Search-Based Synthesis of Probabilistic Models for Quality-of-Service Software Engineering

Simos Gerasimou
Dept. of Computer Science
University of York, UK
simos@cs.york.ac.uk

Giordano Tamburrelli
Dept. of Computer Sciences
Vrije Universiteit, Netherlands
g.tamburrelli@vu.nl

Radu Calinescu
Dept. of Computer Science
University of York, UK
radu.calinescu@york.ac.uk

Abstract—**The formal verification of finite-state probabilistic models supports the engineering of software with strict quality-of-service (QoS) requirements. However, its use in software design is currently a tedious process of manual multiobjective optimisation. Software designers must build and verify probabilistic models for numerous alternative architectures and instantiations of the system parameters. When successful, they end up with feasible but often suboptimal models. The EvoChecker search-based software engineering approach and tool introduced in our paper employ multiobjective optimisation genetic algorithms to automate this process and considerably improve its outcome. We evaluate EvoChecker for six variants of two software systems from the domains of dynamic power management and foreign exchange trading. These systems are characterised by different types of design parameters and QoS requirements, and their design spaces comprise between 2E+14 and 7.22E+86 relevant alternative designs. Our results provide strong evidence that EvoChecker significantly outperforms the current practice and yields actionable insights for software designers.**

I. Introduction

The design of modern software systems requires tools capable of assessing the quality and dependability of the solution under design as early as possible in the engineering process. The late discovery of defects leads to suboptimal operation of the delivered systems, high maintenance costs, and negative impact on their users [14], [26]. This is particularly relevant for business-critical and safety-critical applications, where failures may cause financial loss or loss of human life.

Model Checking (MC) [8], [25] is among the most effective means for developing software for critical applications [61]. Given a finite-state transition model of a concurrent system, MC automatically verifies if the model satisfies requirements formally expressed in propositional temporal logic. MC can be exploited during system design, to devise models that meet the requirements of a software system, and are then used as a basis for its implementation. Alternatively, software engineers can build models of existing systems, and use MC to check their compliance with requirements. Also, a variant of MC called *probabilistic model checking* can verify if software systems meet their reliability, performance and other quality-of-service (QoS) requirements [46]. Probabilistic MC verifies models with state transitions annotated by probabilities or transition rates (e.g., discrete and continuous-time Markov chains, and Markov decision processes), and operates with QoS requirements specified in probabilistic variants of temporal logics.

This paper is concerned with the use of probabilistic MC at design time to identify models that satisfy the QoS requirements of a software system. Currently, this is a tedious trial-and-error process in which software designers must build and verify probabilistic models for numerous alternative architectures and instantiations of the system parameters. Even when this manual attempt at multiobjective optimisation is successful, the models produced are often suboptimal in terms of their tradeoffs between reliability, performance, cost and other QoS requirements. The EvoChecker search-based software engineering approach introduced in our paper automates this process and considerably improves its outcome.

To this end, EvoChecker synthesises (a) a set of probabilistic models that closely approximates the Pareto-optimal model set associated with the QoS requirements of a software system; and (b) the corresponding approximate QoS Pareto front. This provides insight into the tradeoffs between multiple QoS requirements, enabling software designers to make informed decisions about the architecture and parameters of their systems. To achieve this, EvoChecker employs established multiobjective optimisation genetic algorithms (MOGAs) and a fitness function that captures the satisfaction of the QoS requirements. EvoChecker uses as input:

- a *probabilistic model template* that encodes alternative architectures and parameter ranges for the system;
- a set of QoS requirements specifying both *constraints* (e.g., "*At least 95% of the requests must be processed within 200ms*") and *optimisation objectives* (e.g., "*The system should operate with minimal energy consumption*").

EvoChecker uses the probabilistic model checker PRISM [47] for its verification steps. Accordingly, the probabilistic model template is expressed in an extension of the PRISM high-level modelling language, and EvoChecker handles all types of probabilistic models and probabilistic temporal logics supported by PRISM and shown in Table I.

The main contributions of our paper include:

1. The EvoChecker extension of the PRISM modelling language, and the EvoChecker search-based model synthesis approach, which we describe in Section III, and we illustrate using a running example introduced in Section II.
2. The definition of the *probabilistic model synthesis problem* (Section III-C).
3. The open-source EvoChecker tool presented in Section IV, which we make freely available on our project webpage.
4. An extensive evaluation of the EvoChecker approach and prototype tool in two case studies drawn from different application domains, which we summarise in Section V.

Our evaluation based on established Pareto-front quality indicators (i.e., unary epsilon [68], hypervolume [67] and inverted generational distance [60]) shows that EvoChecker significantly outperforms the current "trial and error" practice, makes effective use of state-of-the-art MOGAs such as NSGA-II [27], SPEA2 [51] and MOCell [66], and provides actionable insights for software designers.

To the best of our knowledge, EvoChecker is the first approach that uses search-based software engineering to synthesise approximate Pareto-optimal sets of probabilistic models for the QoS requirements of software systems. The only related work we are aware of is discussed in Section VI, and belongs to the area of *model repair* [10], [16]. However, model repair for probabilistic systems focuses on modifying an existing probabilistic model, only at transition probability level, and only to satisfy a single temporal logic property that is violated by the original model. In contrast, EvoChecker handles multiple QoS requirements and uses multiobjective optimisation to derive an approximate Pareto-optimal set of probabilistic models and its associated Pareto front given a probabilistic model template.

II. Running example

We will illustrate the EvoChecker components and operation using a software-controlled dynamic power management (DPM) system adapted from [55], [58]. As shown in Fig. 1, the system consists of a *service provider* that handles requests generated by a *service requester* and stored in two *request queues* of different priorities. The service provider has four states associated with different power usage, i.e., *busy, idle, standby* and *sleep*. Fig. 1 depicts the power usage of each state (in watts), the possible transitions between states, and the energy consumed by each transition (in joules). These values are taken from [55], and correspond to a Fujitsu disk drive.

When the service provider is in the *busy* state, it processes requests as follows. If the high-priority queue contains $q_H > 0$ requests, then a high-priority request is processed. Otherwise, if the low-priority queue contains requests (i.e., if $q_L > 0$), a low-priority request is handled. After handling the last request (i.e., when both queues become empty), the service provider automatically transitions to the *idle* state. The transitions from *idle* to *busy* are also automatic, and occur whenever the empty-queue DPM system receives a new request. In contrast, all the other transitions are controlled by a software *power manager* that aims to reduce power use while maintaining an acceptable service level for the system. We use the real values from [55] for the state transition times (Table II) and the request

TABLE I: Types of 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 [13], [36]
^{b}Linear Temporal Logic [53]
^{c}PCTL* is a superset of PCTL and LTL
^{d}Continuous Stochastic Logic [7], [9]

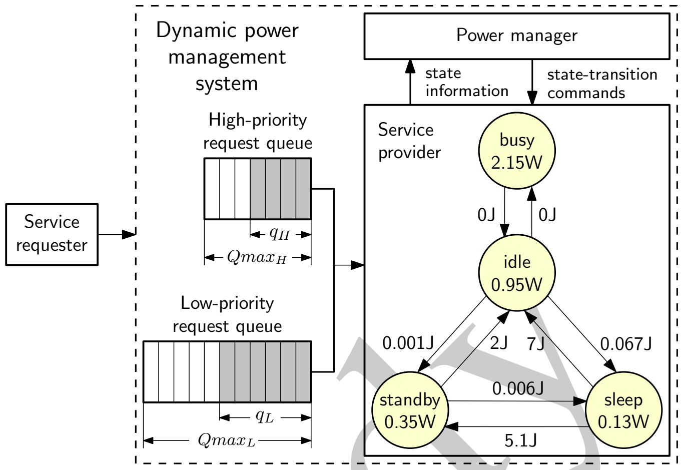


Fig. 1: Dynamic power management system

TABLE II: Average service-provider transition times

State transition	Average time (s)	State transition	Average time (s)
idle → *standby*	0.4	*standby* → *idle*	1.2
idle → *sleep*	0.67	*sleep* → *idle*	1.6
standby → *sleep*	0.3	*sleep* → *standby*	0.6

service rate (i.e., 125s^{-1}), and we assume average arrival rates of 0.05s^{-1} and 0.15s^{-1} for the high-priority and low-priority requests, respectively.

Finally, suppose that the DPM system designer must select:
1. the capacity of the request queues, $Qmax_H$ and $Qmax_L$;
2. one of two alternative power managers (described later);
3. the parameters of the selected power manager

such that the QoS requirements in Table III are satisfied.

III. EvoChecker

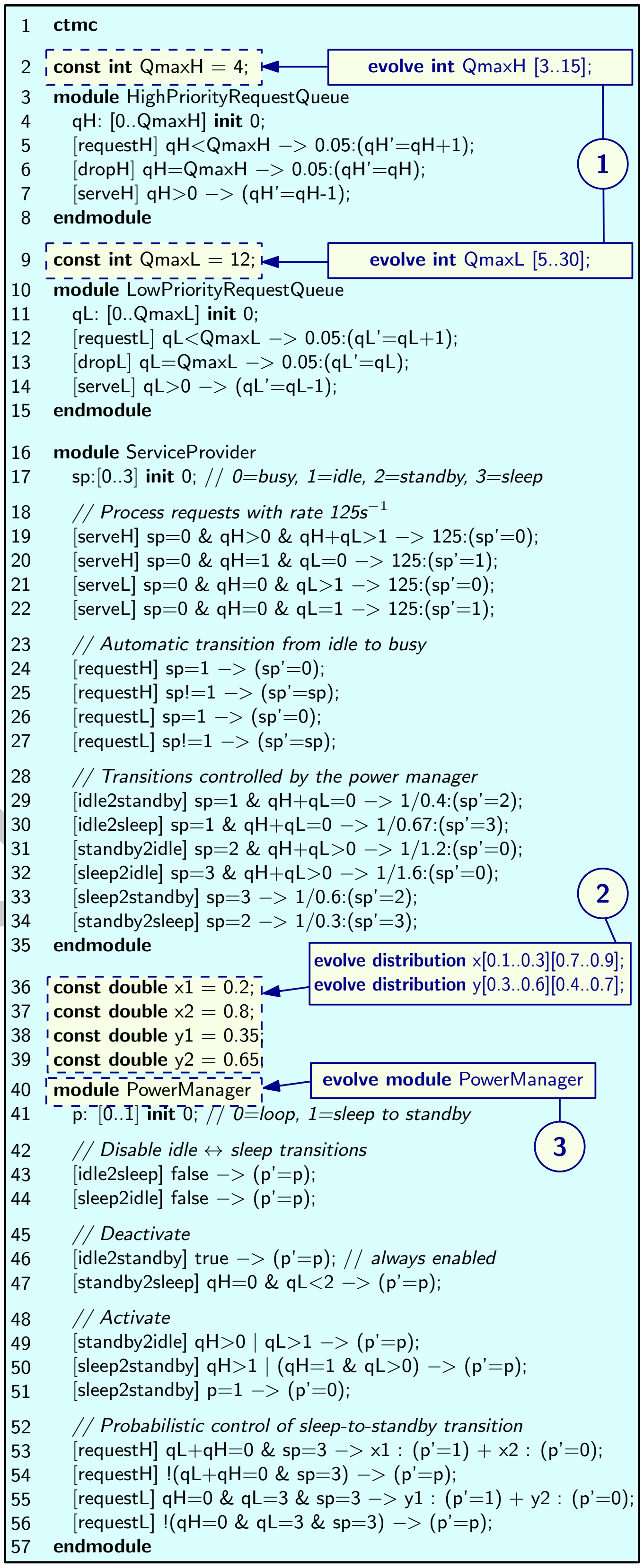
A. Modelling Language

EvoChecker probabilistic model templates are specified in an extended version of the PRISM high-level modelling language [47]. This language is based on the Reactive Modules formalism [3], which describes a system as the parallel composition of a set of *modules*. The state of a *module* is encoded by a set of finite-range local variables, and its state transitions are defined by probabilistic guarded commands that alter these variables, and have the general form:

$$[action]\ guard\ {-}{>}\ e_1 : update_1 + \ldots + e_n : update_n; \quad (1)$$

TABLE III: QoS requirements for the DPM system

ID	Description	Type
R1	The steady-state utilisation of the high-priority queue should be less than 90%	constraint
R2	The steady-state utilisation of the low-priority queue should be less than 90%	constraint
R3	The system should operate with minimum steady-state power utilisation	objective
R4	The number of requests lost at the steady state should be minimised	objective
R5	The capacity of both queues should be minimised	objective


evolve int QmaxH [3..15];
1
evolve int QmaxL [5..30];
2
evolve distribution x[0.1..0.3][0.7..0.9];
evolve distribution y[0.3..0.6][0.4..0.7];
evolve module PowerManager
3

```
1  ctmc
2  const int QmaxH = 4;
3  module HighPriorityRequestQueue
4    qH: [0..QmaxH] init 0;
5    [requestH] qH<QmaxH -> 0.05:(qH'=qH+1);
6    [dropH] qH=QmaxH -> 0.05:(qH'=qH);
7    [serveH] qH>0 -> (qH'=qH-1);
8  endmodule
9  const int QmaxL = 12;
10 module LowPriorityRequestQueue
11   qL: [0..QmaxL] init 0;
12   [requestL] qL<QmaxL -> 0.05:(qL'=qL+1);
13   [dropL] qL=QmaxL -> 0.05:(qL'=qL);
14   [serveL] qL>0 -> (qL'=qL-1);
15 endmodule

16 module ServiceProvider
17   sp:[0..3] init 0; // 0=busy, 1=idle, 2=standby, 3=sleep
18   // Process requests with rate 125s^-1
19   [serveH] sp=0 & qH>0 & qH+qL>1 -> 125:(sp'=0);
20   [serveH] sp=0 & qH=1 & qL=0 -> 125:(sp'=1);
21   [serveL] sp=0 & qH=0 & qL>1 -> 125:(sp'=0);
22   [serveL] sp=0 & qH=0 & qL=1 -> 125:(sp'=1);
23   // Automatic transition from idle to busy
24   [requestH] sp=1 -> (sp'=0);
25   [requestH] sp!=1 -> (sp'=sp);
26   [requestL] sp=1 -> (sp'=0);
27   [requestL] sp!=1 -> (sp'=sp);
28   // Transitions controlled by the power manager
29   [idle2standby] sp=1 & qH+qL=0 -> 1/0.4:(sp'=2);
30   [idle2sleep] sp=1 & qH+qL=0 -> 1/0.67:(sp'=3);
31   [standby2idle] sp=2 & qH+qL>0 -> 1/1.2:(sp'=0);
32   [sleep2idle] sp=3 & qH+qL>0 -> 1/1.6:(sp'=0);
33   [sleep2standby] sp=3 -> 1/0.6:(sp'=2);
34   [standby2sleep] sp=2 -> 1/0.3:(sp'=3);
35 endmodule
36 const double x1 = 0.2;
37 const double x2 = 0.8;
38 const double y1 = 0.35;
39 const double y2 = 0.65;
40 module PowerManager
41   p: [0..1] init 0; // 0=loop, 1=sleep to standby
42   // Disable idle ↔ sleep transitions
43   [idle2sleep] false -> (p'=p);
44   [sleep2idle] false -> (p'=p);
45   // Deactivate
46   [idle2standby] true -> (p'=p); // always enabled
47   [standby2sleep] qH=0 & qL<2 -> (p'=p);
48   // Activate
49   [standby2idle] qH>0 | qL>1 -> (p'=p);
50   [sleep2standby] qH>1 | (qH=1 & qL>0) -> (p'=p);
51   [sleep2standby] p=1 -> (p'=0);
52   // Probabilistic control of sleep-to-standby transition
53   [requestH] qL+qH=0 & sp=3 -> x1 : (p'=1) + x2 : (p'=0);
54   [requestH] !(qL+qH=0 & sp=3) -> (p'=p);
55   [requestL] qH=0 & qL=3 & sp=3 -> y1 : (p'=1) + y2 : (p'=0);
56   [requestL] !(qH=0 & qL=3 & sp=3) -> (p'=p);
57 endmodule
```

Fig. 2: CTMC model of the DPM system; ①–③ represent EvoChecker extensions of the PRISM modelling language

In this command, $guard$ is a boolean expression over all the variables in the model. If $guard$ evaluates to $true$, 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 occurs. The $action$ is optional; when present, it forces all modules comprising commands with this $action$ to perform one of these commands simultaneously (i.e., to synchronise). For a detailed description of the PRISM modelling language, we refer the reader to the PRISM manual, available at http://www.prismmodelchecker.org/manual.

Example 1: Fig. 2 shows the continuous-time Markov chain (CTMC) model of the dynamic power management system from our running example. The model comprises a module for each request queue, a ServiceProvider module, and a module for one of the two power managers considered during the design of the system. The local variables from the high-priority and low-priority request queue modules record the number of requests from the two queues, q_H and q_L, and the ServiceProvider local variable *sp* encodes the state of the service provider. The queue modules and ServiceProvider synchronise through the actions requestH,L and serveH,L, which reflect requests arriving into the queues and being served, respectively. The service provider switches between its *busy* and *idle* states automatically (lines 20, 22, 24 and 26), and can also perform the state transitions from Table II, aiming to switch to lower-power states when both request queues are empty (i.e., $q_H + q_L = 0$) and back towards the *busy* state otherwise (lines 28–34). However, these transitions are controlled by the power manager through the synchronisation of all ServiceProvider and PowerManager commands with common actions. For example, lines 43–44 have false guards and thus disable the transitions between *idle* and *sleep*, the transition between *idle* and *standby* is always enabled due to the true guard in line 46, and constraints are placed on all other transitions.

Software designers use probabilistic models to manually explore an often very large design space comprising alternative module implementations, parameter values and transition probabilities. EvoChecker extends the PRISM modelling language with three constructs that support the specification of all these design alternatives within a *probabilistic model template* whose instantiations correspond to possible system designs. The three constructs are defined below.

1. *Evolvable parameters.* EvoChecker uses the syntax

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

to declare 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 $action$, just like constant model parameters declared using '**const int**' and '**const double**' from the original language.

2. *Evolvable probability distributions.* The syntax

$$\text{evolve distribution } \mathit{dist}\ [\mathit{min}_1..\mathit{max}_1]\ldots[\mathit{min}_n..\mathit{max}_n]; \quad (3)$$

where $[\mathit{min}_i, \mathit{max}_i] \subseteq [0,1]$ for all $1 \leq i \leq n$, is used to declare an n-element discrete probability distribution, and ranges for the n probabilities of the distribution. The name of the distribution 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

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

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

The role of the three EvoChecker constructs is described by the following definitions.

Definition 1: A valid PRISM probabilistic model augmented with a set of EvoChecker constructs (2)–(4) is called a *probabilistic model template*.

Definition 2: A probabilistic model is an *instance* of a probabilistic model template $\mathcal{T}$ if and only if it can 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 $e_1, \ldots, e_n$ of a command (1) are replaced with values from the ranges $[min_1..max_1]$, ..., $[min_n..max_n]$, respectively, such that 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.

Definition 3: The set of all probabilistic models that are instances of a probabilistic model template $\mathcal{T}$ is called the *design space* of $\mathcal{T}$, and is denoted $DS^{\mathcal{T}}$.

Example 2: Fig. 2 illustrates the three EvoChecker constructs used to transform the CTMC model from our running example into a probabilistic model template. The replacement of the elements from the dashed rectangles with those from the continuous rectangles shows how: ① two evolvable parameters are used to specify the sizes of the two request queues; ② two evolvable distributions are used to specify the transition probabilities from lines 53 and 55 of the power manager; and ③ the module PowerManager is declared one of several possible implementations of the power manager. Note that at least one additional implementation of this module needs to be provided in a valid probabilistic model template; due to space constraints, the second implementation is not included here, but we make it available at http://www-users.cs.york.ac.uk/~simos/EvoChecker.

B. Quality-of-Service Requirements

EvoChecker supports two types of QoS requirements: *constraints* and *optimisation objectives*. Both types refer to QoS attributes of the system under design, such as response time, throughput, reliability or cost. Constraints define bounds for the acceptable values of these attributes, while optimisation objectives specify QoS attributes that should be minimised or maximised (subject to all constraints being satisfied). Without loss of generality, we will assume that the latter QoS attributes should be minimised in the remainder of the paper. Formally, a software system considered by EvoChecker needs to satisfy $n_1 \geq 0$ constraints of the form

$$R_i^C: \ attr_i \bowtie_i bound_i, \ 1 \leq i \leq n_1, \quad (5)$$

and $n_2 \geq 1$ optimisation objectives of the form

$$R_i^O: \ \textsf{minimise } attr_i, \ n_1+1 \leq i \leq n_1+n_2, \quad (6)$$

where $\bowtie_i \in \{<, \leq, =, \geq, >\}$ and $bound_i \in \mathbb{R}$ for all $1 \leq i \leq n_1$, and $attr_1, attr_2, \ldots, attr_{n_1+n_2} \in \mathbb{R}$ represent $n_1 + n_2$ QoS attributes of the system. These QoS attributes are formally expressed in the probabilistic temporal logics shown earlier in Table I. Given a probabilistic model $\mathcal{M}$ of a possible system design, and the probabilistic temporal logic formulae Φ_1, Φ_2, ..., $\Phi_{n_1+n_2}$ of the QoS attributes, the probabilistic model checker PRISM [47], [48] automatically establishes the value of the QoS attributes corresponding to this design, i.e.,

$$attr_i(\mathcal{M}) = PMC(\mathcal{M}, \Phi_i), \ 1 \leq i \leq n_1 + n_2. \quad (7)$$

Due to space constraints, we do not present the probabilistic temporal logics supported by EvoChecker; references to detailed descriptions of each logic are provided in Table I.

Example 3: The QoS requirements from our running example (Table III) comprise two constraints (**R1** and **R2**), and three optimisation objectives (**R3–R5**). The QoS attributes for all requirements are specified using *rewards* probabilistic temporal logic formulae [4], [42], [46]. To this end, positive values are associated with specific states and transitions of the CTMC in Fig. 2 by using **rewards...endrewards** structures. For example, the structure below enables the computation of the total number of lost requests for requirement R4:

```
rewards "TotalLost"
    [dropH] true : 1;
    [dropL] true : 1;
endrewards
```

by associating a value of 1 with each transition that corresponds to a request being dropped because of a full queue. The corresponding temporal logic formula is $\Phi_4 = \mathsf{R}_{=?}^{TotalLost}[\mathsf{S}]$, and represents the "Steady-state reward" for the above rewards structure, i.e., the long-run average number of dropped requests.

C. Probabilistic Model Synthesis Problem

Consider a probabilistic model template $\mathcal{T}$ with design space $DS^{\mathcal{T}}$, and a set of QoS requirements comprising n_1 constraints (5) and n_2 optimisation objectives (6). The *probabilistic model synthesis problem* involves finding the *Pareto-optimal design set PS* of models from $DS^{\mathcal{T}}$ that satisfy the n_1 constraints and are *non-dominated* with respect to the n_2 optimisation objectives:

$$PS = \big\{\mathcal{M} \in DS^{\mathcal{T}} \mid (\forall 1 \leq i \leq n_1 \bullet attr_i(\mathcal{M}) \bowtie_i bound_i) \wedge \big(\nexists \mathcal{M}' \in DS^{\mathcal{T}} \bullet \mathcal{M}' \prec \mathcal{M}\big)\big\}, \quad (8)$$

with the *dominance relation* $\prec : DS^{\mathcal{T}} \times DS^{\mathcal{T}} \to \mathbb{B}$ defined by

$$\begin{array}{l}\forall \mathcal{M}_1, \mathcal{M}_2 \in DS^{\mathcal{T}} \bullet \mathcal{M}_1 \prec \mathcal{M}_2 \equiv \\ \quad \forall n_1+1 \leq i \leq n_1+n_2 \bullet attr_i(\mathcal{M}_1) \leq attr_i(\mathcal{M}_2) \wedge \\ \quad \exists n_1+1 \leq i \leq n_1+n_2 \bullet attr_i(\mathcal{M}_1) < attr_i(\mathcal{M}_2).\end{array}$$

Finally, given the Pareto-optimal design set PS, we are also interested in the *Pareto front* defined by

$$\begin{array}{l}PF = \{(a_{n_1+1}, a_{n_1+2}, \ldots, a_{n_1+n_2}) \in \mathbb{R}^{n_2} \mid \\ \quad \exists \mathcal{M} \in PS \bullet \forall n_1+1 \leq i \leq n_1+n_2 \bullet a_i = attr_i(\mathcal{M})\},\end{array} \quad (9)$$

since designers need this information alongside their domain knowledge when choosing between the models in PS.

TABLE IV: EvoChecker gene encoding rules

Evolvable feature	**EvoChecker gene(s)**		
of the probabilistic model template	**Type**	**Cardinality**	**Value range, V_i**
evolve int $param[min..max]$;	int	1	$\{min,...,max\}$
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 $\ldots$ evolve module mod $implementation_m$ endmodule	int	1	$\{1, 2, ..., m\}$

D. EvoChecker Model Synthesis

Computing the Pareto-optimal design set (8) is in most cases unfeasible, as the design space $DS^{\mathcal{T}}$ is typically extremely large and may be infinite (i.e., when the probabilistic model template $\mathcal{T}$ contains evolvable parameters of type double and/or evolvable distributions). Therefore, EvoChecker uses *elitist genetic algorithms for multiobjective optimisation* (e.g., [27], [51], [66]) to synthesise a set of probabilistic models that closely approximates this Pareto-optimal set.

A *genetic algorithm* (GA) [45] encodes possible solutions of a search problem as a sequence of *genes* (i.e., binary representations of the problem variables). For EvoChecker, each instance of an 'evolvable' construct from Section III-A contributes to this sequence with the gene(s) given by the encoding rules in Table IV. These rules are used by the Template parser from the high-level EvoChecker architecture in Fig. 3 to extract the value ranges $V_1, V_2, \ldots, V_n$ for the $n \geq 1$ genes associated with the model template $\mathcal{T}$.

Given the solution encoding described above, GAs start with a randomly generated set (*population*) of feasible solutions (*individuals*). This population is then iteratively evolved into populations containing "fitter" individuals by means of GA *selection*, *crossover* and *mutation*. Selection uses a real-valued *fitness function* to evaluate each individual created during the GA execution, in order to select the population for the next iteration and the *mating pool* of individuals for the current iteration. Crossover randomly selects two individuals from the mating pool, and generates a new individual by combining their genes. Finally, mutation produces a new individual by randomly modifying some of the genes of an individual from the pool. GAs terminate after a fixed number of (individual) evaluations or when a predetermined number of successive iterations generate populations with no significantly fitter individuals.

As shown in Fig. 3, the GA used by EvoChecker is *elitist*, i.e., it preserves the best individuals across iterations. This involves maintaining a finite "archive" of the fittest individuals (e.g., by elitist GAs like MOCell [51] and SPEA2 [66]) or by retaining the fittest individuals from one iteration to the next (e.g., by NSGA-II [27]). The algorithms used by EvoChecker are also *multiobjective optimisation GAs* (MOGAs), i.e., they generate Pareto-optimal set approximations spread as uniformly as possible across the search space. Therefore, they use fitness functions that encode in ways specific to

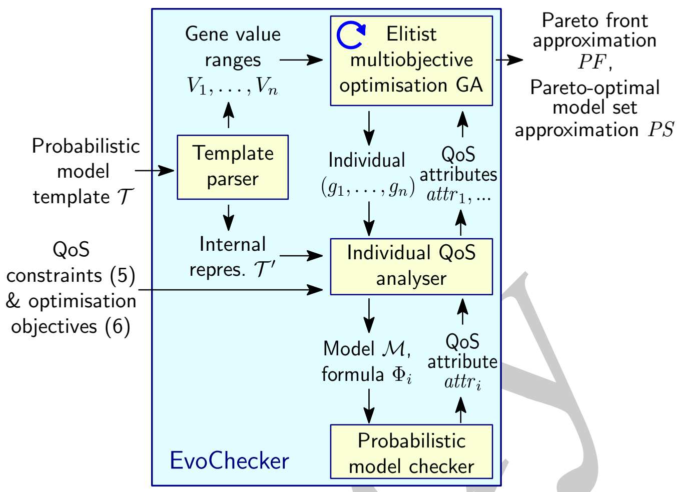

Gene value ranges $V_1, \ldots, V_n$
Elitist multiobjective optimisation GA
Pareto front approximation PF,
Pareto-optimal model set approximation PS
Probabilistic model template $\mathcal{T}$
Template parser
Individual $(g_1, \ldots, g_n)$
QoS attributes $attr_1, \ldots$
Internal repres. $\mathcal{T}'$
QoS constraints (5) & optimisation objectives (6)
Individual QoS analyser
Model $\mathcal{M}$, formula Φ_i
QoS attribute $attr_i$
EvoChecker
Probabilistic model checker

Fig. 3: High-level EvoChecker architecture

each MOGA both the *nondomination level* of each evaluated individual and the *population density* in its area of the search space. For example, NSGA-II [27] 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.[1]; and SPEA2 [66] evaluates population density as the inverse of the distance to the k-th nearest neighbour of the individual. Providing full technical details about the MOGAs that EvoChecker can use is beyond the scope of this paper. The reader can find their descriptions in [27], [51], [66], and a comparison of their merits in [29].

In EvoChecker the MOGA evaluation of the fitness of individuals is supported by an Individual QoS analyser (Fig. 3). This component takes as input the gene sequence $(g_1, \ldots, g_n)$ for an individual, and returns to the MOGA the QoS attributes $attr_1, attr_2, \ldots, attr_{n_1+n_2}$ of the model $\mathcal{M}$ associated with this individual. To this end, the analyser uses an internal representation $\mathcal{T}'$ of the model template produced by the Template parser and the gene sequence $(g_1, \ldots, g_n)$ to obtain the model $\mathcal{M}$, and then invokes a Probabilistic model checker to establish the $n_1 + n_2$ QoS attributes from (7) one at a time. This setup enables the MOGA to generate approximations of the Pareto-optimal model set PS and of the associated Pareto front PF as illustrated in Fig. 3.

Example 4: Consider again the DPM system from our running example. Using the rules in Table IV, EvoChecker encodes each instance $\mathcal{M}$ of the probabilistic model template from Fig. 2 as the sequence of genes

$$(Qmax_H, Qmax_L, x_1, x_2, y_1, y_2, pm), \tag{10}$$

where $Qmax_H \in V_1 = \{3, 4, \ldots, 15\}$ and $Qmax_L \in V_2 = \{5, 6, \ldots, 30\}$ represent the capacities of the two queues; $x_1 \in V_3 = [0.1, 0.3]$ and $x_2 \in V_4 = [0.7, 0.9]$ such that $x_1 + x_2 = 1$, and $y_1 \in V_5 = [0.3, 0.6]$ and $y_2 \in V_6 = [0.4, 0.7]$ such that $y_1 + y_2 = 1$ represent the two probability distributions; and $pm \in V_7 = \{1, 2\}$ is a gene that encodes which of the $m = 2$ implementations of the PowerManager module in Fig. 2 is used by $\mathcal{M}$. The gene value ranges V_1 to V_7 are extracted by the EvoChecker Template parser and passed to the MOGA,

[1]Individuals not satisfying problem constraints receive a default level of ∞.

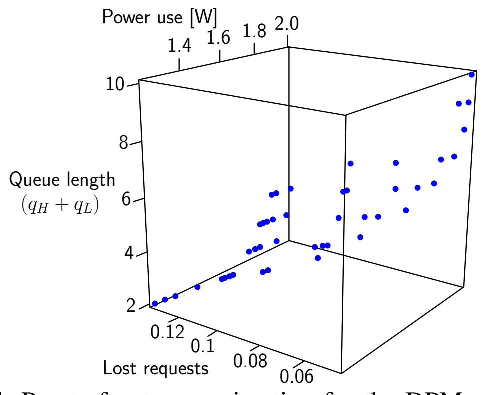


Fig. 4: Pareto front approximation for the DPM system

which uses the Individual QoS analyser to iteratively generate model populations of increasing fitness and eventually outputs the Pareto front and Pareto-optimal set approximations achieved after a pre-established number of evaluations. As an example, Fig. 4 depicts a Pareto front approximation obtained using NSGA-II [27] as the EvoChecker MOGA, with an initial population of 150 individuals and after 5000 evaluations. Remember that the dimensionality of Pareto front for a probabilistic model synthesis problem is given by the number of optimisation objectives n_2. Our DPM system has $n_2 = 3$ optimisation objectives, as shown in Table III.

IV. EvoChecker tool

To automate the EvoChecker synthesis of probabilistic models, we implemented a tool with the architecture in Fig. 3. Our EvoChecker tool uses the established MOGA implementations provided by the Java-based framework for multi-objective optimization with metaheuristics JMetal [28], and the probabilistic model checker PRISM [47]. We developed the other components of the architecture from Fig. 3 in Java, using the Antlr[2] parser generator to build the Template parser, and implementing an Individual QoS analyser specifically for the EvoChecker tool. 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.

V. Evaluation

A. Research Questions

We evaluated the effectiveness of EvoChecker by performing extensive experiments to answer the next research questions.

RQ1 (Validation): How does EvoChecker perform compared to random search? We used this research question to establish if EvoChecker "*comfortably outperforms a random search*" [40], as expected of effective search-based software engineering solutions.

RQ2 (Comparison): How do EvoChecker instances using different MOGAs perform compared to each other? Since we devised EvoChecker to work with any MOGA, we

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

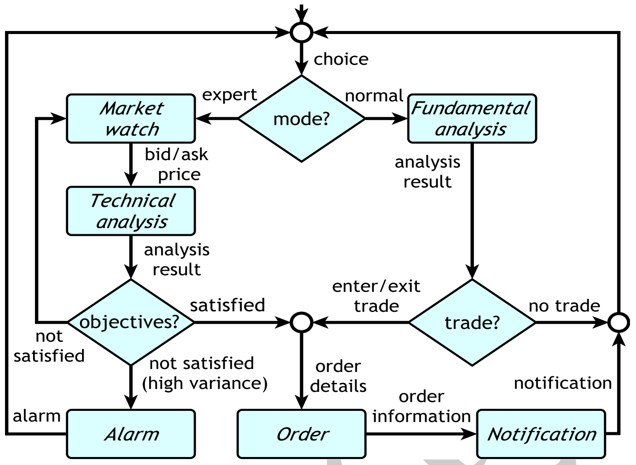


Fig. 5: The FX workflow

examined the results produced by EvoChecker instances using three established such algorithms (i.e., NSGA-II [27], SPEA2 [66] and MOCell [51]).

RQ3 (Insights): Can EvoChecker provide insights into the trade-offs between the QoS attributes of alternative software architectures and configurations? To support software designers in their decision making, EvoChecker must provide insights into the trade-offs between multiple QoS objectives. To address this question, we identified a range of design decisions suggested by the EvoChecker results for the software systems considered in our evaluation.

B. Analysed Software Systems

Our experiments used EvoChecker in multiple scenarios associated with two software systems from different application domains—the dynamic power management (DPM) system from our running example, and a real-world service-based system from the area of foreign exchange trading. The second system, which we anonymise as FX for confidentiality reasons, is used by an European foreign exchange brokerage company, and implements the workflow in Fig. 5.

An FX customer (called a trader) can use the system in two operation modes. In the *expert* mode, FX executes a loop that analyses market activity, identifies patterns that satisfy the trader's objectives, and automatically carries out trades. Thus, the *Market watch* service extracts real-time exchange rates (bid/ask price) of selected currency pairs. This data is used by a *Technical analysis* service that evaluates the current trading conditions, predicts future price movement, and decides if the trader's objectives are: (i) "satisfied" (causing the invocation of an *Order* service to carry out a trade); (ii) "unsatisfied" (resulting in a new *Market watch* invocation); or (iii) "unsatisfied with high variance" (triggering an *Alarm* service invocation to notify the trader about discrepancies/opportunities not covered 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 on the country's currency. If satisfied with this assessment, the trader can use the *Order* service to sell or buy currency, in which case a *Notification* service confirms the completion of the trade.

We assume that the FX designer has to select third-party implementations for each of the $n \geq 1$ services from Fig. 5 for which in-house implementations are not available, in order to meet the QoS requirements from Table V. The designer can use any subset of the $n_i \geq 1$ third-party implementations of the i-th service unavailable in-house, and either a *probabilistic* or a *sequential* selection strategy. The probabilistic strategy involves using a randomly selected third-party service from the subset for each invocation of service i, where the random selection is made according to a discrete probability distribution decided by the designer. The sequential strategy involves invoking the services from the subset in a designer-specified order, until one of the invocations is successful or all the invocations failed. We used an EvoChecker probabilistic model template to capture these alternative FX designs. This template uses all EvoChecker "evolvable" constructs, and is parameterised by the number of services n, the number of third-party service implementations n_1, n_2, …, and by the costs, success probabilities and response times of these implementations.

To evaluate EvoChecker for multiple design space sizes, we applied it to each of the system variants from Table VI. The entries in this table list the int-valued probabilistic model template parameters from (10)—for the DPM system, and described above—for the FX system. The 'Size' column reports the size of the design space that an exhaustive search would need to explore, assuming two-decimal precision for the double-valued parameters of the probabilistic model templates (cf. Table IV). This is a valid assumption, as the nonlinearity of most probabilistic models means that a 0.01 change in a state transition probability often translates into significant changes in the values of model properties. Finally, note that the $n = 8$ services used by FX_Large correspond to using two-part composite services for the *Technical analysis* and *Fundamental analysis* operations from Fig. 5.

C. Evaluation Methodology

In line with the standard practice for evaluating the performance of stochastic optimisation algorithms [6], we performed multiple (i.e., 30) independent runs for each system variant from Table VI and each multiobjective optimisation algorithm—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 system variant parameters. 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. The average run times for the six system variants are shown in Table VI. Note that the EvoChecker run time depends on the size of model $\mathcal{M}$ and the time consumed by the probabilistic model checker to establish the n1 + n2 QoS attributes from (7).

TABLE V: QoS requirements for the FX system

ID	Description	Type
R1	Workflow executions must complete successfully with probability at least 0.9	constraint
R2	The total service response time per workflow execution should be minimised	objective
R3	The probability of a service failure during a workflow execution should be minimised	objective
R4	The total cost of the third-party services used by a workflow execution should be minimised	objective

TABLE VI: Analysed system variants; $\mathrm{T_{run}}$ represents the EvoChecker running time averaged over 30 runs

Variant	Int-valued parameters	Size	$\mathrm{T_{run}}$
DPM_Small	$Qmax_{H,L} \in \{1, ..., 10\}$, $m = 2$	2E+14	1050s
DPM_Medium	$Qmax_{H,L} \in \{1, ..., 15\}$, $m = 2$	4.5E+14	2118s
DPM_Large	$Qmax_{H,L} \in \{1, ..., 20\}$, $m = 2$	8E+14	3796s
FX_Small	$n = 4$, $n_1 = \cdots = n_4 = 3$	4.98E+31	858s
FX_Medium	$n = 6$, $n_1 = \cdots = n_6 = 4$	1.39E+65	1695s
FX_Large	$n = 8$, $n_1 = \cdots = n_6 = 4$	7.22E+86	4162s

Obtaining the actual Pareto front for our system variants is unfeasible because of their very large design spaces. Therefore, we adopted the established practice [65] 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:

1. *Unary additive epsilon* (I_ϵ) [68], i.e., 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*[3]. Smaller I_ϵ values denote better Pareto front approximations.
2. *Hypervolume* (I_{HV}) [67], which measures the volume in the objective space covered by a Pareto front approximation with respect to the reference front (or a reference point). I_{HV} measures both convergence and diversity, and is strictly Pareto compliant [64]. Larger I_{HV} values denote better Pareto front approximations.
3. *Inverted Generational Distance* (I_{IGD}) [60], which provides 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 [6], [40], [65]. As is typical of multiobjective optimisation [65], the Shapiro-Wilk test showed that the quality indicators were not normally distributed, so we used the Kruskal-Wallis non-parametric 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 carried out a post-hoc analysis with pairwise comparisons between the four algorithms by means of 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.

[3]Pareto compliant indicators do not "contradict" the order introduced by the Pareto dominance relation on Pareto front approximations [64].

TABLE VII: Mean quality indicator values for the system variants from Table VI

Problem	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	+

Problem	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	+

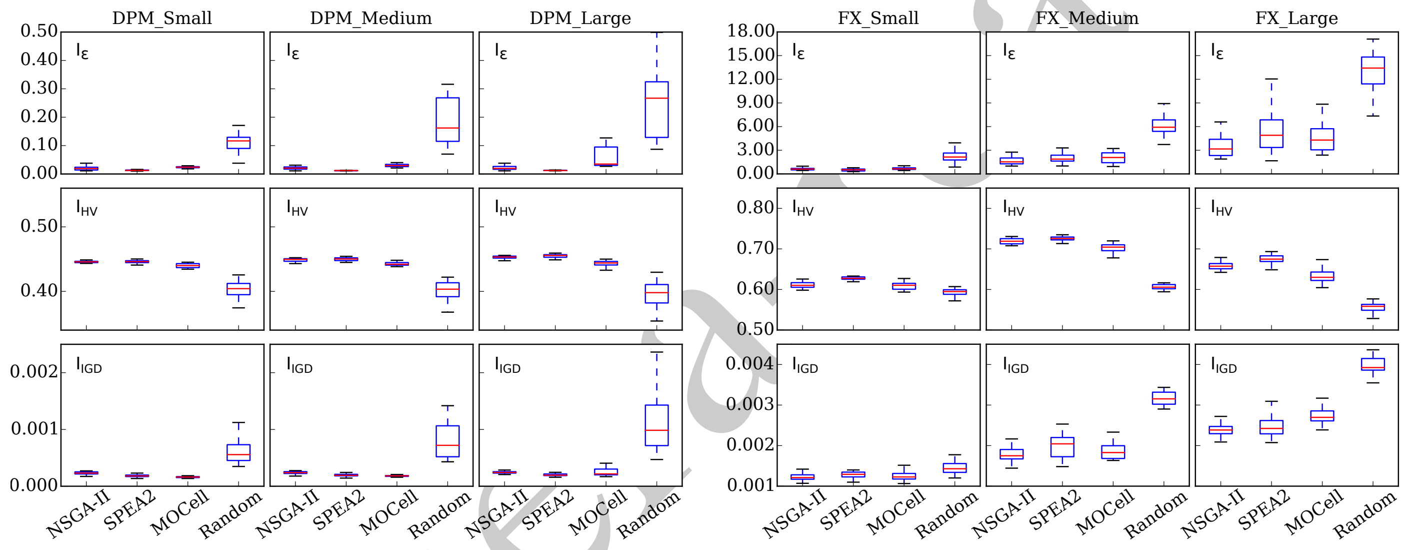


Fig. 6: Boxplots for the six system variants from Table VI, evaluated using the quality indicators I_ϵ, I_{HV} and I_{IGD}

D. Results and Discussion

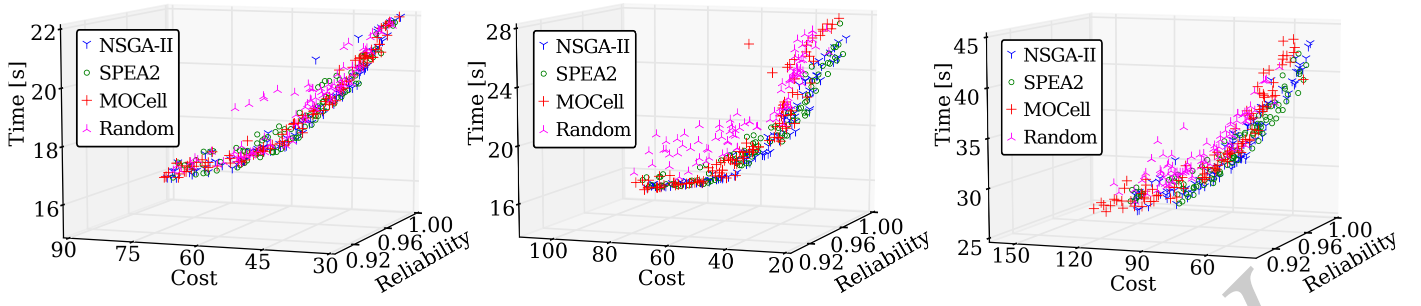
RQ1(Validation). We carried out the experiments described in the previous section and we report their results in Table VII and Fig. 6. The '+' from the last column of the table entries indicate that the Kruskal-Wallis test showed significant difference among the four algorithms (p-value$<$0.05) for all six system variants and all Pareto-front quality indicators.

For both systems, EvoChecker with 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. 6.

We qualitatively support our findings by showing in Fig. 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 FX system variants. 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 in Section II, the parameters we used for the DPM system variants (power usage, transition rates, etc.) correspond to the real-world system. 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 additional experiments comprising 300 independent runs per FX system variant (900 runs in total) in which these parameters were randomly instantiated. The results of this further analysis (reported on our project webpage but not included here due to space constraints) confirm the findings presented above.

All these results provide 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 *probabilistic model synthesis problem* we introduced in Section III-C.



(a) FX_Small (b) FX_Medium (c) FX_Large

Fig. 7: Typical Pareto front approximations for the FX system variants and optimisation objectives **R2-R4** from Table V

RQ2 (Comparison). To compare EvoChecker instances based on different MOGAs, we first observe in Table VII 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 design 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 design spaces of the FX system variants (except the combination FX_small–I_ϵ).

Additionally, we carried out the post-hoc analysis described in Section V-C, for 9 system variants (counting separately the FX system variants with chosen parameters and with randomly instantiated parameters) $\times$ 3 quality indicators = 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 VIII). The five system variant–quality indicator combinations for which the tests were unsuccessful are: FX_Medium–I_ϵ, FX(random)_Small–I_ϵ, FX(random)_Medium–I_ϵ, FX_Small–I_{IGD} and FX_Medium–I_{IGD}.

These results show that, like for any well-formulated optimisation problem, different algorithms are more suitable in dealing with specific problems. They also confirm the generality of EvoChecker, 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, with

TABLE VIII: System variants for which the MOGAs in rows are significantly better than the MOGAs in columns

		NSGA-II	**SPEA2**	**MOCell**
NSGA-II	I_ϵ:		FX_L,RL	DPM_M,L
	I_{HV}:		–	DPM_S,M,L; FX_M,L,RM,RL
	I_{IGD}:		–	FX_L,RL
SPEA2	I_ϵ:	DPM_S,M,L		DPM_S,M,L; FX_S
	I_{HV}:	FX_S,M,L, RS,RM,RL		DPM_S,M,L; FX_S,M,L,RS,RM,RL
	I_{IGD}:	DPM_S,M,L; FX_RS		DPM_L; FX_L,RS,RM,RL
MOCell	I_ϵ:	–	–	
	I_{HV}:	–	–	
	I_{IGD}:	DPM_S,M	DPM_S	

Key: S=Small, M=Medium, L=Large, R=random parameters

the purpose of identifying actionable insights. We illustrate this for the FX and DPM Pareto front approximations from Fig. 7 and Fig. 4, respectively. First, the EvoChecker results enable the identification of the "point of diminishing returns" for each system variant. The results from Fig. 7 show that design options 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 designs achievable for these costs. Likewise, the results in Fig. 4 show that DPM designs with power use above 1.7W yield insignificant reductions in the number of lost requests, whereas designs with even slightly lower power use lead to much higher request loss. This key information helps designers avoid unnecessarily expensive solutions.

Second, we note the high density of design solutions in the areas with low reliability (below 0.95) for the FX system in Fig. 7, and with high request loss (above 0.09) for the DPM system in Fig. 4. For the FX system, for instance, these correspond to the use of the probabilistic service selection strategy, for which numerous service combinations can achieve similar reliability and response time with relatively low, comparable costs. Opting for a design from this area will make the FX system susceptible to failures, as when the only service invoked for an FX operation fails, the entire workflow execution will also fail. In contrast, reliability values above 0.995 correspond to high-cost designs that use the sequential service 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 design parameters that QoS properties are particularly sensitive to. For the FX system, for example, we noticed a strong dependency of the workflow reliability on the service selection strategy and the number of services used for each operation. Designs from high-reliability areas of the Pareto front not only use the sequential selection strategy, but also require multiple services per FX operation (e.g., three FX service providers are needed for success rates above 0.99).

Finally, we note EvoChecker's ability to produce solutions that: (a) cover a wide range of values for the QoS attributes from the optimisation objectives of the FX and DPM systems; and (b) include alternatives with different trade-offs for fixed values of one of these attributes. Thus, for 0.99 reliability, the experiment from Fig. 7 generated four alternative FX_Large designs, 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 designers in their decision making.

E. Threats to Validity

Construct validity threats may be due to the simplifications and assumptions we made when modelling the DPM and FX systems. To mitigate this threat, the DPM system, model and requirements are based on a validated real-world case study taken from the literature [55], [58], which we were familiar with from our previous work [20]. For the FX system, the model and requirements were developed in close collaboration with a foreign exchange domain expert.

Internal validity threats can originate from the stochastic nature of the optimisation algorithms employed in our study. To mitigate these threats, we adopted the recommended practice for empirical studies in this research area [6], [40]. In particular, we reported results over 30 repeated runs of each experiment, and employed statistical tests to check for significance in the achieved results. Thus, we used the Shapiro-Wilk test to assess the normality of data, and the Kruskall-Wallis and Dunn pairwise non-parametric tests to check for statistical significance between the considered optimisation algorithms. Finally, we set the chance of committing a Type I error (α confidence limit) at 0.05, which is the standard value recommended for these studies.

External validity threats might be due to the difficulty of representing a software system and its QoS requirements as a probabilistic model synthesis problem (Section III-C) using EvoChecker constructs (2)–(4), constraints (5) and optimisation objectives (6). We limit this threat by specifying EvoChecker probabilistic model templates in an extended version of the high-level modelling language of PRISM [47], a widely-used probabilistic model checker [52]. Moreover, given the generality of the EvoChecker constructs (2)–(4), other probabilistic modelling languages (e.g., those of the model checkers MRMC [42], [43] and Ymer [62]) can be naturally supported. Additionally, EvoChecker supports a wide range of probabilistic models and temporal logics (Table I). Finally, to further reduce the risk that EvoChecker might be difficult to use in practice, we validated it through application to several variants of two realistic software systems with diverse characteristics in terms of application domain, size, complexity and requirements. Nevertheless, we are aware that our findings are by no means conclusive for all types of software systems, and more experiments are required to confirm the generality of the EvoChecker approach and tool.

VI. Related Work

Search-based software engineering (SBSE) [40] has been successfully used in areas ranging from project management [30], [56], [59] and testing [5], [33], [38] to effort estimation [50], software repair and evolution [21], [54] and software product lines [37], [57]. However, as reported in Harman *et al.*'s recent SBSE survey [39], this success does not yet extend to model checking. The only SBSE applications that we are aware of in this area are the approaches in [41], [44], which employ genetic evolution to synthesise model checking specifications, and the work in [1], [2], which uses ant colony optimisation to find counterexamples for large models.

To the best of our knowledge, EvoChecker is the first SBSE approach to synthesising Pareto-optimal sets of probabilistic models for QoS software engineering. As mentioned in Section I, the only related work we are aware of belongs to the area of *model repair* [16], [63]. Unlike EvoChecker, most model repair research so far has focused on non-probabilistic models [15], [16], [22], [23], [49]. Also different from EvoChecker, these approaches support a single type of model, and target its "repair" with respect to a single temporal logic property.

In the probabilistic model checking domain, model repair involves automatically modifying the transition probabilities of a model that violates a formally-specified property in order to obtain a new model that satisfies the property and is "close" to the original model [10], [11], [24]. As probabilistic model repair modifies the original model only at transition probability level, and only to satisfy a single temporal logic property, its applicability is limited. In contrast, EvoChecker handles multiple QoS requirements and uses multiobjective optimisation to derive a set of probabilistic models that approximates the Pareto-optimal model set associated with these QoS requirements. We confirmed that EvoChecker subsumes the capabilities of probabilistic model repair by replicating the results of the IPv4 Zeroconf Protocol [10] and Network Virus Infection [24] case studies. Due to space constraints, we report the results of these extra experiments on our project website.

Finally, [32] synthesises Pareto front approximations over the policies of Markov decision processes (MDPs). However, its applicability is limited to fully specified MDPs, to a subset of probabilistic computation tree logic (i.e., reachability and expected total reward formulae), and to the finite search spaces that can be encoded as MDP policies. In contrast, EvoChecker fully supports all types of models and logics from Table I, and solves a more general problem by starting from incompletely specified probabilistic model templates with potentially infinite search spaces (due to evolvable `double` parameters and distributions). Moreover, the implementation in [32] currently supports only up to 3 optimisation objectives, while EvoChecker does not have this limitation.

VII. Conclusion

We defined the probabilistic model synthesis problem, and introduced EvoChecker, the first tool-supported search-based approach that tackles this problem with a focus on QoS software engineering. EvoChecker uses multiobjective optimisation genetic algorithms to automate the synthesis of approximate Pareto-optimal probabilistic model sets associated with the QoS requirements of a software system. We evaluated the EvoChecker approach and tool within two case studies from different domains, showing its effectiveness, applicability and flexibility.

The future research directions for EvoChecker include extending its applicability to other modelling formalisms and verification logics by exploiting established quantitative model checkers such as UPPAAL [12], exploring alternative multi-objective optimisation [34], and integrating the EvoChecker approach with our related work on runtime probabilistic model checking [17], [18], [19], [31], [35]. The further exploration of the EvoChecker usability and performance, and its applicability to other domains represent areas of future work.

Acknowledgements

This paper presents research 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

[1] E. Alba and F. Chicano. Finding safety errors with ACO. In *9th Intl. Conf. on Genetic and Evolutionary Computation (GECCO'07)*, pages 1066–1073, 2007.

[2] E. Alba and F. Chicano. Searching for liveness property violations in concurrent systems with ACO. In *10th Intl. Conf. on Genetic and Evolutionary Computation (GECCO'08)*, pages 1727–1734, 2008.

[3] R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[4] S. Andova, H. Hermanns, and J.-P. Katoen. Discrete-time rewards model-checked. In *Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, volume 2791 of *LNCS*, pages 88–104. 2004.

[5] J. Andrews, T. Menzies, and F. Li. Genetic algorithms for randomized unit testing. *IEEE Trans. on Software Engineering*, 37(1):80–94, 2011.

[6] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *33rd Intl. Conf. on Software Engineering (ICSE'11)*, pages 1–10, 2011.

[7] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In *Computer Aided Verification (CAV'96)*, pages 269–276, 1996.

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

[9] C. Baier, J. P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In *10th Intl. Conf. on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 146–161, 1999.

[10] E. Bartocci, R. Grosu, P. Katsaros, C. Ramakrishnan, and S. A. Smolka. Model repair for probabilistic systems. In *17th Intl. Conf. on Tools and Algorithms Construction and Analysis of Systems (TACAS'11)*, pages 326–340. 2011.

[11] M. Benedikt, R. Lenhardt, and J. Worrell. LTL model checking of interval Markov chains. In *17th Intl. Conf. on Tools and Algorithms Construction and Analysis of Systems (TACAS'13)*, pages 32–46. 2013.

[12] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. *UPPAAL - a tool suite for automatic verification of real-time systems*. Springer, 1996.

[13] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *15th Intl. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'95)*, volume 1026 of *LNCS*, pages 499–513, 1995.

[14] B. Boehm and V. R. Basili. Software Defect Reduction Top 10 List. *Computer*, 34(1):135–137, Jan. 2001.

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

[16] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. Enhancing model checking in verification by AI techniques. *Artificial Intelligence*, 112:57–104, 1999.

[17] R. Calinescu, S. Gerasimou, and A. Banks. Self-adaptive software with decentralised control loops. In *18th Intl. Conf. on Fundamental Approaches to Software Engineering (FASE'15)*, volume 9033 of *LNCS*, pages 235–251. 2015.

[18] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM*, 55(9):69–77, September 2012.

[19] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic QoS management and optimization in service-based systems. *IEEE Trans. on Software Engineering*, 37:387–409, 2011.

[20] R. Calinescu and M. Z. Kwiatkowska. Using quantitative analysis to implement autonomic IT systems. In *31st Intl. Conf. on Software Engineering (ICSE'09)*, pages 100–110, 2009.

[21] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. An approach for QoS-aware service composition based on genetic algorithms. In *7th Intl. Conf. on Genetic and Evolutionary Computation (GECCO'05)*, pages 1069–1075, 2005.

[22] M. Carrillo and D. A. Rosenblueth. CTL update of Kripke models through protections. *Artificial Intelligence*, 211(0):51 – 74, 2014.

[23] G. Chatzieleftheriou, B. Bonakdarpour, S. A. Smolka, and P. Katsaros. Abstract model repair. In *NASA Formal Methods*, pages 341–355. 2012.

[24] T. Chen, E. M. Hahn, T. Han, M. Kwiatkowska, H. Qu, and L. Zhang. Model repair for Markov decision processes. In *7th Intl. Symp. on Theoretical Aspects of Software Engineering. (TASE'13)*, pages 85–92, 2013.

[25] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[26] L. O. Damm and L. Lundberg. Company-wide implementation of metrics for early software fault detection. In *29th Intl. Conf. on Software Engineering (ICSE'07)*, pages 560–570, 2007.

[27] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. on Evolutionary Computation*, 6(2):182–197, 2002.

[28] J. J. Durillo and A. J. Nebro. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42:760–771, 2011.

[29] J. J. Durillo, A. J. Nebro, C. A. Coello Coello, J. García-Nieto, F. Luna, and E. Alba. A study of multiobjective metaheuristics when solving parameter scalable problems. *IEEE Trans. on Evolutionary Computation*, 14(4):618–635, 2010.

[30] F. Ferrucci, M. Harman, J. Ren, and F. Sarro. Not going to take this anymore: Multi-objective overtime planning for software engineering projects. In *35th Intl. Conf. on Software Engineering (ICSE'13)*, pages 462–471, 2013.

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

[32] V. Forejt, M. Kwiatkowska, and D. Parker. Pareto curves for probabilistic model checking. In *10th Intl. Symp. on Automated Technology for Verification & Analysis*, volume 7561 of *LNCS*, pages 317–332, 2012.

[33] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Trans. on Software Engineering*, 39(2):276–291, 2013.

[34] X. Gandibleux. *Metaheuristics for multiobjective optimisation*, volume 535. Springer Science & Business Media, 2004.

[35] S. Gerasimou, R. Calinescu, and A. Banks. Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration. In *9th Intl. Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*, pages 115–124, 2014.

[36] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[37] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang. Search based software engineering for software product line engineering: A survey and directions for future work. In *18th Intl. Software Product Line Conf.*, pages 5–18, 2014.

[38] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *19th ACM SIGSOFT Symposium and the 13th European Conf. on Foundations of Software Engineering (ESEC/FSE'11)*, pages 212–222, 2011.

[39] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11:1–11:61, 2012.

[40] M. Harman, P. McMinn, J. de Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical Software Engineering and Verification*, volume 7007 of *LNCS*, pages 1–59. 2012.

[41] C. Johnson. Genetic programming with fitness based on model checking. In *Genetic Programming*, volume 4445 of *LNCS*, pages 114–124. 2007.

[42] J. P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *2nd Intl. Conf. on Quantitative Evaluation of Systems (QEST'05)*, pages 243–244, 2005.

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

[44] G. Katz and D. Peled. Synthesis of parametric programs using genetic programming and model checking. In *15th Intl. Workshop on Verification of Infinite-State Systems (INFINITY'13)*, pages 70–84, 2013.

[45] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[46] M. Kwiatkowska. Quantitative verification: Models, techniques and tools. In *6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*, pages 449–458, 2007.

[47] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *23rd Intl. Conf. on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591, 2011.

[48] M. Z. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. *Int. Journal on Software Tools for Technology Transfer*, 6(2):128–142, 2004.

[49] U. Martinez-Araiza and E. Lopez-Mellado. A CTL model repair method for Petri Nets. In *World Automation Congress (WAC'14)*, pages 654–659, 2014.

[50] L. L. Minku and X. Yao. Software effort estimation as a multiobjective learning problem. *ACM Trans. on Software Engineering and Methodology*, 22(4):35:1–35:32, 2013.

[51] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba. MOCell: A cellular genetic algorithm for multiobjective optimization. *Intl. Journal of Intelligent Systems*, 24(7):726–746, 2009.

[52] G. Norman and D. Parker. Quantitative verification: Formal guarantees for timeliness, reliability and performance. Technical report, The London Mathematical Society and the Smith Institute, 2014.

[53] A. Pnueli. The temporal logic of programs. In *18th Annual Symp. on Foundations of Computer Science*, pages 46–57, 1977.

[54] K. Praditwong, M. Harman, and X. Yao. Software module clustering as a multi-objective search problem. *IEEE Trans. on Software Engineering*, 37(2):264–282, March 2011.

[55] Q. Qiu, Q. Qu, and M. Pedram. Stochastic modeling of a power-managed system-construction and optimization. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(10):1200–1217, 2001.

[56] J. Ren, M. Harman, and M. Di Penta. Cooperative co-evolutionary optimization of software project staff assignments and job scheduling. In *3rd Intl. Symp. on Search Based Software Engineering (SSBSE'11)*, volume 6956 of *LNCS*, pages 127–141. 2011.

[57] A. Sayyad, J. Ingram, T. Menzies, and H. Ammar. Scalable product line configuration: A straw to break the camel's back. In *28th Intl. Conf. on Automated Software Engineering (ASE'13)*, pages 465–474, 2013.

[58] A. Sesic, S. Dautovic, and V. Malbasa. Dynamic power management of a system with a two-priority request queue using probabilistic-model checking. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 27(2):403–407, 2008.

[59] C. Stylianou, S. Gerasimou, and A. Andreou. A novel prototype tool for intelligent software project scheduling and staffing enhanced with personality factors. In *24th Intl. Conf. on Tools with Artificial Intelligence (ICTAI'12)*, pages 277–284, 2012.

[60] D. A. Van Veldhuizen. *Multiobjective Evolutionary Algorithms: Classifications, Analyses, and New Innovations*. PhD thesis, 1999.

[61] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):19, 2009.

[62] H. L. S. Younes. Ymer: A statistical model checker. In *17th Intl. Conf. on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 429–433. 2005.

[63] Y. Zhang and Y. Ding. CTL model update for system modifications. *Journal of Artificial Intelligence Research (JAIR)*, 31:113–155, 2008.

[64] E. Zitzler, D. Brockhoff, and L. Thiele. The hypervolume indicator revisited: On the design of pareto-compliant indicators via weighted integration. In *4th Intl. Conf. on Evolutionary Multi-criterion Optimization (EMO'07)*, pages 862–876, 2007.

[65] E. Zitzler, J. Knowles, and L. Thiele. Quality assessment of Pareto set approximations. In *Multiobjective Optimization*, volume 5252 of *LNCS*, pages 373–404. 2008.

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

[67] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Trans. on Evolutionary Computation*, 3(4):257–271, 1999.

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