic et al. (2008) and described on our project webpage.

We performed a wide range of experiments using the system variants from Table 6. The column 'Details' reports the number of third-party implementations for each service of the FX system[6]; and the capacity of the two request queues ($Qmax_H$ and $Qmax_L$) and the number of power managers available ($m = 2$) for the DPM system. The column 'Size' lists the configuration space size assuming a two-decimal points discretisation of the real parameters and probability distributions of the probabilistic model template (cf. Table 5). Given the nonlinearity of most probabilistic models, this is the minimum precision we could assume as an 0.01 increase or decrease in one of these parameters can have a significant effect in the evaluation of a QoS attribute. Finally, the column 'T_{run}' shows the average running time per system variant for evaluating a configuration. Note that the EvoChecker run time depends on the size of

[6] The $n = 8$ services used by FX_Large correspond to using two-part composite service implementations for the Technical analysis and Fundamental analysis services from Fig. 1.

model $\mathcal{M}$ and the time consumed by the probabilistic model checker to establish the $n_1 + n_2$ QoS attributes from Eq. (5) and on the computer used for the evaluation.

We conducted a two-part evaluation for EvoChecker. First, to assess the stochasticity of the approach when different MOGAs are adopted and also to eliminate the possibility that any observations may have been obtained by chance, we used specific scenarios for the system variants from Table 6. For the FX system variants, we chose realistic values for reliability, performance and invocation cost of third-party services implementations, while the values of parameters for the DPM system variants (i.e., power usage and transition rates) correspond to the real-world system from Qiu et al. (2001), Sesic et al. (2008). Second, to mitigate further the risk of accidentally choosing values that biased the EvoChecker evaluation, we defined a set of 30 different scenarios per FX system variant with varied services characteristics for each scenario.

7.1.3 Evaluation methodology

We used the following established MOGAs to evaluate the use of EvoChecker at design time: NSGA-II (Deb et al. 2002), SPEA2 (Zitzler et al. 2001) and MOCell (Nebro et al. 2009).

In line with the standard practice for evaluating the performance of stochastic optimisation algorithms (Arcuri and Briand 2011), we performed multiple (i.e., 30) independent runs for each system variant from Table 6 and each multiobjective optimisation algorithm, i.e., NSGA-II, SPEA2, MOCell and random search. Each run comprised 10,000 evaluations, each using a different initial population of 100 individuals, single-point crossover with probability $p_c = 0.9$, and single-point mutation with probability $p_m = 1/n_p$, where n_p is the number of configuration parameters for a particular system variant. All the experiments were run on a CentOS Linux 6.5 64bit server with two 2.6GHz Intel Xeon E5-2670 processors and 32GB of memory.

Obtaining the actual Pareto front for our system variants is unfeasible because of their very large configuration spaces. Therefore, we adopted the established practice (Zitzler et al. 2008) of comparing the Pareto front approximations produced by each algorithm with the *reference Pareto front* comprising the nondominated solutions from all the runs carried out for the analysed system variant. For this comparison, we employed the widely-used *Pareto-front quality indicators* below, and we will present their means and box plots as measures of central tendency and distribution, respectively:

- I_ϵ (**Unary additive epsilon**) (Zitzler et al. 2003). This is the minimum additive term by which the elements of the objective vectors from a Pareto front approximation must be adjusted in order to dominate the objective vectors from the reference front. This indicator presents convergence to the reference front and is *Pareto compliant*.[7] Smaller I_ϵ values denote better Pareto front approximations.
- I_{HV} (**Hypervolume**) (Zitzler and Thiele 1999). This indicator measures the volume in the objective space covered by a Pareto front approximation with respect

[7] Pareto compliant indicators do not "contradict" the order introduced by the Pareto dominance relation on Pareto front approximations (Zitzler et al. 2007).

to the reference front (or a reference point). It measures both convergence and diversity, and is strictly Pareto compliant (Zitzler et al. 2007). Larger I_{HV} values denote better Pareto front approximations.

I_{IGD} (**Inverted Generational Distance**) (Van Veldhuizen 1999). This indicator gives an "error measure" as the Euclidean distance in the objective space between the reference front and the Pareto front approximation. I_{IGD} shows both diversity and convergence to the reference front. Smaller I_{IGD} values signify better Pareto front approximations.

We used inferential statistical tests to compare these quality indicators across the four algorithms (Arcuri and Briand 2011; Harman et al. 2012b). As is typical of multiobjective optimisation (Zitzler et al. 2008), the Shapiro–Wilk test showed that the quality indicators were not normally distributed, so we used the Kruskal–Wallis nonparametric test with 95% confidence level ($\alpha = 0.05$) to analyse the results without making assumptions about the distribution of the data or the homogeneity of its variance. We also performed a post-hoc analysis with pairwise comparisons between the algorithms using Dunn's pairwise test, controlling the family-wise error rate with the Bonferroni correction $p_{crit} = \alpha/k$, where k is the number of comparisons.

7.1.4 Results and discussion

RQ1 (Validation) We carried out the experiments described in the previous section and we report their results in Table 7 and Fig. 5. The '+' from the last column of the table entries indicate that the Kruskal–Wallis test showed significant difference among the four algorithms (p value<0.05) for all six system variants and all Pareto-front quality indicators.

For both systems, EvoChecker using any MOGA achieved considerably better results than random search, for all quality indicators and system variants. The post hoc analysis of pairwise comparisons between random search and the MOGAs provided statistical evidence about the superiority of the MOGAs for all system variants and for all quality indicators. The best and, when obtained, the second best outcomes of this analysis per system variant and quality indicator are shaded and lightly shaded in the result tables, respectively. This superiority of the results obtained using EvoChecker with any of the MOGAs over those produced by random search can also be seen from the boxplots in Fig. 5.

We qualitatively support our findings by showing in Figs. 6 and 7 the Pareto front approximations achieved by EvoChecker with each of the MOGAs and by random search, for a typical run of the experiment for the DPM and FX system variants, respectively. We observe that irrespective of the MOGA, EvoChecker achieves Pareto front approximations with more, better spread and higher quality nondominated solutions than random search.

As explained earlier, the parameters we used for the DPM system variants (power usage, transition rates, etc.) correspond to the real-world system (Qiu et al. 2001; Sesic et al. 2008). In contrast, for the FX system variants we chose realistic values for the reliability, performance and cost of the third-party services. To mitigate the risk of accidentally choosing values that biased the EvoChecker evaluation, we performed

Table 7 Mean quality indicator values for a specific scenario of the FX system variants (top) and DPM system variants (bottom) from Table 6

Variant	NSGA-II	SPEA2	MOCell	Random	
		I_ϵ (Epsilon)			
FX_Small	0.6258	0.5083	0.6745	2.2274	+
FX_Medium	1.6379	2.0105	2.0486	6.1529	+
FX_Large	3.8528	5.2777	4.6366	13.0234	+
		I_{HV} (Hypervolume)			
FX_Small	0.611	0.628	0.608	0.593	+
FX_Medium	0.719	0.725	0.702	0.606	+
FX_Large	0.657	0.675	0.633	0.555	+
		I_{IGD} (Inverted Generational Distance)			
FX_Small	0.00123	0.00129	0.00125	0.00145	+
FX_Medium	0.00192	0.00207	0.00200	0.00316	+
FX_Large	0.00244	0.00255	0.00272	0.00395	+

Variant	NSGA-II	SPEA2	MOCell	Random	
		I_ϵ (Epsilon)			
DPM_Small	0.0209	0.0130	0.0242	0.1403	+
DPM_Medium	0.0225	0.0123	0.0489	0.1996	+
DPM_Large	0.0229	0.0147	0.0884	0.2497	+
		I_{HV} (Hypervolume)			
DPM_Small	0.4455	0.4458	0.4396	0.4022	+
DPM_Medium	0.4487	0.4499	0.4386	0.3946	+
DPM_Large	0.4528	0.4549	0.4395	0.3947	+
		I_{IGD} (Inverted Generational Distance)			
DPM_Small	0.00023	0.00018	0.00016	0.00062	+
DPM_Medium	0.00024	0.00019	0.00028	0.00091	+
DPM_Large	0.00024	0.00020	0.00038	0.00109	+

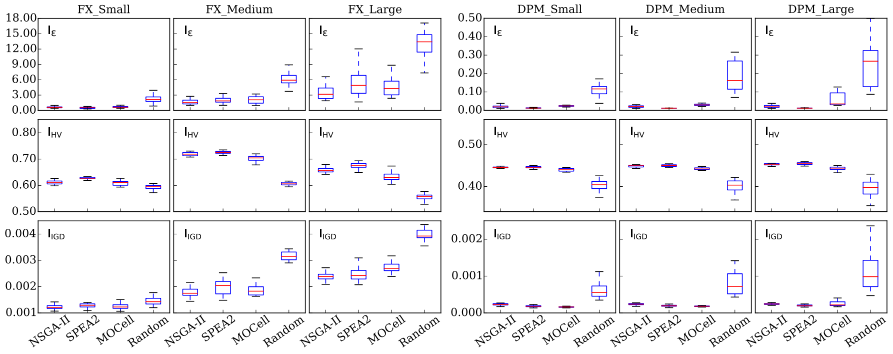

Fig. 5 Boxplots for a specific scenario of the FX system variants (left) and DPM system variants (right) from Table 6, evaluated with quality indicators I_ϵ, I_{HV} and I_{IGD}

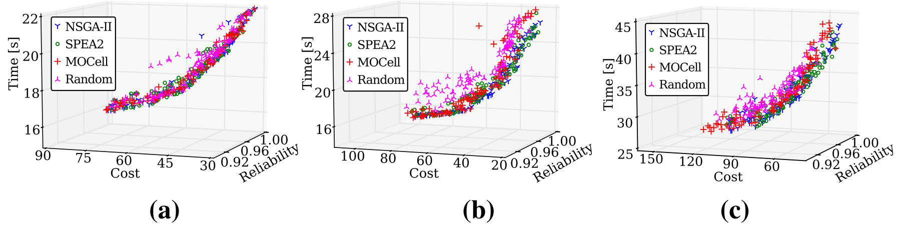

Fig. 6 Typical Pareto front approximations for the FX system variants and optimisation objectives R2–R4 from Table 4. **a** FX_Small, **b** FX_Medium, **c** FX_Large

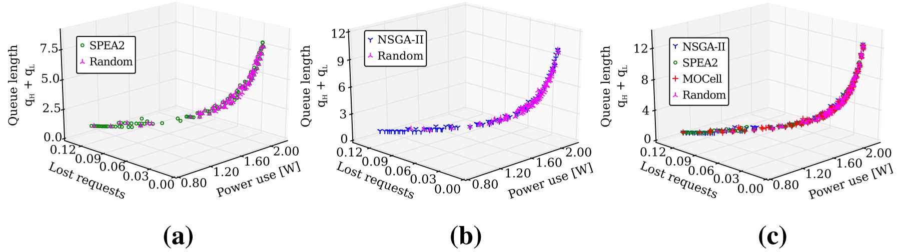

Fig. 7 Typical Pareto front approximations for the DPM system variants. The DPM optimisation objectives involve minimising the steady-state power utilisation ("Power use"), minimising the number of requests lost at the steady state ("Lost requests"), and minimising the capacity of the DPM queues ("Queue length"). **a** DPM_Small, **b** DPM_Medium, **c** DPM_Large

additional experiments comprising 300 independent runs per FX system variant (900 runs in total) in which these parameters were randomly instantiated. To allow for a fair comparison across the experiments comprising the 30 different FX scenarios, and to avoid undesired scaling effects, we normalise the results obtained for each quality indicator per experiment within the range [0,1]. The results of this further analysis, shown in Table 8 and Fig. 8, validate our findings.

Considering all these results, we have strong empirical evidence that the EvoChecker significantly outperforms random search, for a range of system variants from two different domains, and across multiple widely-used MOGAs. This also confirms the challenging and well-formulated nature of the *multi-objective probabilistic model synthesis problem* we introduced in Sect. 5.1.1.

RQ2 (Comparison) To compare EvoChecker instances based on different MOGAs, we first observe in Table 7 that NSGA-II and SPEA2 outperformed MOCell for all system variant–quality indicator combinations except DPM_Small (I_{IGD}). Between SPEA2 and NSGA-II, the former achieved slightly better results for the smaller configuration spaces of the DPM system variants (across all indicators) and for the I_{HV} indicator (across all system variants), whereas NSGA-II yielded Pareto-front approximations with better I_ϵ and I_{IGD} indicators for the larger configuration spaces of the FX system variants (except the combination FX_Small (I_ϵ)).

Additionally, we carried out the post-hoc analysis described in Sect. 7.1.3, for 9 system variants (counting separately the FX system variants with chosen services characteristics and those comprising the adaptation scenarios) × 3 quality indicators =

Table 8 Mean quality indicator values across 30 different scenarios for the FX system variants from Table 6

Variant	NSGA-II	SPEA2	MOCell	Random	
		I_ϵ **(Epsilon)**			
FX_Small	0.2212	0.2209	0.2272	0.6200	+
FX_Medium	0.3393	0.3664	0.3645	0.7568	+
FX_Large	0.3396	0.3764	0.3625	0.7970	+
		I_{HV} **(Hypervolume)**			
FX_Small	0.9374	0.9914	0.9337	0.9016	+
FX_Medium	0.9514	0.9848	0.9219	0.8138	+
FX_Large	0.9467	0.9804	0.8962	0.7868	+
		I_{IGD} **(Inverted Generational Distance)**			
FX_Small	0.6365	0.5348	0.6390	0.8000	+
FX_Medium	0.5919	0.5790	0.6114	0.7957	+
FX_Large	0.5887	0.5622	0.6561	0.8884	+

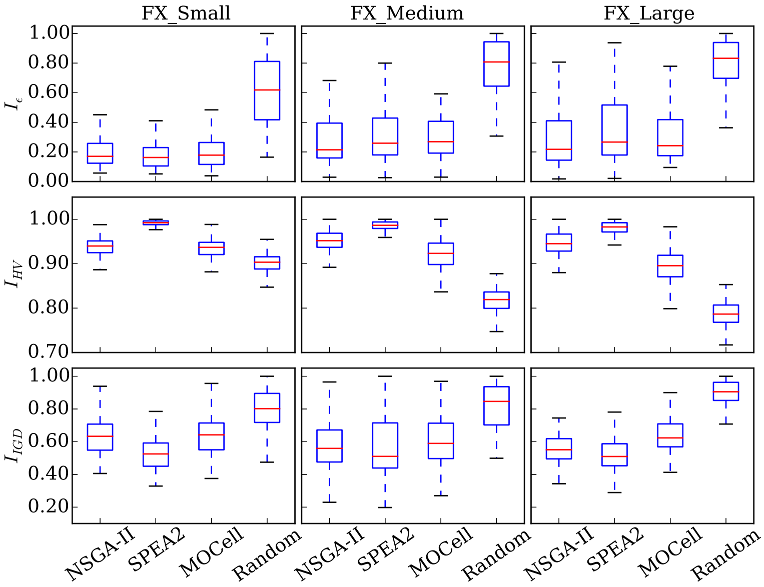

Fig. 8 Boxplots for the FX system variants (Table 6) across 30 different scenarios, evaluated using the quality indicators I_ϵ, I_{HV} and I_{IGD}

27 tests. Out of these tests, 22 tests (i.e., a percentage of 81.4%) showed high statistical significance in the differences between the performance achieved by EvoChecker with different MOGAs (Table 9). The five system variant–quality indicator combinations for which the tests were unsuccessful are: FX_Medium (I_ϵ), FX_Small_Adapt (I_ϵ), FX_ Medium_Adapt(I_ϵ), FX_Small(I_{IGD}) and FX_Medium(I_{IGD}).

Table 9 System variants for which the MOGAs in rows are significantly better than the MOGAs in columns

		NSGA-II									SPEA2									MOCell								
		1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9
NSGA-II	I_ϵ															✓			✓		✓	✓						
	I_{HV}																			✓	✓	✓		✓	✓		✓	✓
	I_{IGD}																								✓			✓
SPEA2	I_ϵ	✓	✓	✓																✓	✓	✓	✓					
	I_{HV}				✓	✓	✓	✓	✓	✓										✓	✓	✓	✓	✓	✓	✓	✓	✓
	I_{IGD}	✓	✓	✓	✓																	✓			✓	✓	✓	✓
MOCell	I_ϵ																											
	I_{HV}																											
	I_{IGD}	✓	✓								✓																	

Key: 1:DPM_Small, 2:DPM_Medium, 3:DPM_Large, 4:FX_Small, 5:FX_Medium, 6:FX_Large, 7:FX_Small_Random, 8:FX_Medium_Random, 9:FX_Large_Random

These results show that if the probabilistic model synthesis problem can be formulated as a multi-objective optimisation problem, then several MOGAs can be used to synthesise the Pareto approximation sets PF and PS effectively. Selecting between alternative MOGAs entails using domain knowledge about the synthesis problem, and analysing the individual strengths of the MOGAs (Harman et al. 2012a). The results also confirm the generality of the EvoChecker approach, showing that its functionality can be realised using multiple established MOGAs.

RQ3 (Insights) We performed qualitative analysis of the Pareto front approximations produced by EvoChecker, in order to identify actionable insights. We present this for the FX and DPM Pareto front approximations from Figs. 6 and 7, respectively.

First, the EvoChecker results enable the identification of the "point of diminishing returns" for each system variant. The results from Fig. 6 show that configurations with costs above approximately 52 for FX_Small, 61 for FX_Medium and 94 for FX_Large provide only marginal response time and reliability improvements over the best configurations achievable for these costs. Likewise, the results in Fig. 7 show that DPM configurations with power use above 1.7W yield insignificant reductions in the number of lost requests, whereas configurations with even slightly lower power use lead to much higher request loss. This key information helps system experts to avoid unnecessarily expensive solutions.

Second, we note the high density of solutions in the areas with low reliability (below 0.95) for the FX system in Fig. 6, and with high request loss (above 0.09) for the DPM system in Fig. 7. For the FX system, for instance, these areas correspond to the use of the probabilistic invocation strategy, for which numerous service combinations can achieve similar reliability and response time with relatively low, comparable costs. Opting for a configuration from this area will make the FX system susceptible to failures, as when the only implementation invoked for an FX service fails, the entire workflow execution will also fail. In contrast, reliability values above 0.995 correspond to expensive configurations that use the sequential selection strategy; e.g., FX_Small must use the sequential strategy for the *Market watch* and *Fundamental analysis* services in order to achieve 0.996 reliability.

Third, the EvoChecker results reveal configuration parameters that QoS attributes are particularly sensitive to. For the FX system, for example, we noticed a strong dependency of the workflow reliability on the service invocation strategy and the number of implementations used for each service. Configurations from high-reliability areas of the Pareto front not only use the sequential strategy, but also require multiple services per FX service (e.g., three FX service providers are needed for success rates above 0.99).

Finally, we note EvoChecker's ability to produce solutions that: (i) cover a wide range of values for the QoS attributes from the optimisation objectives of the FX and DPM systems; and (ii) include alternatives with different tradeoffs for fixed values of one of these attributes. Thus, for 0.99 reliability, the experiment from Fig. 6 generated four alternative FX_Large configurations, each with a different cost and execution time. Similar observations can be made for a specific value of either of the other two QoS attributes. These results support the system experts in their decision making.

7.2 Evaluating EvoChecker at runtime

7.2.1 Research questions

We evaluated the runtime use of EvoChecker to answer the research questions below.

RQ4 **(Effectiveness): Can EvoChecker support dependable adaptation?** With this research question we examine whether our approach can identify new effective configurations at runtime.

RQ5 **(Validation): How does EvoChecker perform compared to random search?** Following the standard practice in search-based software engineering (Harman et al. 2012b), with this research question we aim to determine whether our approach performs better than random search.

RQ6 **(Archive-strategy comparison): How do EvoChecker instances based on different archive updating strategies compare to each other?** We used this research question to analyse the impact of various *archive updating strategies* in the performance of EvoChecker. To this end, we study whether specific *strategies* improve the quality of a search and/or help identifying faster an effective configuration. We also investigate possible relationships between *archive updating strategies* and specific adaptation events.

7.2.2 Experimental setup

For the experimental evaluation, we used two self-adaptive software systems from different application domains. The first is the FX service-based system from Sect. 2 and the second is an embedded system from the area of unmanned underwater vehicles (UUVs) adapted from Calinescu et al. (2015), Gerasimou et al. (2014), Gerasimou et al. (2017) and described on our project webpage.

Table 10 Analysed system variants for the runtime EvoChecker

Variant	Details	Size	$T_{run}[s]$
UUV_Medium	$m = 5, r_1, r_2, \ldots, r_5 \in [0Hz, 8Hz], sp \in [0, 10m/s]$	1.04E+19	0.0076
UUV_Large	$m = 10, r_1, r_2, \ldots, r_{10} \in [0Hz, 8Hz], sp \in [0, 10m/s]$	1.09E+35	0.1622
FX_Small	$m_1 = \cdots = m_4 = 3, m_5 = m_6 = 1$	4.98E+31	0.0312
FX_Medium	$m_1 = \cdots = m_6 = 4$	1.39E+65	0.0953

Table 11 QoS requirements for the UUV system

ID	Informal description
R1	"The UUV must take at least 500 accurate measurements for each 100 m travelled"
R2	"The UUV sensors must not consume more than 1000 J per 100 m travelled"
R3	"The speed with which the UUV travels should be maximised"
R4	"The energy consumed by the UUV sensors should be minimised"

We applied EvoChecker at runtime to the system variants from Table 10, aiming to assess its behaviour for multiple configuration space sizes. As before (cf. Table 6), the column 'Details' shows for the UUV system the number of sensors, their measurement rates and the UUV speed, while for the FX system the number of third-party implementations for each service. The column 'Size' reports the size of the configuration space that an exhaustive search would need to explore using two-decimal precision for the real parameters and probability distributions of the probabilistic model template (cf. Table 5). Finally, the column 'T_{run}' shows the average time required by EvoChecker to evaluate a configuration on a 2.6GhZ Intel Core i5 Macbook Pro computer with 16GB memory, running Mac OSX 10.9.

To evaluate EvoChecker at runtime, we identified several changes that can cause each UUV and FX system variant to adapt. These changes cover a wide range of the possible values that the environment parameters of each system variant can take (Table 12). Due to these changes, the systems experience problems while providing service (e.g., service degradation, violation of QoS requirements) and therefore are forced to adapt. Sensors in the UUV variants, beyond normal behaviour, encounter periods of unexpected changes (C1–C12) during which their rates change dramatically, including sensor failures and recovery from these failures, and significant variation in measurement rates. Changes C1–C13 in FX comprise sudden minor or significant increase in response time and decline in reliability of service implementations, and complete failure or recovery of service implementations. For instance, change C7 in FX_Small represents a deviation from the nominal reliability of the first and third service implementations of the *Market_Watch* service (cf. Fig. 2): before the change, $r_{11} = 0.98$ and $r_{13} = 0.993$; and, after the change, $r_{11} = 0.89$ and $r_{13} = 0.93$. This is a significant change because the FX system cannot meet the reliability requirement (Table 1) using

Table 12 Changes in environment state of UUV and FX system variants used for evaluating EvoChecker at runtime

ID	UUV_Medium	UUV_Large	FX_Small	FX_Medium
C1	$r_1 \leftrightarrow, \ldots, r_5 \leftrightarrow$	$r_1 \leftrightarrow, \ldots, r_9 \leftrightarrow$	$r_{11}\leftrightarrow, \ldots, r_{61}\leftrightarrow$ $t_{11} \leftrightarrow, \ldots,$ $t_{61} \leftrightarrow$	$r_{11}\leftrightarrow, \ldots, r_{64}\leftrightarrow$ $t_{11} \leftrightarrow, \ldots,$ $t_{64} \leftrightarrow$
C2	$r_1 \leftrightarrow, \ldots, r_5 \leftrightarrow$	$r_1 \leftrightarrow, \ldots, r_9 \leftrightarrow$	$r_{11}\leftrightarrow, \ldots, r_{61}\leftrightarrow$ $t_{11} \leftrightarrow, \ldots,$ $t_{61} \leftrightarrow$	$r_{11}\leftrightarrow, \ldots, r_{64}\leftrightarrow$ $t_{11} \leftrightarrow, \ldots,$ $t_{64} \leftrightarrow$
C3	$r_1 \downarrow, r_5 \downarrow$	$r_1 \downarrow, r_4 \downarrow r_9 \downarrow$	$r_{11} \downarrow, r_{13} \downarrow$	$r_{11} \downarrow, r_{13} \downarrow, r_{14} \downarrow$
C4	$r_1 \leftrightarrow, r_5 \leftrightarrow$	$r_1 \leftrightarrow, r_4 \leftrightarrow r_9 \leftrightarrow$	$r_{11} \leftrightarrow, r_{13} \leftrightarrow$	$r_{11} \leftrightarrow, r_{13} \leftrightarrow,$ $r_{14} \leftrightarrow$
C5	$r_2 \downarrow, r_4 \downarrow$	$r_2 \downarrow, r_4 \downarrow, r_8 \downarrow,$ $r_{10} \downarrow$	$r_{21} \downarrow, r_{22} \downarrow$	$r_{21} \downarrow, r_{22} \downarrow, r_{24} \downarrow$
C6	$r_2 \leftrightarrow, r_4 \leftrightarrow$	$r_2 \leftrightarrow, r_4 \leftrightarrow, r_8 \leftrightarrow,$ $r_{10} \leftrightarrow$	$r_{21} \leftrightarrow, r_{22} \leftrightarrow$	$r_{21} \leftrightarrow, r_{22} \leftrightarrow,$ $r_{24} \leftrightarrow$
C7	$r_2 \downarrow$	$r_8 \downarrow, r_{10} \downarrow$	$r_{11} \downarrow, r_{13} \downarrow$	$r_{11} \downarrow, r_{13} \downarrow, r_{14} \downarrow$
C8	$r_2 \leftrightarrow$	$r_8 \leftrightarrow, r_{10} \leftrightarrow$	$r_{11} \leftrightarrow, r_{13} \leftrightarrow$	$r_{11} \leftrightarrow, r_{13} \leftrightarrow,$ $r_{14} \leftrightarrow$
C9	$r_1 \downarrow, r_5 \downarrow$	$r_1 \downarrow, r_5 \downarrow r_9 \downarrow$	$t_{41} \uparrow, t_{42} \uparrow$	$t_{41} \uparrow, t_{42} \uparrow, t_{44} \uparrow$
C10	$r_1 \leftrightarrow, r_5 \leftrightarrow$	$r_1 \leftrightarrow, r_5 \leftrightarrow r_9 \leftrightarrow$	$t_{41} \leftrightarrow, t_{42} \leftrightarrow$	$t_{41} \leftrightarrow, t_{42} \leftrightarrow,$ $t_{44} \leftrightarrow$
C11	$r_1 \downarrow, r_3 \downarrow, r_5 \downarrow$	$r_1 \downarrow, r_3 \downarrow, r_5 \downarrow$ $r_7 \downarrow, r_9 \downarrow, r_{10} \downarrow$	$t_{51} \uparrow, t_{52} \downarrow$	$t_{51} \uparrow, t_{52} \uparrow, t_{53} \uparrow$
C12	$r_1 \leftrightarrow, r_3 \leftrightarrow, r_5 \leftrightarrow$	$r_1 \leftrightarrow, r_3 \leftrightarrow, r_5 \leftrightarrow$ $r_7 \leftrightarrow, r_9 \leftrightarrow,$ $r_{10} \leftrightarrow$	$t_{51} \leftrightarrow, t_{52} \leftrightarrow$	$t_{51} \leftrightarrow, t_{52} \leftrightarrow,$ $t_{53} \leftrightarrow$
C13			$r_{11} \downarrow, r_{12} \downarrow, r_{21} \downarrow,$ $r_{22} \downarrow, r_{31} \downarrow,$ $r_{33} \downarrow, r_{42} \downarrow,$ $r_{43} \downarrow$	$r_{11} \downarrow, r_{12} \downarrow, r_{13} \downarrow,$ $r_{21} \downarrow, r_{22} \downarrow,$ $r_{31} \downarrow, r_{33} \downarrow,$ $r_{43} \downarrow, r_{44} \downarrow$ $r_{51} \downarrow, r_{52} \downarrow,$ $r_{54} \downarrow r_{62} \downarrow,$ $r_{64} \downarrow$

Key↔: nominal value of environment characteristic (i.e., reliability, response time)
↓(↑): decrease(increase), i.e., a change, in value of environment characteristic
r_i: sensor reliability for the UUV_Medium and UUV_Large variants from Table 10
$r_{im_i}(t_{im_i})$: service reliability (response time) for the FX_Small and FX_Medium variants from Table 10

only the degraded service implementations (i.e., $x_{11} = 1$, $x_{12} = 0$, $x_{13} = 1$). Instead, a valid configuration should always realise the functionality of the *Market_Watch* service by selecting its second service implementation (thus setting $x_{12} = 1$).[8] We make available the EvoChecker templates for the changes in Table 12 on our project webpage.

[8] Although the other service implementations have lower reliability, they are still functional and can be used within the sequential strategy in conjunction with the second service implementation to improve further the FX workflow reliability.

Answering research question **RQ1** entails making the configuration space size tractable for exhaustive search. Searching exhaustively through the configuration space of the UUV_Medium variant (which has the smallest configuration space and the shortest average time per evaluation) would take an estimated $2.19 \cdot 10^{13}$ h (given the $1.04 \cdot 10^{19}$ configurations to analyse, and a mean analysis time of 0.0076 s). Thus, we used the UUV_Medium variant but disabled three of its sensors, leaving just under $2.56 \cdot 10^{9}$ possible configurations. We also disregarded the adaptation time, since it is too large for exhaustive search. For the same reason, we performed this assessment on a subset of the UUV changes (i.e. C1, C3, C4); these changes correspond to a representative sample of the UUV changes from Table 12.

For using EvoChecker at runtime, we define the QoS optimisation objectives as a *loss* function; see Eq. (10) in Sect. 5.2. We used the *loss* function from Example 5 with $w_1 = 0.2$, $w_2 = 0.004$ and $w_3 = 0.016$ for the FX system variants, and a similarly defined loss function (provided on our project webpage) for the UUV system variants.

Since EvoChecker at runtime employs a single optimisation objective, we employ an elitist single objective GA. Recall that an elitist GA propagates the best individuals to the next generation. With elitism, if the GA discovers the best solution, then the entire population will eventually converge to this solution (Coello et al. 2006).

To investigate whether different archive updating strategies (cf. Def. 5) can improve the efficiency of EvoChecker, we realised the strategies from (13)–(16). To this end, we created four different GA variants, each enhanced with one of the following archive updating strategies:

PGA: a *prohibitive strategy* (13) that does not keep any configurations in the archive. Thus, a search for a new configuration starts without using any prior knowledge.

CRGA: a *complete recent strategy* (14) that puts in the archive the entire population from the current adaptation step and discards all previous configurations.

LRGA: a *limited recent strategy* (15) that stores in the archive the two best configurations (i.e. $x = 2$) from the current adaptation step, and removes all the other configurations from the archive.

LDGA: a *limited deep strategy* (16) that accumulates in the archive the two best configurations (i.e. $x = 2$) from *all* previous adaptation steps. If the archive size exceeds the initial size of the GA population, then a random selection is carried out to select the configurations that will comprise the seed for the next search.

7.2.3 Evaluation methodology

We adopted the established procedure in search-based software engineering for the analysis of optimisation algorithms (Arcuri and Briand 2011). Thus, for each system variant from Table 10 we carried out 30 independent runs per optimisation algorithm using the adaptation events (changes) in Table 12 sequentially. We assume that the time interval between successive changes is long enough that enables running EvoChecker. All algorithms used a population of 50 individuals. The GAs used single-point crossover with probability $p_c = 0.9$ and single-point mutation with probability

$p_m = 1/n_k$, where n_k is the number of system configuration parameters from the configuration space *Cfg*. Each algorithm was executed for 5000 iterations. When no improvement was detected for 1000 successive iterations (i.e. 20% of the allocated evolution time), the evolution terminated early. The solution corresponding to the best individual from the last population was used to reconfigure the system. After normalisation, and for ease of presentation, we assigned the maximum loss of 1.00 for each event in which an algorithm failed to find a configuration satisfying the QoS constraints of the system. We used the loss corresponding to the selected configuration as a quality indicator to compare the effectiveness of the optimisation algorithms and answer research questions **RQ4–RQ6**. Furthermore, we combined these quality results with data about the number of iterations executed by the algorithms (for all 30 independent runs) to assess their ability both to identify good solutions and converge (or stagnate, if no effective solution was found within the available time).

Following the standard advice for assessing the performance of optimisation algorithms, we used inferential statistical tests (Arcuri and Briand 2011; Coello et al. 2006). First, we analysed the normality of data and confirmed its deviation from the normal distribution using the Shapiro–Wilk test. Then, we used the non-parametric tests Mann–Whitney and Kruskal–Wallis with 95% confidence level ($\alpha = 0.05$) to analyse the results without making assumptions about the data distribution or the homogeneity of its variance. Also, to compare the EvoChecker instantiations with different archive updating strategies, we ran a post-hoc analysis using Dunn's pairwise test, controlling the family-wise error rate using the Bonferroni correction $p_{crit} = \alpha/k$, where k is the number of comparisons.

Finally, when statistical significance exists, we establish the practical importance of the observed effect. Therefore, we used the Varga and Delaney's effect size measure (Vargha and Delaney 2000; Arcuri and Briand 2011). When comparing algorithms A and B, this measure returns the probability $A_{AB} \in [0, 1]$ that algorithm A will yield better results than algorithm B. For instance, if $A_{AB} = 0.5$ then the algorithms are equivalent, while if $A_{AB} = 0.8$ then algorithm A will achieve better results 80% of the time.

7.2.4 Results and discussion

RQ4 (Effectiveness) We begin the presentation of our results by examining whether EvoChecker at runtime can identify new effective configurations in response to unexpected environment and/or system events. To answer this research question we performed two types of experiments.

First, we used the UUV_Medium system variant and assessed the effectiveness of the selected configurations using PGA compared to those generated by exhaustive search. We reduced the configuration space of UUV_Medium, using the process described in Sect. 7.2.2, to make it tractable for exhaustive search. For all the events, the EvoChecker with PGA found configurations satisfying system QoS constraints R1 and R2 (cf. Table 11) with average loss not more than 9% of the optimal loss given by the configurations found by exhaustive search. Both time and memory overheads incurred by exhaustive search were approximately two orders of magnitude larger than PGA.

For the second experiment, we analysed how the events in FX_Small system variant from Table 12 affected its compliance with QoS requirement R1 (i.e. workflow reliability) and varied the loss before (using the current configuration) and after (using the new configuration) each adaptation. Figure 9 depicts a typical run (timeline) of these changes and the impact of the configurations selected by the no-archive version of EvoChecker (i.e. PGA) in workflow reliability and loss.

Irrespective of the change in environment state, either being a serious decrease in workflow reliability or a moderate increase in response time, EvoChecker always managed to successfully self-adapt the system by identifying configurations that met QoS constraint R1 (cf. Table 1). Furthermore, EvoChecker maintained a balanced system loss of approximately 0.845. Given that searching exhaustively the configuration space is unfeasible and that the average running time for evaluating a single configuration is less than 1s (cf. Table 10), these experimental results indicate that our approach can support system adaptation.

We also analysed changes C3, C5, C7 and C13, in which the system exhibited a significant decrease in workflow reliability, caused by decrease in reliability of the service implementations used at various points in time. Due to this abrupt change, the currently used service implementations failed to meet requirement R1 and EvoChecker was invoked to carry out the search for a new configuration. As an example, for change C13, the system experienced a serious disruption in about 50% of the available service implementations. As a result, workflow reliability fell to only 72%. The newly found configuration restored compliance with R1 (i.e. approximately 98.5%), but increased the probabilities of using more expensive implementations, yielding a significantly higher expected loss of 0.935.

Another interesting observation concerns change C10 (cf. Table 12) in which two previously under-performing service implementations (those with increased response time t_{41} and t_{42}) recover. Although no requirement violation occurs, i.e. workflow reliability R1 is not affected by this change, the system loss corresponding to the new configuration selected by EvoChecker is slightly higher compared to the configuration before the change. Since for each change PGA starts a new search and does not use any knowledge gained from previous adaptation steps, this is expected. As we explain in research question RQ6, this issue can be addressed using one of the other archive updating strategies which seed a new GA search with configurations from the archive.

RQ5 (Validation). To answer this research question we compared the no-archive version of incremental EvoChecker (i.e. PGA) with random search (RS). For conciseness, we include a representative sample of reconfiguration events. Thus, Figs. 10 and 11 show the evolution of the algorithms every 500 iterations (i.e., 10 generations) for the FX_Small variant for changes C4, C7, C11 and C13, and for the UUV_Large variant for changes C7 and C12, respectively. When an algorithm terminated early, we propagated the final loss to the remaining evolution stages (i.e. until the 5000th iteration). An asterisk '$*$' next to each algorithm's boxplot denotes when the algorithm terminated for *all* 30 runs.

For both variants of the FX and UUV systems and for all 25 events, the EvoChecker employing PGA identified configurations that met QoS requirements and achieved lower loss than RS. We obtained statistical significance (p value < 0.05) using the

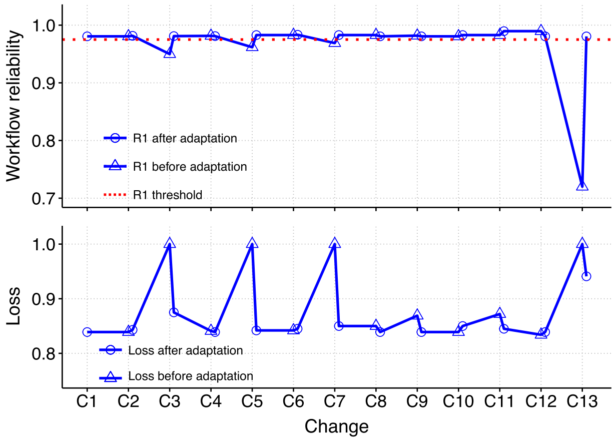

Fig. 9 Variation in workflow reliability and system loss of the FX_Small variant due to the changes from Table 12 and system adaptation using EvoChecker with no archive use (i.e. PGA)

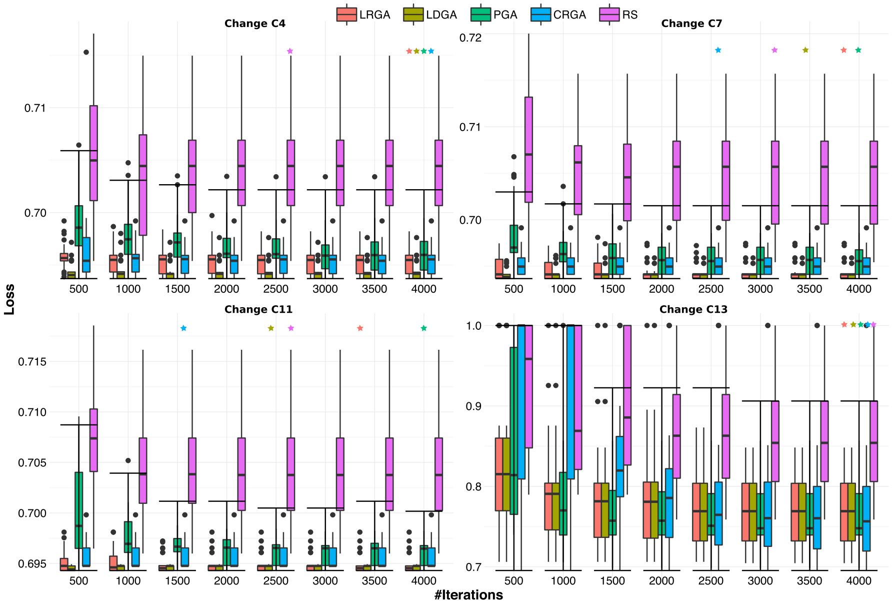

Fig. 10 Boxplots for changes in environment state C4, C7, C11, C13 of the FX_Small system variant using LRGA, LDGA, PGA, CRGA, and RS. The asterisk next to each algorithm's boxplot signifies when the algorithm terminated for *all* 30 runs

Mann–Whitney test for all system variants and for all events, with the p value being in the range [1.689E−02, 1.669E−11]. In fact, as the size of the system increases (cf. Table 10), PGA's ability to outperform RS becomes more evident.

We also measured the improvement magnitude using the $A_{\mathrm{PGA,RS}}$ effect size metric (Vargha and Delaney 2000). For all evaluated events and evolution stages, the effect size was large with $A_{\mathrm{PGA,RS}} \in [0.696, 1.00]$. Thus, PGA achieved better results than

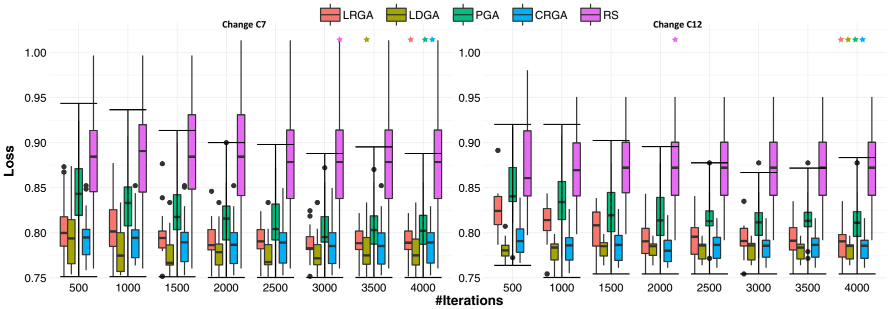

Fig. 11 Boxplots for changes C7, C12 of the UUV_Large system variant using LRGA, LDGA, PGA, CRGA, and RS. The asterisk next to each algorithm's boxplot signifies when the algorithm terminated for *all* 30 runs

RS at least 69.6% of the time, while in some events, especially for the larger system variants FX_Medium and UUV_Large, the dominance reached 100%.

Another interesting finding concerns the evolution of the populations of these algorithms. Despite the overall performance difference, at the beginning of the evolution, i.e. 200–300 iterations, both the *p* value and effect size are on the lower end of their respective value ranges. During these iterations, PGA operates pseudo-randomly and the impact of its selection and reproduction mechanisms, i.e. crossover and mutation, are not strong yet. As the evolution progresses, the performance gap between PGA and RS increases, reaching eventually the upper end of the *p* value and effect size ranges.

Considering these results, we conclude that EvoChecker instantiated with GA-based algorithm that uses a prohibitive selection strategy (PGA) significantly outperforms random search (RS) with large effect size in all adaptation steps and for all FX and UUV system variants. Thus, the use of evolutionary search-based approaches produces configuration with better quality.

RQ6 (Archive-strategy comparison) We analysed the system configurations selected by a GA using the archive updating strategies – prohibitive (PGA), complete recent (CRGA), limited recent (LRGA) and limited deep (LDGA) – in order to identify actionable insights. Note that these strategies are used on top of a basic GA and therefore have similar computation overheads (i.e. negligible CPU and memory use). Hence, the incurred overheads from the use of these strategies are not discussed further. In the interest of conciseness, we show a subset of these adaptation steps; similar reasoning applies to the other steps. Table 13 shows an excerpt of the pairwise comparisons carried out to check for significant difference and, when the difference exists, its effect size in parenthesis.

First, for change C1, i.e. the starting state of the examined systems (not shown in Table 13), and for all FX and UUV variants, all examined archive updating strategies identified configurations of comparable quality. No statistical difference was detected in any evolution stage for this event. Since all algorithms used a randomly generated initial population for change C1, this observation was not surprising.

Second, we found that GA variants using the archive (LRGA, CRGA, LDGA) performed significantly better than PGA for changes C2–C12 in FX and for most events

Table 13 Pairwise comparison of archive selection strategies for various stages of changes C4 and C11 of the FX variants showing the significantly better strategy and effect size (in parenthesis)

Strategies	C4				C11			
	1000	2000	3000	4000	1000	2000	3000	4000
FX_Small								
RS vs PGA	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)
PGA vs LRGA	LRGA(L)	LRGA(M)	LRGA(M)	LRGA(S)	LRGA(L)	LRGA(L)	LRGA(L)	LRGA(M)
PGA vs CRGA	CRGA(L)	CRGA(S)	–	–	–	–	–	–
LRGA vs CRGA	–	–	–	–	LRGA(L)	LRGA(L)	LRGA(L)	LRGA(L)
FX_Medium								
RS vs PGA	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)
PGA vs LRGA	LRGA(L)	LRGA(M)	LRGA(S)	–	LRGA(L)	LRGA(M)	LRGA(S)	–
PGA vs CRGA	CRGA(M)	CRGA(S)	–	–	–	–	–	–
LRGA vs CRGA	LRGA(S)	LRGA(S)	LRGA(S)	LRGA(S)	–	–	–	–

Key: S = Small, M = Medium, L = Large

in UUV during the majority of the evolution stages. No comparison showed statistical significance in favour of PGA for any change or evolution stage. As expected, as the evolution progressed all the GA variants had the opportunity to refine their solutions and the performance gap between the algorithms decreased. More specifically, there was a distinct performance gap favouring LRGA, CRGA and LDGA at the early evolution stages (p value $\in$ [3.38E$-$5,1.67E$-$11]), while PGA was able to find configurations that achieve similar cost towards the end of evolution (p value $\approx$ 0.05 in some cases). Looking at change C4 in Fig. 10, for instance, PGA is significantly worse until the 2500th iteration (i.e., 50 generations), but it approaches the others after that.

Third, in changes with statistical difference between PGA and the other variants, we observed a similar declining trend regarding the effect size. At the beginning of the evolution, the effect size is mostly large ([0.69.0.88] and [0.77,1.0] for UUV and FX, respectively), at the intermediate stages it changes to medium/small before it becomes small/negligible towards the end. Given these observations, we can state that using an archive updating strategy to select configurations from the archive and seed the initial population produces better configurations and faster, compared to a prohibitive strategy that ignores the archive. Given sufficient time, however, PGA will potentially catch up. Thus, archive-based GA variants are useful in the frequently encountered situations where the reconfiguration time and/or computation resources are limited.

Fourth, the archive-based GA variants (LRGA, CRGA, LDGA) identified configurations of similar quality to each other, demonstrating effective use of the archive. The post-hoc analysis, however, showed a performance difference between the three variants. In particular, we obtained statistically significant results in favour of LDGA against LRGA in 202 out of 500 tests (40.4%). For most changes, this difference concerned the first few evolution stages; after that LRGA performed similarly (e.g., C7 and C11 in Fig. 10). Furthermore, CRGA failed to produce better configurations than LDGA for any change and system variant, whereas it was marginally better than LRGA (6.2%) in changes that had similar characteristics to the preceding change. Like before, the performance difference involved only the initial stages. On the other hand, both LRGA and LDGA outperformed CRGA in a range of changes and evolution stages. We obtained statistical difference favouring LRGA and LDGA in 18.2% (91/500) and 47.8% (239/500) of these tests, respectively. This difference occurs because CRGA's population already identified good configurations and/or converged to a particular area in the fitness landscape. Since population variation is achieved only through crossover and mutation, CRGA finds difficulties to evolve the population in successive generations and produce better configurations. This leads to stagnation and early termination; see for instance changes C7 and C11 in Fig. 10 in which CRGA terminated in the 2500th and 1500th iteration, respectively. Therefore, reusing the final population from the current adaptation event does not offer a distinct advantage in producing better configurations over the other strategies. However, exploiting a subset of configurations from previous reconfiguration events (e.g, LDGA) could speed up the search significantly.

Finally, we note the inability of any archive-based GA variant to deal efficiently with disruptive change C13 affecting FX. For this event, about 50% of the available service implementations suffered a serious service degradation (cf. Table 12). For C13, we did not find any statistical significance between PGA, CRGA, LRGA and LDGA in any evolution stage in both FX system variants (Fig. 10). Moreover, at the initial evolution stages, CRGA had difficulties to select configurations that satisfy QoS requirements; its cost is close to the maximum value. Hence, when a disruptive change occurs, it does not have much impact which archive updating strategy is used. Using instead a population that is not biased towards a particular area (due to previous experience) would facilitate exploration of the fitness landscape.

We suppose that a hybrid approach which considers the types of changes in the system and its environment would be more effective. In this hybrid approach, some of the initial population would be derived from the archive (to exploit knowledge gained from previous reconfiguration events) and some would be randomly generated (to enable exploration of new events). The ratio between exploration and exploitation should be based on the expected ratio between small changes and radical changes in the environment.

7.3 Threats to validity

Several *construct*, *internal*, and *external* validity threats could affect the validity of the experiments conducted in this work.

Construct validity threats correspond to the methodology adopted when designing the experimental study and any underpinning assumptions. This includes any assumptions and simplifications made when modelling the DPM, FX and UUV systems. To mitigate this threat, the DPM system, model and QoS requirements are based on a validated real-world case study taken from the literature (Qiu et al. 2001; Sesic et al. 2008), which we are familiar with from our previous work (Calinescu and Kwiatkowska 2009). This is also the case for the UUV system, model and requirements (Calinescu et al. 2015; Gerasimou et al. 2014, 2017). For the FX system, the model and requirements were developed in close collaboration with a foreign exchange domain expert. Also, the environment changes cover a wide range of system scenarios that could cause service degradation and/or violation of QoS requirements, including minor changes and disruptive events.

Internal validity threats might be due to any bias introduced when establishing the causality between our findings and the evolutionary algorithms employed in our study. To mitigate this threat, we followed the established practice in search-based software engineering (Arcuri and Briand 2011; Harman et al. 2012b). In particular, we reported results over 30 independent runs of each experiment and used inferential statistical tests to check for significant difference in the performance of the algorithms. To this end, we evaluated whether the data conformed to the normal distribution using the Shapiro–Wilk test and used the non-parametric tests Mann–Whitney and Kruskal–Wallis to check for statistical significance. We also conducted a post-hoc analysis using Dunn's pairwise test. All these tests used a 95% confidence level; hence, the probability of committing a Type I error is 0.05, which is the recommended value in empirical studies in this area. Finally, we employed the Varga and Delaney's effect size (Vargha and Delaney 2000) measure to establish the magnitude of an improvement.

External validity threats might be due to the difficulty of representing a software system using the EvoChecker constructs (2)–(4), QoS attributes (5) constraints (6), optimisation objectives (7) and loss (10). We limit this threat by specifying the EvoChecker modelling language based on the modelling language of established probabilistic model checkers [PRISM (Kwiatkowska et al. 2011), Storm (Dehnert et al. 2017)]. Moreover, given the generality of the EvoChecker constructs (2)–(4), other probabilistic modelling languages [e.g., those of the model checkers MRMC (Katoen et al. 2011) and Ymer (Younes 2005)] can be naturally supported. Additionally, EvoChecker supports a wide range of probabilistic models and temporal logics (Table 2). We also examined various archive updating strategies, but other more sophisticated strategies can be developed. Finally, to further reduce the risk that EvoChecker might be difficult to use in practice, we validated it through application to several variants of three realistic software systems with diverse characteristics in terms of application domain, size, complexity and QoS requirements. Nevertheless, our findings are not conclusive for all types of software systems, and more experiments are needed to confirm the generality of the EvoChecker approach and tool.

8 Related work

The research underpinning EvoChecker spans the areas of probabilistic model checking (Kwiatkowska 2007) and search-based software engineering (SBSE) (Harman et al. 2012a, b). The closest work related to EvoChecker is Forejt et al. (2012), which uses policies of Markov decision processes (MDPs) to synthesise Pareto front approximations. Nevertheless, this approach requires fully specified MDPs and it is limited by the finite search spaces that can be encoded as MDP policies. Furthermore, the approach currently supports only up to three optimisation objectives and it is applicable only to a subset of probabilistic computation tree logic (i.e., reachability and expected total reward formulae). In contrast, EvoChecker deals with probabilistic model templates that can encode infinite search spaces (due to evolvable double parameters and distributions) and supports all types of models and temporal logics from Table 2. Additionally, the EAs used by EvoChecker can generate Pareto front approximations for more than three optimisation objectives.

Search-based techniques (Harman et al. 2012b) have been successfully used in areas ranging from project management (Ferrucci et al. 2013; Gerasimou et al. 2012; Ren et al. 2011; Stylianou et al. 2012), effort estimation (Minku and Yao 2013) and testing (Andrews et al. 2011; Fraser and Arcuri 2013) to software repair and evolution (Canfora et al. 2005; Praditwong et al. 2011), software product lines (Harman et al. 2014a; Sayyad et al. 2013) and software architectures (Martens et al. 2010). A general survey on using SBSE within software engineering is available in Harman et al. (2012a), while the comprehensive survey from Aleti et al. (2013) focuses on the application of SBSE to software architecture design. However, the application of SBSE to model checking is limited and related research focuses on non-probabilistic models and design-time activities (Harman et al. 2012a). In Johnson (2007), Katz and

Peled (2013), genetic evolution is applied to synthesise model checking specifications, while in Alba and Chicano (2007), Alba and Chicano (2008) ant colony optimisation is used for generating counterexamples in medium–large stochastic models.

Despite the increasing interest in dynamic adaptive search-based techniques, their use in reconfiguring software systems based on QoS requirements is rather limited (Harman et al. 2012b). Harman et al. (2014b) report that a combination of machine learning and search-based techniques will enable software systems to adapt while providing service. Early work in this direction is presented in Coker et al. (2015). The only other approach that we are aware of in this area is Plato (Ramirez et al. 2011), which employs genetic algorithms in the decision-making process of a self-adaptive system and generates new configurations that balance functional and non-functional requirements. However, Plato does not consider environment or system stochasticity, as EvoChecker does with its probabilistic model template. Also, Plato does not employ any knowledge acquired during system operation to speed up the search, as EvoChecker at runtime does with its archive and archive updating strategies.

Our work is also related to research that explores ways to incorporate problem specific knowledge into an evolutionary algorithm through seeding its initial population (Grefenstette 1987). As advocated by recent research (Kazimipour et al. 2014), if prior knowledge is available or can be generated with reasonable computational effort, effective seeding may yield better quality solutions and lead to faster convergence. The effect of various seeding options (between 25%-100% of the population size) was studied in Oman and Cunningham (2001) for the travelling salesman and the job-shop scheduling problems. The authors reported that seeding produced most of the time significantly better solutions than no seeding, although a 100% seed did not always generate better results. In the domain of search-based software testing, Fraser and Arcuri (2012) assessed the effectiveness of various seeding strategies for generating test cases in object-oriented languages. They found that the impact of effective seeding is heavier during the early stages of the search, while weaker seeding strategies or no seeding will perform similarly from intermediate stages onwards. These observations validate our findings regarding the impact of the archive updating strategies (13)–(16).

EvoChecker also partially overlaps with research on *stochastic controller synthesis*, in which formally verified stochastic controllers are used to disable certain (controllable) system behaviours or to vary the probability with which these behaviours occur. Draeger et al. (2014) propose the synthesis of a multi-strategy controller that enables a set of actions at any state and which is optimally permissive with respect to a penalty function. Irrespective of the action carried out, the controller guarantees compliance with system requirements. However, unlike our work which covers the full PCTL and CSL, Draeger et al. (2014) focuses only on probabilistic reachability and expected total rewards.

Moreno et al. (2015) propose a controller synthesis approach by combining lookahead and latency awareness. Lookahead projects the expected system evolution over a limited horizon, while latency awareness considers the time between making and realising an adaptation decision. The synthesised controller performs a limited lookahead, but it ignores any previous knowledge and thus fails to support incremental synthesis.

A complementary approach to controller synthesis is proposed by Ulusoy et al. (2014). The key idea is based on partitioning the synthesis task into several steps and then to refine the controller incrementally. Initially, the technique considers a high-level system model and adds extra details as the synthesis progresses, until a termination criterion is met (e.g., exhausted computational resources). Unlike EvoChecker, though, which supports a variety of specification logics (Table 2), this approach supports specifications defined only in linear temporal logic.

Another research area related to EvoChecker is *probabilistic model repair*, which automatically "repairs" a Markov model that violates a probabilistic temporal logic formula (Bartocci et al. 2011). Given this situation, probabilistic model repair involves modifying the transition probabilities of the model to generate new models that satisfy the formula and are "close" to the original model (Bartocci et al. 2011; Chen et al. 2013). The proposed approaches have limited applicability since they consider only a single temporal logic formula and modify only the transition probabilities of the original model. In contrast, EvoChecker operates with multiple formulae and uses multiobjective optimisation to evolve a set of probabilistic models that approximates the Pareto-optimal model set corresponding to these formulae. Furthermore, we replicated the results of the IPv4 Zeroconf Protocol (Bartocci et al. 2011) and Network Virus Infection (Chen et al. 2013) case studies, demonstrating that EvoChecker subsumes the capabilities of probabilistic model repair.

The concept of *model repair* has been also explored for non-probabilistic models (Bonakdarpour and Kulkarni 2012; Buccafurri et al. 1999; Zhang and Ding 2008). Similarly to probabilistic model repair, these approaches can handle a single type of model and can repair a single temporal logic formula (Carrillo and Rosenblueth 2014; Chatzieleftheriou et al. 2012; Martinez-Araiza and Lopez-Mellado 2014).

Probabilistic model checking at runtime involves the continual verification of Markov models to support the analyse and plan stages of the MAPE-K control loop (Kephart and Chess 2003) of self-adaptive systems (Calinescu and Kwiatkowska 2009; Calinescu et al. 2011, 2012; Epifani et al. 2009). Recent research aims to tackle the state explosion problem (Baier et al. 1999) and improve the efficiency of runtime probabilistic model checking. More specifically, *compositional* methods use assume-guarantee reasoning to verify component-based systems one component at a time (Kwiatkowska et al. 2010), *incremental* methods establish the current verification results using results obtained in previous verification runs (Johnson et al. 2013; Kwiatkowska et al. 2011; Meedeniya and Grunske 2010), and *pre-computation-based* methods transform temporal logic formulae into easy-to-evaluate algebraic expressions (Filieri et al. 2016). For a detailed overview, see Calinescu et al. (2017d) and Chapter 2 in Gerasimou (2017). These methods reduce the probabilistic model checking overheads but they are only applicable to discrete-time models, can support only the simplest structural changes in the verified model, and make limiting assumptions (e.g., that the model can be partitioned into strongly connected components each of which is much smaller than the original model). Our work, on the other hand, is model and property agnostic.

In recent work, we integrated probabilistic model checking with established efficiency-improvement methods from other software engineering areas, i.e., caching, limited lookahead and nearly-optimal reconfiguration (Gerasimou et al. 2014).

Although these methods reduce the overheads of PMC at runtime, they need to perform (in the worst case) an exhaustive search through the entire configuration space when selecting new configurations for self-adaptive systems. EvoChecker does not suffer from this limitation and can search efficiently through configuration spaces that are too large for exhaustive search.

Finally, the approach to developing distributed self-adaptive systems we introduced in Calinescu et al. (2015) operates with reduced overheads by only performing PMC at component level, but relies on analysing all possible component configurations at runtime. As such, this approach cannot handle very large (component) configuration spaces, which is the challenge addressed by the runtime variant of EvoChecker.

9 Conclusions

The synthesis of probabilistic models is key for the cost-effective engineering of software. Nevertheless, techniques like exhaustive search, trial-and-error and simple heuristics are insufficient, as they cannot deal with large configuration spaces and produce models that may not represent satisfactory tradeoffs between multiple QoS requirements.

Our EvoChecker search-based software engineering approach automates this process and improves its outcome. EvoChecker can be used at design time to identify suitable architectures and parameter values for a software system under design. The design-time use of EvoChecker employs multi-objective evolutionary algorithms to generate a set of probabilistic models that closely approximates the Pareto-optimal model set associated with the QoS requirements and the corresponding approximate QoS Pareto front. EvoChecker can be also used at runtime to support the reconfiguration of a self-adaptive software system. This involves the incremental synthesis of probabilistic models using single-objective evolutionary algorithms. When used at runtime, EvoChecker maintains an archive of configurations from recent adaptations, and uses the archived historical configurations to seed the initial population of a new search and, thus, to identify effective new configurations faster.

We evaluated EvoChecker on three case studies from the domains of unmanned underwater vehicles (Gerasimou et al. 2014), dynamic power-management (Qiu et al. 2001) and service-based systems. Our results indicate that the design-time use of EvoChecker can generate Pareto-optimal approximation sets and help system experts to make informed decisions (e.g., identify "point of diminishing returns", find architectures and configuration parameters that have significant impact on QoS requirements). We also found that NSGA-II (Deb et al. 2002) and SPEA2 (Zitzler et al. 2001) performed equally good in the considered case studies and for all analysed quality indicators (i.e., hypervolume, epsilon, and inverted generational distance). Hence, any of these algorithms is a good choice for instantiating the EvoChecker at design time. For the use of EvoChecker at runtime, we observed that combining the external archive with a suitable updating strategy helps EvoChecker to identify effective configurations much faster than EvoChecker instances that do not use the archive. Thus, using an archive to store configurations from recent adaptations and seeding a new population with archived historical configurations can speed up the search, especially when similar environment states are encountered often.

Our planned future work on EvoChecker is twofold. First, we aim to enhance the capabilities of the approach. To this end, we will extend the range of modelling formalisms and verification logics that EvoChecker can support by exploiting other established quantitative model checkers such as UPPAAL (Behrmann et al. 2006) and MRMC (Katoen et al. 2011). We also plan to integrate the EvoChecker approach with our recent work on runtime probabilistic model checking (Calinescu et al. 2017b, 2015, 2017; Gerasimou et al. 2014). Furthermore, we intend to enhance the EvoChecker synthesis capabilities by supporting other evolutionary and natured-inspired optimisation algorithms like evolutionary strategies, particle swarm optimisation and ant-colony optimisation (Coello et al. 2006). Finally, adapting techniques that analyse the fitness landscape of the induced search space (Aleti et al. 2017) is another possible extension for EvoChecker.

Second, we intend to evaluate the use of EvoChecker in other domains and by other projects, in order to extract lessons, insights and best practices from the practical application of the approach to real systems. This area of future work was made possible by the recent adoption of our approach within several projects carried out by teams that include researchers and engineers not involved in the EvoChecker development. These projects have used or will use EvoChecker to devise safe reinforcement learning solutions (Mason et al. 2017, 2018), to synthesise robust designs for software-based systems (Calinescu et al. 2017b, c), and to suggest safe evacuation routes for communities affected by adverse events such as natural disasters. This will show how easy it is to define and validate EvoChecker models and requirements in real applications, allowing us to improve the usability of the approach.

Acknowledgements This paper presents research partly sponsored by UK MOD. The information contained in it should not be interpreted as representing the views of the UK MOD, nor should it be assumed it reflects any current or future UK MOD policy.

References

Alba, E., Chicano, F.: Finding safety errors with ACO. In: 9th International Conference on Genetic and Evolutionary Computation (GECCO'07), pp. 1066–1073 (2007)

Alba, E., Chicano, F.: Searching for liveness property violations in concurrent systems with ACO. In: 10th International Conference on Genetic and Evolutionary Computation (GECCO'08), pp. 1727–1734 (2008)

Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., Meedeniya, I.: Software architecture optimization methods: a systematic literature review. IEEE Trans. Softw. Eng. **39**(5), 658–683 (2013)

Aleti, A., Moser, I., Grunske, L.: Analysing the fitness landscape of search-based software testing problems. Autom. Softw. Eng. **24**(3), 603–621 (2017)

Alur, R., Henzinger, T.A.: Reactive modules. Form. Methods Syst. Des. **15**(1), 7–48 (1999)

Alur, R., Henzinger, T.A., Vardi, M.Y.: Theory in practice for system design and verification. ACM SIGLOG News **2**(1), 46–51 (2015)

Andova, S., Hermanns, H., Katoen, J.P.: Discrete-time rewards model-checked. In: FORMATS 2003, vol. 2791, pp. 88–104 (2004)

Andrews, J., Menzies, T., Li, F.: Genetic algorithms for randomized unit testing. IEEE Trans. Softw. Eng. **37**(1), 80–94 (2011)

Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: 33rd International Conference on Software Engineering (ICSE'11), pp. 1–10 (2011)

Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model checking continuous-time Markov chains. ACM Trans. Comput. Log. **1**(1), 162–170 (2000)

Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.P.: Performance evaluation and model checking join forces. Commun. ACM **53**(9), 76–85 (2010)

Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)

Baier, C., Katoen, J.P., Hermanns, H.: Approximate symbolic model checking of continuous-time Markov chains. In: 10th International Conference on Concurrency Theory (CONCUR'99), pp. 146–161 (1999)

Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and run-time. In: Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER'10), pp. 17–22 (2010)

Bartocci, E., Grosu, R., Katsaros, P., Ramakrishnan, C., Smolka, S.: Model repair for probabilistic systems. In: 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11), vol. 6605, pp. 326–340. Springer (2011)

Behrmann, G., David, A., Larsen, K.G., Hakansson, J., Petterson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: 3rd International Conference on the Quantitative Evaluation of Systems (QEST'06), pp. 125–126 (2006)

Bianco, A., Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Foundations of Software Technology and Theoretical Computer Science, vol. 1026, pp. 499–513. Springer (1995)

Bonakdarpour, B., Kulkarni, S.S.: Automated model repair for distributed programs. ACM SIGACT News **43**(2), 85–107 (2012)

Buccafurri, F., Eiter, T., Gottlob, G., Leone, N.: Enhancing model checking in verification by AI techniques. Artif. Intell. **112**, 57–104 (1999)

Calinescu, R., Autili, M., Cmara, J., Di Marco, A., Gerasimou, S., Inverardi, P., Perucci, A., Jansen, N., Katoen, J.P., Kwiatkowska, M., Mengshoel, O., Spalazzese, R., Tivoli, M.: Synthesis and Verification of Self-aware Computing Systems, pp. 337–373. Springer (2017)

Calinescu, R., Ceska, M., Gerasimou, S., Kwiatkowska, M., Paoletti, N.: Designing robust software systems through parametric Markov chain synthesis. In: 2017 IEEE International Conference on Software Architecture (ICSA), pp. 131–140 (2017)

Calinescu, R., Ceska, M., Gerasimou, S., Kwiatkowska, M., Paoletti, N.: RODES: A robust-design synthesis tool for probabilistic systems. In: 14th International Conference on Quantitative Evaluation of Systems (QEST), pp. 304–308 (2017)

Calinescu, R., Gerasimou, S., Banks, A.: Self-adaptive software with decentralised control loops. In: 18th International Conference on Fundamental Approaches to Software Engineering (FASE'15), pp. 235–251 (2015)

Calinescu, R., Gerasimou, S., Johnson, K., Paterson, C.: Using runtime quantitative verification to provide assurance evidence for self-adaptive software. In: Software Engineering for Self-Adaptive Systems III. Assurances, pp. 223–248. Springer (2017)

Calinescu, R., Ghezzi, C., Johnson, K., Pezzé, M., Rafiq, Y., Tamburrelli, G.: Formal verification with confidence intervals to establish quality of service properties of software systems. IEEE Trans. Reliab. **65**(1), 107–125 (2016)

Calinescu, R., Ghezzi, C., Kwiatkowska, M., Mirandola, R.: Self-adaptive software needs quantitative verification at runtime. Commun. ACM **55**(9), 69–77 (2012)

Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic QoS management and optimization in service-based systems. IEEE Trans. Softw. Eng. **37**(3), 387–409 (2011)

Calinescu, R., Kwiatkowska, M.: Using quantitative analysis to implement autonomic IT systems. In: 31st International Conference on Software Engineering (ICSE'09), pp. 100–110 (2009)

Calinescu, R., Weyns, D., Gerasimou, S., Iftikhar, M.U., Habli, I., Kelly, T.: Engineering trustworthy self-adaptive software with dynamic assurance cases. IEEE Trans. Softw. Eng. **PP**(99), 1–31 (2017)

Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: An approach for QoS-aware service composition based on genetic algorithms. In: 7th International Conference on Genetic and Evolutionary Computation (GECCO'05), pp. 1069–1075 (2005)

Carrillo, M., Rosenblueth, D.A.: CTL update of Kripke models through protections. Artif. Intell. **211**, 51–74 (2014)

Chatzieleftheriou, G., Bonakdarpour, B., Smolka, S.A., Katsaros, P.: Abstract model repair. In: NASA Formal Methods, pp. 341–355. Springer (2012)

Chen, T., Hahn, E.M., Han, T., Kwiatkowska, M., Qu, H., Zhang, L.: Model repair for Markov decision processes. In: 7th International Symposium on Theoretical Aspects of Software Engineering (TASE'13), pp. 85–92 (2013)

Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)

Coello, C.A.C., Lamont, G.B., Veldhuizen, D.A.V.: Evolutionary Algorithms for Solving Multi-objective Problems. Springer, Berlin (2006)

Coker, Z., Garlan, D., Le Goues, C.: SASS: self-adaptation using stochastic search. In: 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'15), pp. 168–174 (2015)

Damm, L.O., Lundberg, L.: Company-wide implementation of metrics for early software fault detection. In: ICSE, pp. 560–570 (2007)

Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)

Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A Storm is coming: a modern probabilistic model checker. In: 29th International Conference on Computer Aided Verification, pp. 592–600 (2017)

Draeger, K., Forejt, V., Kwiatkowska, M., Parker, D., Ujma, M.: Permissive controller synthesis for probabilistic systems. In: 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14), vol. 8413, pp. 531–546 (2014)

Durillo, J.J., Nebro, A.J.: jMetal: a Java framework for multi-objective optimization. Adv. Eng. Softw. **42**, 760–771 (2011)

Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: 31st International Conference on Software Engineering (ICSE'09), pp. 111–121 (2009)

Ferrucci, F., Harman, M., Ren, J., Sarro, F.: Not going to take this anymore: multi-objective overtime planning for software engineering projects. In: 35th International Conference on Software Engineering (ICSE'13), pp. 462–471 (2013)

Filieri, A., Tamburrelli, G., Ghezzi, C.: Supporting self-adaptation via quantitative verification and sensitivity analysis at run time. Trans. Softw. Eng. **42**(1), 75–99 (2016)

Fonseca, C.M., Fleming, P.J.: Multiobjective optimization. In: Handbook of Evolutionary Computation, vol. 1, pp. C4.5:1–C4.5:9 (1997)

Forejt, V., Kwiatkowska, M., Parker, D.: Pareto curves for probabilistic model checking. In: 10th International Symposium on Automated Technology for Verification and Analysis (ATVA'12), vol. 7561, pp. 317–332 (2012)

Fraser, G., Arcuri, A.: The seed is strong: Seeding strategies in search-based software testing. In: Fifth International Conference on Software Testing, Verification and Validation (ICST'12), pp. 121–130 (2012)

Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Trans. Softw. Eng. **39**(2), 276–291 (2013)

Gerasimou, S.: Runtime quantitative verification of self-adaptive systems. Ph.D. thesis, University of York, York, UK (2017)

Gerasimou, S., Calinescu, R., Banks, A.: Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration. In: 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14), pp. 115–124 (2014)

Gerasimou, S., Calinescu, R., Shevtsov, S., Weyns, D.: Undersea: an exemplar for engineering self-adaptive unmanned underwater vehicles. In: 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'17), pp. 83–89 (2017)

Gerasimou, S., Stylianou, C., Andreou, A.S.: An investigation of optimal project scheduling and team staffing in software development using particle swarm optimization. In: 14th International Conference on Enterprise Information Systems (ICEIS'12), pp. 168–171 (2012)

Gerasimou, S., Tamburrelli, G., Calinescu, R.: Search-based synthesis of probabilistic models for quality-of-service software engineering. In: 30th International Conference on Automated Software Engineering (ASE'15), pp. 319–330 (2015)

Ghezzi, C.: Evolution, adaptation, and the quest for incrementality. In: Large-Scale Complex IT Systems. Development, Operation and Management, vol. 7539, pp. 369–379 (2012)

Grefenstette, J.J.: Incorporating problem specific knowledge into genetic algorithms. Genetic algorithms and simulated annealing, pp. 42–60 (1987)

Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Form. Asp. Comput. **6**(5), 512–535 (1994)

Harman, M., Jia, Y., Krinke, J., Langdon, W.B., Petke, J., Zhang, Y.: Search based software engineering for software product line engineering: a survey and directions for future work. In: 18th International Software Product Line Conference, pp. 5–18 (2014)

Harman, M., Jia, Y., Langdon, W.B., Petke, J., Moghadam, I.H., Yoo, S., Wu, F.: Genetic improvement for adaptive software engineering. In: 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14), pp. 1–4 (2014)

Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: trends, techniques and applications. ACM Comput. Surv. **45**(1), 11:1–11:61 (2012a)

Harman, M., McMinn, P., de Souza, J., Yoo, S.: Search based software engineering: techniques, taxonomy, tutorial. In: Empirical Software Engineering and Verification, vol. 7007, pp. 1–59. Springer (2012b)

Helwig, S., Wanka, R.: Theoretical analysis of initial particle swarm behavior. In: 10th International Conference on Parallel Problem Solving from Nature (PPSN'08), pp. 889–898 (2008)

Johnson, C.: Genetic programming with fitness based on model checking. In: Genetic Programming, vol. 4445, pp. 114–124. Springer (2007)

Johnson, K., Calinescu, R., Kikuchi, S.: An incremental verification framework for component-based software systems. In: 16th International Symposium on Component-Based Software Engineering (CBSE'13), pp. 33–42 (2013)

Katoen, J.P., Khattri, M., Zapreev, I.S.: A Markov reward model checker. In: Quantitative Evaluation of Systems (QEST'05), pp. 243–244 (2005)

Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. Perform. Eval. **68**(2), 90–104 (2011)

Katz, G., Peled, D.: Synthesis of parametric programs using genetic programming and model checking. In: 15th International Workshop on Verification of Infinite-State Systems (INFINITY'13), pp. 70–84 (2013)

Kazimipour, B., Li, X., Qin, A.K.: A review of population initialization techniques for evolutionary algorithms. In: IEEE Congress on Evolutionary Computation (CEC'14), pp. 2585–2592 (2014)

Kephart, J., Chess, D.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)

Kwiatkowska, M.: Quantitative verification: models, techniques and tools. In: 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers (ESEC-FSE'07), pp. 449–458 (2007)

Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07), pp. 220–270. Springer (2007)

Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: 23rd International Conference on Computer Aided Verification (CAV'11), pp. 585–591 (2011)

Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Assume-guarantee verification for probabilistic systems. In: 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10), vol. 6015, pp. 23–37. Springer (2010)

Kwiatkowska, M., Parker, D., Qu, H.: Incremental quantitative verification for Markov decision processes. In: 41st International Conference on Dependable Systems Networks (DSN'11), pp. 359–370 (2011)

Martens, A., Koziolek, H., Becker, S., Reussner, R.: Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In: First Joint WOSP/SIPEW International Conference on Performance Engineering, WOSP/SIPEW '10, pp. 105–116. ACM (2010)

Martinez-Araiza, U., Lopez-Mellado, E.: A CTL model repair method for Petri Nets. In: World Automation Congress (WAC'14), pp. 654–659 (2014)

Mason, G., Calinescu, R., Kudenko, D., Banks, A.: Assured reinforcement learning with formally verified abstract policies. In: 9th International Conference on Agents and Artificial Intelligence (ICAART'17), vol. 2, pp. 105–117. SciTe Press (2017)

Mason, G., Calinescu, R., Kudenko, D., Banks, A.: Assurance in reinforcement learning using quantitative verification. In: Advances in Hybridization of Intelligent Methods: Models, Systems and Applications, pp. 71–96. Springer (2018)

Meedeniya, I., Grunske, L.: An efficient method for architecture-based reliability evaluation for evolving systems with changing parameters. In: 21st International Symposium on Software Reliability Engineering (ISSRE'10), pp. 229–238 (2010)

Minku, L.L., Yao, X.: Software effort estimation as a multiobjective learning problem. Trans. Softw. Eng. Methodol. **22**(4), 35:1–35:32 (2013)

Moreno, G.A., Cámara, J., Garlan, D., Schmerl, B.: Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In: 10th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15), pp. 1–12 (2015)

Nebro, A.J., Durillo, J.J., Luna, F., Dorronsoro, B., Alba, E.: MOCell: a cellular genetic algorithm for multiobjective optimization. Int. J. Intell. Syst. **24**(7), 726–746 (2009)

Oman, S., Cunningham, P.: Using case retrieval to seed genetic algorithms. Int. J. Comput. Intell. Appl. **01**(01), 71–82 (2001)

Pnueli, A.: In transition from global to modular temporal reasoning about programs. Log. Models Concurr. Syst. **13**, 123–144 (1985)

Praditwong, K., Harman, M., Yao, X.: Software module clustering as a multi-objective search problem. IEEE Trans. Softw. Eng. **37**(2), 264–282 (2011)

Qiu, Q., Qu, Q., Pedram, M.: Stochastic modeling of a power-managed system-construction and optimization. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **20**(10), 1200–1217 (2001)

Ramirez, A., Knoester, D., Cheng, B., McKinley, P.: Plato: a genetic algorithm approach to run-time reconfiguration in autonomic computing systems. Clust. Comput. **14**(3), 229–244 (2011)

Ren, J., Harman, M., Di Penta, M.: Cooperative co-evolutionary optimization of software project staff assignments and job scheduling. In: 3rd International Symposium on Search Based Software Engineering (SSBSE'11), vol. 6956, pp. 127–141. Springer (2011)

Salehie, M., Tahvildari, L.: Self-adaptive software: landscape and research challenges. ACM Trans. Auton. Adapt. Syst. **4**(2), 14:1–14:42 (2009)

Sayyad, A., Ingram, J., Menzies, T., Ammar, H.: Scalable product line configuration: A straw to break the camel's back. In: 28th International Conference on Automated Software Engineering (ASE'13), pp. 465–474 (2013)

Sesic, A., Dautovic, S., Malbasa, V.: Dynamic power management of a system with a two-priority request queue using probabilistic-model checking. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **27**(2), 403–407 (2008)

Stylianou, C., Gerasimou, S., Andreou, A.: A novel prototype tool for intelligent software project scheduling and staffing enhanced with personality factors. In: 24th International Conference on Tools with Artificial Intelligence (ICTAI'12), pp. 277–284 (2012)

Ulusoy, A., Wongpiromsarn, T., Belta, C.: Incremental controller synthesis in probabilistic environments with temporal logic constraints. Int. J. Robot. Res. **33**(8), 1130–1144 (2014)

Van Veldhuizen, D.A.: Multiobjective evolutionary algorithms: classifications, analyses, and new innovations. Ph.D. thesis (1999)

Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J. Educ. Behav. Stat. **25**(2), 101–132 (2000)

Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: practice and experience. ACM Comput. Surv. **41**(4), 19:1–19:36 (2009)

Younes, H.L.S.: Ymer: A statistical model checker. In: 17th International Conference on Computer Aided Verification (CAV'05), vol. 3576, pp. 429–433. Springer (2005)

Zhang, Y., Ding, Y.: CTL model update for system modifications. J. Artif. Intell. Res. (JAIR) **31**, 113–155 (2008)

Zitzler, E., Brockhoff, D., Thiele, L.: The hypervolume indicator revisited: on the design of Pareto-compliant indicators via weighted integration. In: 4th International Conference on Evolutionary Multi-criterion Optimization (EMO'07), pp. 862–876 (2007)

Zitzler, E., Knowles, J., Thiele, L.: Quality assessment of Pareto set approximations. In: Multiobjective Optimization, vol. 5252, pp. 373–404. Springer (2008)

Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the strength Pareto evolutionary algorithm. In: Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems (EUROGEN'01), pp. 95–100 (2001)

Zitzler, E., Thiele, L.: Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. IEEE Trans. Evol. Comput. **3**(4), 257–271 (1999)

Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C., da Fonseca, V.: Performance assessment of multiobjective optimizers: an analysis and review. IEEE Trans. Evol. Comput. **7**(2), 117–132 (2003)