Information and Software Technology 51 (2009) 957–976

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A systematic review of search-based testing for non-functional system properties

Wasif Afzal *, Richard Torkar, Robert Feldt

Department of Systems and Software Engineering, School of Engineering, Blekinge Institute of Technology, S-372 25 Ronneby, Sweden

ARTICLE INFO

Article history:
Received 7 May 2008
Received in revised form 16 December 2008
Accepted 18 December 2008
Available online 4 February 2009

Keywords:
Systematic review
Non-functional system properties
Search-based software testing

ABSTRACT

Search-based software testing is the application of metaheuristic search techniques to generate software tests. The test adequacy criterion is transformed into a fitness function and a set of solutions in the search space are evaluated with respect to the fitness function using a metaheuristic search technique. The application of metaheuristic search techniques for testing is promising due to the fact that exhaustive testing is infeasible considering the size and complexity of software under test. Search-based software testing has been applied across the spectrum of test case design methods; this includes white-box (structural), black-box (functional) and grey-box (combination of structural and functional) testing. In addition, metaheuristic search techniques have also been applied to test non-functional properties. The overall objective of undertaking this systematic review is to examine existing work into non-functional search-based software testing (NFSBST). We are interested in types of non-functional testing targeted using metaheuristic search techniques, different fitness functions used in different types of search-based non-functional testing and challenges in the application of these techniques. The systematic review is based on a comprehensive set of 35 articles obtained after a multi-stage selection process and have been published in the time span 1996–2007. The results of the review show that metaheuristic search techniques have been applied for non-functional testing of execution time, quality of service, security, usability and safety. A variety of metaheuristic search techniques are found to be applicable for non-functional testing including simulated annealing, tabu search, genetic algorithms, ant colony methods, grammatical evolution, genetic programming (and its variants including linear genetic programming) and swarm intelligence methods. The review reports on different fitness functions used to guide the search for each of the categories of execution time, safety, usability, quality of service and security; along with a discussion of possible challenges in the application of metaheuristic search techniques.

Contents

* Corresponding author. Tel.: +46 457 385840; fax: +46 457 27914.
E-mail addresses: wasif.afzal@bth.se (W. Afzal), richard.torkar@bth.se (R. Torkar), robert.feldt@bth.se (R. Feldt).

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2008.12.005

1. Introduction

Search-based software engineering (SBSE) is the application of optimization techniques in solving software engineering problems [1,2]. The applicability of optimization techniques in solving software engineering problems is suitable as these problems frequently encounter competing constraints and require near optimal solutions. Search-based software testing (SBST) research has attracted much attention in recent years as part of a general interest in SBSE approaches. The growing interest in SBST can be attributed to the fact that generation of software tests is generally considered as an undecidable problem, primarily due to the many possible combinations of a program's input [3]. All approaches to SBST are based on satisfaction of a certain test adequacy criterion represented by a fitness function [2]. McMinn [3] has written a comprehensive survey on search-based software test data generation. The survey shows the application of metaheuristics in white-box, black-box and grey-box testing. Within the domain of non-functional testing, the survey indicates the application of metaheuristic search techniques for checking the best-case and worst case execution times (BCET, WCET) of real-time systems. McMinn highlights possible directions of future research into non-functional testing, which includes searching for input situations that break memory or storage requirements, automatic detection of memory leaks, stress testing and security testing. Our work extends the survey by McMinn [3] as it analyses actual evidence supporting McMinn's ideas of future directions in search-based testing of non-functional properties. Moreover, we anticipated studies making use of search-based techniques to test non-functional properties not highlighted by McMinn. This work also supports McMinn's survey by finding further evidence into search-based execution time testing. Another review by Mantere and Alander [4] highlights work using evolutionary computation within software engineering, especially software testing. According to the review, genetic algorithms are highly applicable in testing coverage, timings, parameter values, finding calculation tolerances, bottlenecks, problematic input combinations and sequences. This study also extends and supports Mantere and Alander's review in actually finding the evidence in support of proposed future extensions.

Within non-functional search-based software testing (NFSBST) research, it is both important and interesting to know the extent of application of metaheuristic search techniques to non-functional testing, not covered by previous studies. This allows us to identify potential non-functional properties suitable for applying these techniques and provides an overview of existing non-functional properties tested using metaheuristic search techniques. In this paper, after identifying existing non-functional properties, we review each of the properties to determine any constraints and limitations. We also identify the range of different fitness functions used within each non-functional property, since the fitness function is crucial in guiding search into promising areas of solution space and is the differentiating factor between quality of different solutions. The contribution of this review is therefore an exploration of non-functional properties tested using metaheuristic search techniques, identification of constraints and limitations encountered and an analysis of different fitness functions used to test individual non-functional property.

Section 2 describes the method of our systematic review that includes the research questions, search strategy, study selection criteria, study quality assessment and data extraction. Sections 3 and 4 discusses the results, synthesis of findings, areas of future research and validity threats. Conclusions are presented in Section 6.

2. Method

A systematic review is a process of assessment and interpretation of all available research related to a research question or subject of interest [5]. Kitchenham [5] also describes several reasons of undertaking a systematic review, the most common are to synthesize the available research concerning a treatment or technology, identification of topics for further investigation and formulation of a background in positioning new research activities.

This section describes our review protocol, consisting of several steps as outlined in Kitchenham [5].

2.1. Research questions

In order to examine the evidence of testing non-functional properties using metaheuristic search techniques, we have the following research questions:

- *RQ 1.* In which non-functional testing areas have metaheuristic search techniques been applied?
  After having identified these areas, we have three additional research questions applicable in each area:
- *RQ 1.1.* What are the different metaheuristic search techniques used for testing each non-functional property?
- *RQ 1.2.* What are the different fitness functions used for testing each non-functional property?
- *RQ 1.3.* What are the current challenges or limitations in the application of metaheuristic search techniques for testing each non-functional property?

The *population* in this study is the domain of software testing. *Intervention* includes application of metaheuristic search techniques to test different types of non-functional properties. The *comparison intervention* is not applicable in our case as our research questions are not aimed at making a comparison. However, we discuss the comparisons within the scope of each primary study to support our argumentation of obtained results in Section 4. The *outcome* of our interest represents different types of non-functional testing that use metaheuristic search techniques. In terms of *context* and *experimental design*, we do not enforce any restrictions.

2.2. Generation of search strategy

The search strategy was based on the following steps:

(i) *Identification of alternate words and synonyms for terms used in the research questions.* This is done to minimize the effect of differences in terminologies.
(ii) *Identify common non-functional properties for searching.* We take non-functional properties as to encompass the three aspects of software quality defined in ISO/IEC 9126-1 [6]. These aspects are quality in use, external quality and internal quality. Quality in use refers to software product quality in a specific context of use, while external quality is the

quality observable at software execution. Lastly, internal quality is measured against the internal quality requirements.

Since there are different systems of categorizing non-functional properties, we take guidance from four existing taxonomies to aid our search strategy and to have a representative set of non-functional properties. These are Boehm software quality model (as described in Fenton [7]), ISO/IEC 9126-1 [6], IEEE Standard 830-1998 [8] and Donald G. Firesmith's taxonomy [9]. The non-functional properties used for searching are usability, safety, robustness, capacity, integrity, efficiency, reliability, maintainability, testability, flexibility, reusability, portability, interoperability, security, performance, availability and scalability. To cover other potential non-functional properties, we explicitly used the term 'non-functional' in our search strings.

The non-functional properties obtained from existing taxonomies are restricted to high-level external attributes only for the sole purpose of guiding the search strategy. The different non-functional testing areas that are discussed later in the paper cannot be mapped one to one with these listed non-functional properties. Therefore, while quality of service includes attributes; namely availability and reliability, we have retained the term quality of service for the later part of the paper to better reflect the terms as used by the original authors. Similarly, one can argue execution time to fit under performance, but we stick to the term execution time in the later part of the paper to remain consistent with the terms used by the original authors.

(iii) *Use of Boolean* `OR` *to join alternate words and synonyms.*
(iv) *Use of Boolean* `AND` *to join major terms.*

We used the following search terms:

- *Population:* testing, software testing, testing software, test data generation, automated testing, automatic testing.
- *Intervention:* evolutionary, heuristic, search-based, metaheuristic, optimization, hill-climbing, simulated annealing, tabu search, genetic algorithms, genetic programming.
- *Outcomes* non-functional, safety, robustness, stress, security, usability, integrity, efficiency, reliability, maintainability, testability, flexibility, reusability, portability, interoperability, performance, availability, scalability

We used a two-phase strategy for searching. In the first phase, we searched electronic databases and performed a manual search of specific conference proceedings and journals. We selected 1996 as the starting year for the search since this year marked the first publication of the application of genetic algorithms to execution time testing [3] (one of the earliest non-functional properties to be tested using metaheuristic search techniques). We searched within the following electronic databases:

- IEEEXplore.
- EI Compendex.
- ISI Web of Science (WoS).
- ACM Digital Library.

In the first phase of the search strategy, we piloted the search strings thrice for the year 2007, each time refining them to eliminate irrelevant hits. We found it as a useful activity to pilot the search strings in iterations as it resulted in much refinement of search results. It also helped us to deal with the challenging task of balancing comprehensiveness versus precision of our search. We applied separate search strings for searching within titles, abstracts and keywords. Complete search strings are given in Appendix A.

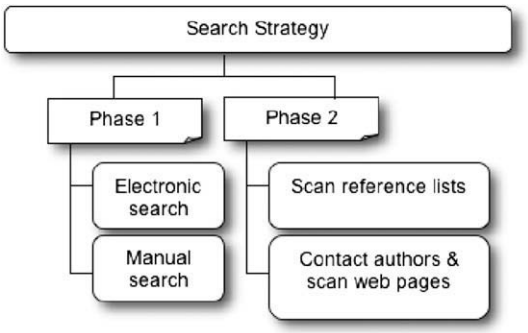

Fig. 1. The two-phase search strategy.

We manually searched selected journals (J) and conference proceedings (C). These journals and conferences were chosen as they had previously published primary studies relevant to our domain. They include: real time systems symposium (RTSS) (C), real-time systems (RTS) (J), genetic and evolutionary computation conference (GECCO)[1] – Search-based software engineering (SBSE) track (C), software testing, verification and reliability (STVR) (J) and software quality journal (SQJ) (J).

We initiated a second phase of search to have a more representative set of primary studies. In this phase, we scanned the reference lists of all the primary studies to identify further papers. We then contacted the researchers who authored most of the papers in a particular non-functional area for additional papers. Moreover, we scanned the personal web pages maintained by these researchers. A total of four researchers were contacted. Fig. 1 shows our two-phase search strategy.

In order to assess the results of the search process, we compared the results with a small sample of primary studies we already knew about [10–12], to ensure that the search process was able to find the sample (as described in [13]). All the three known papers were found using nine sources, namely (IEEE Xplore, Compendex, web of science, ACM digital library, real-time systems symposium (C), real-time systems (J), GECCO SBSE track (C), software testing, verification and reliability (J), software quality journal (J)).

2.3. Study selection criteria and procedures for including and excluding primary studies

Metaheuristic search techniques have been applied across different engineering and scientific disciplines. Within software testing, metaheuristic search techniques have found application in different phases, from planning to execution. Therefore, it is imperative that we define comprehensive inclusion/exclusion criteria to select only those primary studies that provide evidence related to the research questions. The following exclusion criteria is applicable in this review, i.e. exclude studies that:

- Do not relate to software engineering/development.
- Do not relate to software testing.
- Do not report application of metaheuristics. (We consider metaheuristics to include hill-climbing, simulated annealing, tabu search, ant colony methods, swarm intelligence and evolutionary methods [14].)
- Describe search-based testing approaches, which are inherently structural (white-box), functional (black-box) or grey-box (combination of structural and functional).

[1] GECCO was not part of ACM until 2005.

Grey-box testing includes assertion testing and exception condition testing [3]. This exclusion criterion is relaxed to include those studies where a structural test criterion is used to test non-functional properties, e.g. [15].
- Are not related to the testing of the end product, e.g. [16].
- Are related to test planning, e.g. [17].
- Make use of model checking and formal methods, e.g. [18,19].
- Report performance of a particular metaheuristic instead of its application to software testing, e.g. [20].
- Report on test case prioritization, e.g. [21].
- Are used for prediction and estimation of software properties, e.g. [22].

The first phase of research resulted in a total of 501 papers. After eliminating duplicates found by more than one electronic database, we were left with 404 papers. Table 1 shows the distribution of papers before duplicate removal among different sources.

The exclusion was done using a tollgate approach (Fig. 2). To begin with, a single researcher excluded 37 references out of a total of 404 primarily based on title and abstract, which were clearly out of scope and did not relate to the research question. The remaining 367 references were subject to detailed exclusion criteria, which involved three researchers. First, each researcher applied the exclusion criteria independently. Out of 367 references, the three researchers were in agreement on 229 references to exclude, 25 to include and 113 required a meeting to reach consensus. In the meeting, the researcher in the minority for a paper tried to convince others; otherwise the majority decision was taken. This application of detailed exclusion criteria resulted in 60 remaining references, which were further filtered out by reading full-text. A final figure of 24 primary studies was reached after excluding similar studies that were published in different venues. The 24 primary studies were complemented with 11 more papers from phase 2 of the search strategy (Fig. 1). The fact that we gathered 11 papers from phase 2 of the search strategy indicates that making a generic search string that would give the entire relevant set of primary studies from searching only within electronic databases is difficult in the field of study under investigation. The terminologies used by various authors differed a lot; both in terms of specifying the non-functional property and the used metaheuristic. As an example, if we consider the primary study [23], identified using phase 2, we observed that although it does mention using *genetic programming* in the title, it does not mention the target non-functional property of *security*. Similarly, in the abstract and key words, we do not find words synonymous to *testing*. Therefore, we believe that the phase 2 of the search strategy helped us to gather a more representative set of primary studies.

Table 1
Distribution of papers before and after duplicate removal among different publication sources.

Source	Count
IEEE Xplore	209 (179)
EI Compendex	140 (87)
ISI Web of Science	61 (48)
ACM Digital Library	58 (57)
Conferences and Journals	33 (33)
Total	501 (404)

2.4. Study quality assessment and data extraction

Since we did not impose any restriction in terms of any specific research method or experimental design, therefore the study quality assessment covered both quantitative and qualitative studies. The quality data can be used to devise a detailed inclusion/exclusion criteria and/or to assist data analysis and synthesis [5]. We applied the study quality assessment primarily as a means to guide the interpretation of findings for data analysis and synthesis [5], so as to avoid any misinterpretation of results due to study quality. We did not assign any scores to the criterion (because our aim was

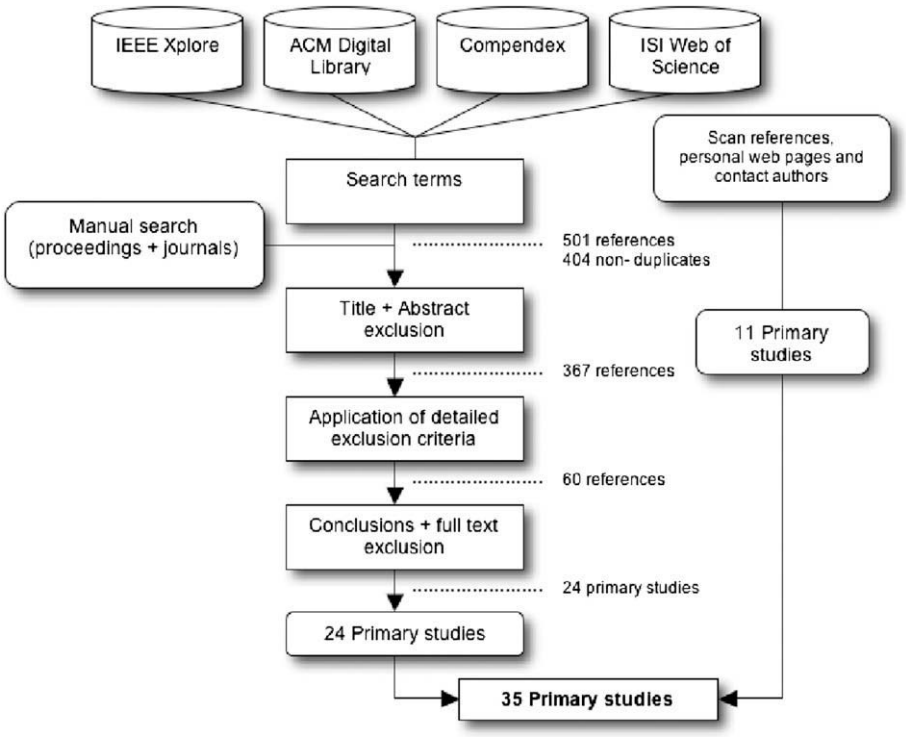

Fig. 2. Multi-step filtering of studies (tollgate approach) and final number of primary studies.

not to rank studies according to an overall quality score) but used a binary 'yes' or 'no' scale [24]. Table 15 in Appendix A shows the application of the study quality assessment criteria where a ($\surd$) indicates 'yes' and ($\times$) indicates 'no'. Most of our developed criteria were fulfilled by all of the studies, exceptions being [34,35,12,15,54] where evidence of a comparison group was missing, but these quality differences were not found to be largely confounded with study outcomes. What follows next is the list of developed criteria:

- Is the reader able to understand the aims of the research?
- Is the context of study clearly stated, that includes population being studied (e.g. academic vs. industrial) and tasks to be performed by population (e.g. small scale vs. large scale)
- Was there a comparison or control group?
- Are the measures used in the study fully defined [5]?
- Is there an adequate description of the data collection methods?
- Does the data collection methods relate to the aims of the research?
- Is there a description of the method used to analyze data?
- Are the findings clearly stated and relate to the aims of research?
- Is the research useful for software industry and research community?
- Do the conclusions relate to the aim and purpose of research defined?

We designed a data extraction form to collect information needed to address the review questions and data synthesis. Study quality data was not part of data extraction form as it was assessed separately. To assess the consistency of data extraction, a small sample of primary studies were used to extract data for the second time. In addition to the standard information of title, author(s), journal and publication details; the data extraction form included information about main theme of study, motivation for the main theme, type of non-functional testing addressed, type of metaheuristic search technique used, examples of application of approach, constraints/limitations in the application of the metaheuristic search technique, identified areas of future research and major conclusion. For each primary study, we further extracted the information relating to the method of evaluation, number of test objects, performance factor evaluated and the experimental outcomes.

3. Results and synthesis of findings

In this section we describe the descriptive evaluation of the assessed literature in relation to the research questions. The 35 primary studies were related to the application of metaheuristic search techniques for testing five non-functional properties: execution time, quality of service (QoS), security, usability, and safety. The number of primary studies describing each non-functional property is: 15 (execution time), 2 (quality of service), 7 (security), 7 (usability) and 4 (safety). Relevant information describing the distribution of primary studies within each non-functional property is shown in Table 2.

Fig. 3 shows the year-wise distribution of primary studies within each non-functional property as well as the frequency of application of different metaheuristics [55]. The bubble at the intersection of axes contains the reference number(s) of the contribution(s). It is evident from the figure that genetic algorithms are the most widely used metaheuristic with applications in 21 papers across different types of non-functional testing. In the left quadrant of Fig. 3, each bubble represents the reference numbers of primary studies within each non-functional area in respective years from 1996 to 2007. More details on Fig. 3 can be found in [55] which is a systematic mapping study, giving a broad overview of studies without reviewing the studies in detail.

Table 2
Distribution of primary studies per non-functional area.

Non-functional property	Author(s)	Year	References
Execution time (42.86%)	Wegener et al.	1996	[25]
	Alander et al.	1997	[26]
	Wegener et al.	1997	[27]
	Wegener et al.	1998	[10]
	O'Sullivan et al.	1998	[28]
	Tracey et al.	1998	[29]
	Mueller et al.	1998	[30]
	Puschner et al.	1998	[31]
	Pohlheim et al.	1999	[32]
	Wegener et al.	2000	[33]
	Groß et al.	2000	[34]
	Groß	2001	[35]
	Groß	2003	[36]
	Briand et al.	2005	[12]
	Tlili et al.	2006	[11]
Quality of service (5.71%)	Canfora et al.	2005	[37]
	Di Penta et al.	2007	[38]
Security (20%)	Dozier et al.	2004	[39]
	Kayacik et al.	2005	[40]
	Budynek et al.	2005	[41]
	Del Grosso et al.	2005	[42]
	Kayacik et al.	2006	[43]
	Kayacik et al.	2007	[23]
	Del Grosso et al.	2007	[44]
Usability (20%)	Stardom	2001	[45]
	Cohen et al.	2003	[46]
	Cohen et al.	2003	[47]
	Cohen et al.	2003	[48]
	Nurmela	2004	[49]
	Shiba et al.	2004	[50]
	Bryce et al.	2007	[51]
Safety (11.43%)	Tracey et al.	1999	[52]
	Abdellatif-Kaddour et al.	2003	[53]
	Baresel et al.	2003	[15]
	Pohlheim et al.	2005	[54]

3.1. Execution time

The application of evolutionary algorithms to test real-time requirements in embedded computer systems involves finding the best and worst case execution times (BCET, WCET) to determine if timing constraints are fulfilled. A violation of the timing constraint or temporal error means that either the outputs are produced too early, or their computation takes too long [10]. The use of evolutionary computation to find the input situations causing longest or shortest execution times is an example of evolutionary testing. Evolutionary testing is seen as a promising approach for verifying timing constraints and a number of studies proving the efficacy of the approach can be found in literature. This dynamic approach to verify timing constraints involves testing the run-time behavior of an embedded system based on execution in an application environment. Testing of real-time systems is found to be costly as compared to conventional applications as additional requirements of timeliness, simultaneity and predictability needs to be tested. Although there are numerous methods to test logical correctness, the lack of support for testing temporal behavior [30] motivated the use of evolutionary computation in testing extreme execution times.

In Wegener et al. [25], genetic algorithms (GA) were used to search for input situations that produce very long or very short

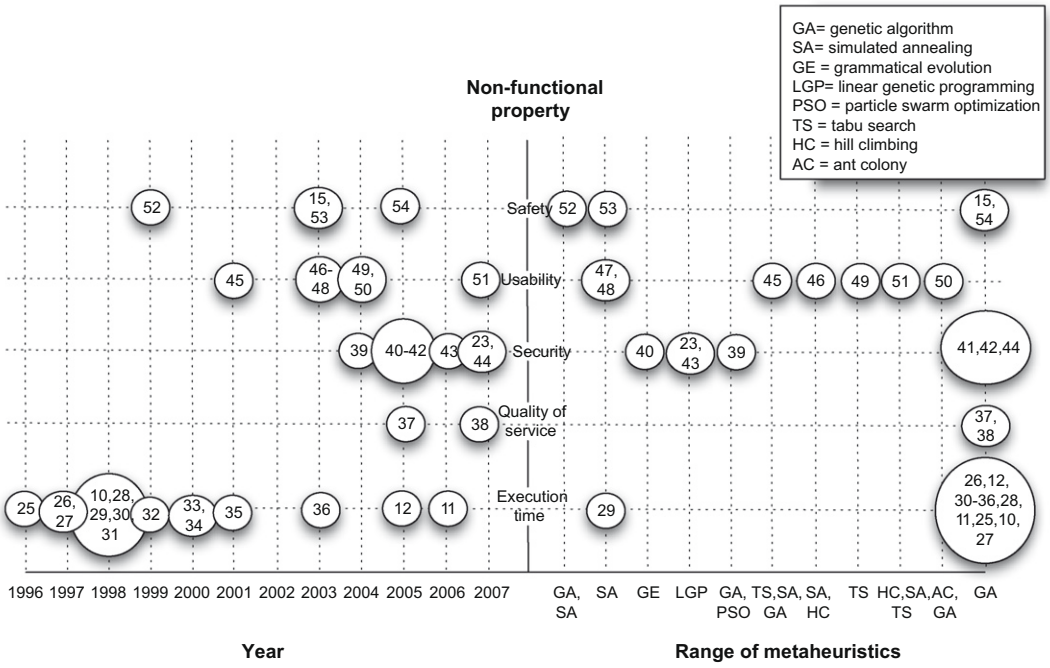

Fig. 3. Distribution of NFSBST research over range of applied metaheuristics and time period (adapted from [55]).

execution times. The fitness function used was the execution time of an individual measured in micro seconds. The experimental results using a simple C function showed that the longest execution time of 26.27 μs was found very quickly with GA in less than 20 generations. Moreover, a new shortest execution time of 5.27 μs, which was not discovered by statistical and systematic testing, was found. This study marks one of the earliest use of GA to test temporal correctness of real-time systems.

Alander et al. [26] presented experiments performed in a simulator environment to measure response time extremes of protection relay software using genetic algorithms. The fitness function used was the response time of the tested software. The results showed that GA generated more input cases with longer response times. In Wegener et al. [27], experiments were performed using genetic algorithms involving five test objects from different application domains having varying lines of code and integer input parameters. This time the fitness function used was the execution time measured in terms of processor cycles rather than seconds. The results show that GA consistently outperformed random testing by finding more extreme times.

The research community soon realized the benefits of measurement in terms of processor cycles; being more precise and independent of the interrupts from the operating system (e.g. context switching and paging). Also measurement in terms of processor cycles is deterministic in the sense that it is independent of system load and results in the same execution times for the same set of input parameters. However, such a measurement is dependent on the compiler and optimizer used, therefore, the processor cycles differ for each platform [27].

Wegener and Grochtmann [10] continued further with experimentation to compare evolutionary testing (using genetic algorithms) with random testing. The fitness function used was duration of execution measured in processor cycles. This time the range of input parameters was raised to 5000. The results showed that, with a large number of input parameters, evolutionary testing obtained more extreme execution times with less or equal testing effort than random testing. In order to better evaluate the application of evolutionary testing, Groß et al. [56] presented the design of an operational experimental environment for evolutionary testing with the integration of a commercially available timing package.

The aforementioned experimental results identified several limitations when using evolutionary testing. The instrumentation required for the software under test (SUT), which extends the executable programming code by inserting hardware dependent counting instructions, is bound to affect the execution times. Also as a typical search strategy, it is difficult to ensure that the execution times generated in the experiments represents global optimum. More experimentation is also required to determine the most appropriate and robust parameters. Lastly, there is a need for an adequate termination criterion to stop the search process.

In Sullivan et al. [28], cluster analysis was used on the population from the most recent generation to determine if GA should terminate. Execution time measured in processor cycles was used as a fitness function and a complex algorithm from the domain of automotive electronics was used for seven runs of GA. Cluster analysis was performed on the final population (generation 399) of each run. With cluster analysis, it was possible to examine which of the test runs converged to local optima and thus continuing with these runs would not yield better results. The results of the study demonstrated the potential of incorporating cluster analysis as a useful termination criterion for evolutionary tests and suggested appropriate changes in the search strategy to include cluster analysis information.

Tracey et al. [29] applied simulated annealing (SA) to test four simple programs for WCET as part of a generalized test data generation framework. The fitness function used is the measure of actual dynamic execution time. The WCET of the programs was already known and a valid test case was one that exercised a path yielding the already known WCET. The results of the experiment showed that the use of SA was more effective with larger parameter space. The authors highlighted the need of a detailed comparison of various optimization techniques to explore WCET and BCET of the SUT. With this goal in mind, Pohlheim and Wegener [32] used an extension of genetic algorithms making use of multiple sub-populations, each using a different search strategy. The authors name

the approach as extended evolutionary algorithms. The duration of execution measured in processor cycles was taken as the fitness function. The extended evolutionary algorithm was applied on two test objects. The first test object was the bubble sort algorithm and the results from this experiment was used to find appropriate evolutionary parameters for the second test object which contained software modules from a motor control project. The evolutionary algorithm found longer execution times for all the given modules in comparison with systematic testing.

As mentioned earlier, it is difficult to ensure that the execution times generated in the experiments represent global optimum. Therefore it appears interesting to compare the results of evolutionary testing with static analysis to find a bound within which WCET and BCET might lie. Mueller et al. [30] presented such a comparison. Both approaches were used in five experiments to determine the WCET and BCET of different programs. Three programs were from real-time systems while the remaining two were general-purpose algorithms. The fitness function used is the execution time measured in processor cycles. The results showed that methods of static analysis and evolutionary testing bound the actual execution times. For WCET, the estimates of static analysis provided an upper bound while the measurements of evolutionary testing yielded a lower bound. Conversely, static analysis' estimates provided a lower bound for BCET while evolutionary testing measurements constituted an upper bound. In Puschner and Nossal [31], genetic algorithms were applied to find WCET for seven programs and the results were compared with those from random search, static analysis and best effort timings that were researchers' own efforts to find input data to yield WCET. The execution time measured in processor cycles as well as time units were used as a fitness function for different programs. Genetic algorithms found same or longer times than random search. In comparison with best effort timings, genetic algorithms matched the timings and found a longer time in one case, while in comparison with static analysis, the upper bounds were not broken but were matched on several occasions. In another study, Wegener et al. [33] used genetic algorithms to test temporal behavior of six time critical tasks in an engine control system, with the fitness function used was execution time measured in processor cycles. Genetic algorithms outperformed both random search and developer-made tests.

In Groß [36], 15 example test programs were used in experiments to measure the maximal execution times using genetic algorithms. The fitness function used was the execution time of the test object for a particular input situation measured in microseconds. The results of evolutionary testing were compared with random testing and with the performance of an experienced human tester. The results indicated that evolutionary testing outperforms random testing as random testing could only produce about 85% of the maximum execution times found by evolutionary testing. The human tester was more successful in 4 out of 15 test programs, which indicated the presence of properties of test objects that inhibit evolutionary testability [35] i.e. the ability of an evolutionary algorithm to successfully generate test cases that violates the timing specification.

Groß et al. [34] presented a prediction model based on complexity of the test object, which can be used to predict evolutionary testability. It was found that there were several properties inhibiting evolutionary testability, which included small path domains, high-data dependence, large input vectors, and nesting.

Additionally, several source code measures, which map program attributes inhibiting evolutionary testability, were also presented in Groß [35]. Code annotations were inserted into the test object's source code along their shortest and longest algorithmic execution paths. The fitness function was then based on maximal and minimal possible annotation coverage by the generated input situations. The individual measures were combined to form a prediction system that successfully forecasted evolutionary testability with 90% accuracy.

Results from the two studies [34,35] also confirmed that there is a relationship between the complexity of a test object and the ability of evolutionary algorithm to produce input parameters according to B/WCET. The results also confirmed the properties (given above) of the test programs that caused most problems for evolutionary testing. Due to program properties inhibiting evolutionary testability [36], pointed out that an ideal testing strategy is a combination of evolutionary testing supported by human knowledge of the test object. The initial population of individuals can benefit from human knowledge to direct the search in those areas of search space that are difficult to reach as the fitness function does not provide information to generate such unlikely input combinations.

In one of the more recent studies, Tlili et al. [11] used the approach of seeding an evolutionary algorithm with test data achieving a high structural coverage and reduction in the amount of search space by restricting the range of input variables in the initial population. The fitness function was the measurement of the execution time of test data as number of CPU clock ticks. The results indicated that for almost all the test objects, application of seeding and range restriction outperform standard evolutionary real-time testing with random initial population when measuring long execution times. Also with seeding and range restriction, fewer generations found the longest execution times. Similar results were achieved for finding the shortest execution times.

In Briand et al. [12,57], another approach to use genetic algorithms for critical deadlines misses was used. The authors called the approach stress testing because the system was exercised in such a way that some tasks were close to missing a deadline. This approach can also be called robustness testing but since the basic objective of the paper was to find the sequence of arrival times of events for aperiodic tasks, which will cause the greatest delays in the execution of the target task, we choose to discuss this paper under execution time. The study was restricted to seeding times for aperiodic tasks and the tasks synchronization, since input data were accounted for in execution time estimates. In comparison with other approaches to evolutionary testing for finding the WCET, this approach was different in the sense that test data design did not require the implementation of the system and, secondly, did not consider the tasks in isolation. Genetic algorithms were used to search for the sequence of arrival times of events for aperiodic tasks that would cause the greatest delays in execution of the target task. The fitness function was expressed in an exponential form, based on the difference between the deadline of an execution and the executions actual completion. A prototype tool called real-time test tool (RTTT) was built to facilitate the execution of runs of genetic algorithm. Two case studies were conducted; one case study consisted of researchers own scenarios while the second consisted of an actual real-time system. The results from the timing diagrams illustrated that RTTT was a useful tool to stress the system more than the scenarios covered by schedulability theory.

A summary of results of applying metaheuristics for testing temporal properties is given in Table 3.

3.2. Quality of service

Search-based testing of Quality of Service (QoS) represents a mix of search-based software engineering and service-oriented software engineering. Metaheuristic search techniques have been used for quality of service aware composition and violation of service level agreements (SLAs) between the integrator and the end user.

Table 3
Summary of results applying metaheuristics for testing temporal properties. The last column on the right covers any issues such as constraints, limitations and highlights. (GA is short for Genetic Algorithm, SA is short for Simulated Annealing while EGA is short for Extended GA.)

Article	Applied metaheuristic	Fitness function used	Limitations and highlights
Wegener et al. [25]	GA	Exec. time, microseconds	**Instrumentation** of the test objects causes probe effects. **The** execution times do not always represent global optimum.
Alander et al. [26]	GA	Exec. time, milliseconds	**The** experiments are performed in a simulator environment. **Non-determinism** of the fitness function is problematic.
Wegener et al. [27]	GA	Exec. time, processor cycles	**The** decision about when to stop the search is arbitrary.
Wegener and Grochtmann [10]	GA	Exec. time, processor cycles	**There** is a need to find most appropriate and robust parameters for evolutionary testing. **Cluster** analysis can be a used as a measure for termination of search.
O'Sullivan et al. [28]	GA	Exec. time, processor cycles	**The** search strategy needs to make use of cluster analysis to react to stagnations.
Tracey et al. [29]	SA	Exec. time, time units	**Ways** to reduce the amount of search space is useful. **There** is a need to devise different software metrics to guide the search.
Pohlheim and Wegener [32]	EGA	Exec. time, processor cycles	**A** combination of systematic and evolutionary testing is required for thoroughly testing real-time systems. **Instead** of random generation of initial population, use of testers knowledge improves search performance.
Mueller and Wegener [30]	GA	Exec. time, processor cycles	**For** WCET, estimates of static analysis provide an upper bound, evolutionary testing gives a lower bound. **For** BCET, static analysis estimates provide a lower bound, evolutionary testing constitutes an upper bound.
Puschner and Nossal [31]	GA	Exec. time, processor cycles & time units	**Further** investigation is required to escape the large plateaus of equal fitness function values.
Wegener et al. [33]	GA	Exec. time, processor cycles	**Static** analysis techniques can support evolutionary testing in search space reduction.
Groß [36]	GA	Exec. time, microseconds	**Evolutionary** testability is inhibited by source code properties of small path domains and high-data dependence.
Groß [35]	GA	Coverage of code annotations along shortest and longest execution paths	**The** measures did not cater for the reliance on the parameter setting of the GA. **Effects** of underlying hardware are not taken into account. **The** number of considered data samples is low.
Groß et al. [34]	GA	Coverage of code annotations along shortest and longest execution paths	**Traditional** design principles of low coupling and high cohesion are an important issue for an evolutionary approach.
Tlili et al. [11]	EGA	Exec. time, processor cycles	**The** nature of the test objects is not evident in the study. **Using** branch coverage as a criterion for seeding has the limitation that it does not handle the execution of all the possible values of the predicates forming the conditions.
Briand et al. [12]	GA	Exponential fitness function based on the difference between executions deadline and executions actual completion	**The** termination criterion is not adaptive and is taken as fixed number of generations. **The** specification of test cases does not require any running implementation of system. **It** takes into account tasks synchronizations.

In Canfora et al. [37], genetic algorithms were used to determine the set of service concretizations (i.e. bindings between abstract and concrete services) that lead to QoS constraint satisfaction while optimizing the objective function. An abstract service is the feature required in a service orchestration while concrete services represent functionally equivalent services realizing the required feature. In a contract with potential users, the service provider can estimate ranges for the QoS attributes as part of service level agreement (SLA), i.e. a contract between an integrator and end-user for a given QoS level. QoS attributes consist of non-functional properties such as cost, response time and availability so the fitness function optimized the QoS attribute chosen by the service integrator. The QoS attributes of composite services were determined using rules with aggregation function for each workflow construct. The fitness function was designed in a way to maximize some QoS attributes (e.g. reliability and availability) while minimizing others (e.g. cost and response time). Based on the type of penalty factor (static vs. dynamic), static fitness function and dynamic fitness functions were proposed. With experiments on 18 invocations of 8 distinct abstract services, the performance of genetic algorithms was compared with integer programming. The results showed that when the number of concrete services is small, integer programming outperformed genetic algorithms. But as the number of concrete services increased, genetic algorithm was able to keep its time performance while integer programming grew exponentially.

Di Penta et al. [38] used genetic algorithms to generate test data that violated QoS constraints causing SLA violations. The generated test data included combinations of inputs and bindings for the service-oriented system. The test data generation process was composed of two steps. In the first step, the risky paths for a particular QoS attribute were identified and in the second step, genetic algorithms were used to generate test cases that covered the path and violated the SLA. The fitness function combined a distance-based fitness that rewards solutions close to QoS constraint violation, with a fitness guiding the coverage of target statements. The two fitness factors were dynamically weighted. The approach was applied to two case studies. The first case study was an audio processing workflow containing invocations to four abstract services. The second case study, a service producing charts, applied the black-box approach with fitness calculated only on the basis of how close solutions violate QoS constraint. In case of audio workflow, the genetic algorithm using the proposed fitness function, which combined distance-based fitness with coverage of target statements, outperformed random search. For the service producing charts, use of black-box approach successfully violated the response time constraint, showing the violation of QoS constraints for a real service available on the Internet.

A summary of results of applying metaheuristics for QoS-aware composition and violation of SLA is given in Table 4.

3.3. Security

A variety of metaheuristic search techniques have been applied to detect security vulnerabilities like detecting buffer overflows; including grammatical evolution, linear genetic programming, genetic algorithm and particle swarm optimization.

In Kayacik et al. [40], grammatical evolution (GE) was used to discover the characteristics of a successful buffer overflow. The example vulnerable application in this case performed a data copy without checking the internal buffer size.

The exploit was represented by a sample C program that approximated the desired return address and assembled the malicious buf-

Table 4
Summary of results applying metaheuristics for QoS-aware composition and violation of SLA. The last column on the right covers any issues such as constraints, limitations and highlights. (GA is short for Genetic Algorithm.)

Article	Applied metaheuristic	Fitness function used	Limitations and highlights
Canfora et al. [37]	GA	Based on the maximization of desired QoS attributes while minimizing others, including a static or dynamic penalty function	**The** QoS attributes of component services needs to be computed for workflow constructs. **The** fitness function needs to incorporate the constraints of balancing different QoS attributes. Also weights need to be assigned to a particular QoS attribute to indicate the importance.
Di Penta et al. [38]	GA	Combination of distance based fitness that rewards solutions close to QoS constraint violation with a fitness guiding the coverage of target statements	**The** study does not deal with the dependency of violation of some QoS attributes on the network and server load. **Instrumentation** of the workflow can cause probe effects which can cause the deviation of fitness calculation.

fer exploit. The malicious buffer contained a shell code, representing attacker's arbitrary code, that overwrites the return address to gain control.

For the attack to be successful, it was important to jump to the first instruction of the shell code or to the sequence of no operation instructions called NoOP sled. The fitness function represented six characteristics of malicious buffer which included existence of the shell-code, success of the attack, NoOP sled score, back-to-back desired return addresses, desired return address accuracy and score calculated on NoOP sled size. Three sets of experiments were performed, namely basic grammatical evolution (GE), GE with niching (to include population diversity) and GE with niching and NoOP minimization. The results found were comparable. In Kayacik et al. [43], the vulnerable system call was taken to be the UNIX `execve` command and linear GP was used to evolve variants of an attack. The UNIX `execve` command required the registers `EAX`, `EBX`, `ECX`, `EDX` to be correctly configured and the stack to contain the program name to be executed. The fitness function returned a maximum fitness of 10 if all conditions were satisfied. The experimental results indicated that evolved attacks discovered different ways of attaining sub-goals associated with building buffer overflow attacks and expanding the instruction set provided better results as compared to basic GP.

Kayacik et al. [23] used linear GP for automatic generation of mimicry attacks to perform evasion of intrusion detection system (IDS), which in this case was an open source target anomaly detector called `stride`. The candidate mimicry attacks were in the form of system call sequences. The system call sequences consisted of most frequently executed instructions from the vulnerable application, which in this case is `traceroute`, a tool used to determine the route taken by packets across an IP network.

An acceptable anomaly rate was established for `stride`. The objective of the attacker therefore was to reduce the anomaly rate below this acceptable limit. The study described an attack denoted by successful completion of three steps i.e. open a UNIX password file, write a line and close the file. The fitness function rewarded attacks that successfully followed the steps and at the same time minimized the anomaly rate. The results showed that the approach was able to reduce the anomaly rate to ~2.97% for the entire attack.

In Dozier et al. [39], security vulnerabilities in an artificial immune system (AIS) based IDS were identified using genetic algorithms and particle swarm optimization. The study used GENERITA Red Team (GRT), a system based on evolutionary algorithms, which performed the vulnerability analysis of the IDS. The AIS-based IDS communicates with the GRT by receiving red packets in the form of attacks and returns the percentage of the detector set that failed to detect the red packet. This percentage was the fitness of the red packet. The packets took the form of triplets (`ip_address`, `port`, `src`) while the AIS maintained a population of detectors with different ranges of IP addresses and ports. Matching rules were applied to match the data triple and a detector. The GRTs used consisted of a steady-state GA and six variants of particle swarm optimization. Experiments were performed using data representing 35 days of simulated network traffic. The results showed that genetic algorithms outperform all of the swarms with respect to the number of distinct vulnerabilities discovered.

Budynek et al. [41] modeled the hacker behavior along with the creation of a hacker grammar for exploring hacker scripts using genetic algorithms. A hacker script contained sequences of UNIX commands issued by the hacker upon logging in to the system. One script was one individual with a single UNIX command acting as a gene. A fitness function was defined based on the efficiency and effectiveness of the hacking scripts i.e. the script fitness value was calculated by number of goals achieved, number of pieces of evidence discovered by the log analyzer, number of bad commands used by the hacker and the length of the script used by the hacker. The results of experiments showed various top-scoring scripts obtained from various runs.

Grosso et al. [42] used static analysis and program slicing to identify vulnerable statements and their relationships, that were further explored using genetic algorithms for buffer overflows. Three different fitness functions were compared. The first one (vulnerable coverage fitness) included weighted values for statement coverage, vulnerable statement coverage and number of executions of vulnerable statements. The second fitness function (nesting fitness) incorporated observed maximum nesting level corresponding to the current test case while the third fitness function (buffer boundary fitness) included a term accounting for the distance from the buffer boundaries. Two programs were used for experimentation. The case in which the expert's knowledge was used to define the initial search space and also for the case having random initial population showed that buffer boundary fitness outperformed both the vulnerable coverage and the nesting fitness. This showed that fitness functions using distance from the limit of buffers were helpful for deriving genetic algorithm evolution. In DelGrosso et al. [44], the authors improved on the previous basic boundary fitness [42] to propose a dynamic weight fitness in which the genetic algorithm weights were calculated by solving a maximization problem via linear programming. So with weights that could be tuned at each genetic algorithm generation, fast discovery of buffer overflows could be achieved. The dynamic weight fitness outperformed the previous basic boundary fitness on experiments with two different sets of C applications.

A summary of results of applying metaheuristics for detecting security vulnerabilities is given in Table 5.

3.4. Usability

Usability testing in the context of application of metaheuristics is concerned with construction of covering array which is a combinatorial object.

The user is involved in numerous interactions taking place through the user interface. With the number of different features available and their respective levels, the interactions cannot be tested exhaustively due to a combinatorial explosion. Interaction

Table 5
Summary of results applying metaheuristics for detecting security vulnerabilities. The last column on the right covers any issues such as constraints, limitations and highlights. (GE is short for Grammatical Evolution, LGP is short for Linear Genetic Programming and PSO is short for Particle Swarm Optimization.)

Article	Applied metaheuristic	Fitness function used	Limitations and highlights
Kayacik et al. [40]	GE	Representation of six characteristics of malicious buffer reflecting multiple behavioral objectives	**The** shell code or the attackers arbitrary code needs to be modified to increase the success chances of malicious buffer.
Kayacik et al. [43]	LGP	Fitness function based on the configuration of registers and stack	**A** mechanism for maintaining diversity of population is necessary.
Kayacik et al. [23]	LGP	Evaluation in terms of completion of steps leading to an attack and minimization of anomaly rate	**The** proposed approach is dependent on the core attack which is then used to create mimicry attacks. **The** approach is also dependent on the set of permitted system calls as defined by the user.
Dozier et al. [39]	GA, PSO	Percentage of the detector set that failed to detect the red packet from GRT	**An** attack is not as such constructed but is represented as a triple packet.
Budynek et al. [41]	GA	The fitness is calculated based upon the scripts ability of how much damage it can inflict with the most compact possible sequence of commands	**The** goals of the hacker script grammar needs to be defined beforehand.
Grosso et al. [42]	GA	Three different fitness functions covering vulnerable statements, maximum nesting level and buffer boundary	**Dependency** on tools for static analysis and program slicing. **Use** of instrumentation is a probable obstacle in expanding the approach for larger case studies.
Grosso et al. [44]	GA	A dynamic weight fitness function in which the weight determination is a maximization problem	**Dependency** on tools for static analysis and program slicing. **Use** of instrumentation is a probable obstacle in expanding the approach for larger case studies. **Additional** computational time required for linear programming calculation.

testing offers savings, in that it aims to cover every combination of pair-wise (or t-way) interaction at least once [58]. It is interesting to see that there are different competing constraints. On one hand, the objective is high-coverage, while on the other hand the test suite size needs to be small to reduce overall testing cost. Covering array (CA) needs to be constructed to capture the t-way interactions. For software systems, each factor (feature or component) comprises of different levels (options or parameters or values), therefore a mixed level covering array (MCA) is proposed. However, as compared to CA, there are few results on the upper bound and construction algorithms for mixed level covering arrays, especially using heuristic search techniques [46]. Algebraic constructs, greedy methods and metaheuristic search techniques have been applied to construct covering arrays. Our interest here is to explore the use of metaheuristic search techniques for constructing covering arrays. Hoskins et al. provide definitions relevant to covering array and mixed level covering array [59]:

> *A covering array, $CA\lambda(N; t, k, v)$, is an $N \times k$ array for which every $N \times t$ sub-array has the property that every t-tuple appears at least λ times. In this application, t is the strength, k is the number of factors (degree), and v is the number of symbols for each factor (order). When λ is 1, every t-way interaction is covered at least once; this is the case of most interest, and we often omit the subscript λ when it is 1. The covering array is optimal if it contains the minimum possible number of rows. The size of such a covering array is the covering array number $CAN(t, k, v)$.*
> *A mixed level covering array, $MCA\lambda(N; t, k, (v_1, v_2, \ldots, v_k))$, is an $N \times k$ array in which, for each column i, there are exactly v_i levels; again every $N \times t$ sub-array has the property that each possible t-tuple occurs at least λ times. Again λ is omitted from the notation when it is 1. A mixed covering array provides the flexibility to construct test suites for systems in which components are not restricted to having the exact same number of levels.*

To adapt to the practical concerns of software testing, it is desirable that some subset of features have higher interaction coverage. For example, the overall system might have 100% two-way coverage, but a subset of features might have 100% three-way coverage. To this end, in Cohen et al. [46], Cohen et al. propose variable strength covering arrays. As with mixed level covering arrays, construction methods and algorithms for variable strength test suites is in its preliminary stages with [46] providing some initial results for test suite sizes constructed using simulated annealing (SA).

With respect to the application of metaheuristics, the fitness function used for constructing a covering array is the number of uncovered t-subsets, so the covering array itself will have a cost of 0. Since one does not know the size of the test suite *a priori*, therefore, heuristic search techniques apply transformations to a fixed size array until constraints are satisfied. The results of implementing SA for handling t-way coverage in fixed level cases are provided by Cohen et al. [46]. The results showed that in comparison with greedy search techniques used in test case generator (TCG) [60] and automatic efficient test generator (AETG) [61], SA improved on the bounds of minimum test cases in a test suite of strength two, e.g. for $MCA(N; 2, 5^1 3^8 2^2)$, SA gave 15 as minimum test cases as compared to 20 and 19 by TCG and AETG, respectively. In case of strength 3 constructions, the SA algorithm did not performe as well as the algebraic constructions. Therefore, the initial results indicated SA as more effective than other approaches for finding smaller sized test suites. On the other hand, SA took much more execution time as compared to simpler heuristics. Stardom [45] also used SA, GA and tabu search (TS) for constructing covering arrays. The results indicated that SA and TS were best in constructing covering arrays. A genetic algorithm turned out to be least effective; taking more time and moves to find good covering arrays. Stardom reported new upper bounds on size of covering array using SA, some of which were later improved by Cohen et al. [46]. Stardom's study indicated that SA's main advantage was the capability of executing many moves in a short time; therefore if the search space was dense, SA quickly located objects. On the other hand, TS performed much better when the size of an array's neighborhood was smaller.

Along with the application of metaheuristic techniques for constructing covering arrays, there is also evidence of integrated approaches (Table 6). Cohen et al. [46] proposed one such approach for using algebraic construction along with search techniques. An example of this integrated approach is given in Cohen and Ling [48] where a new strategy called augmented annealing takes advantage of computational efficiency of algebraic construction and generality of SA. Specifically, algebraic construction reduced a problem to smaller sub-problems on which SA runs faster. The experimental results reported new bounds for some strength three covering arrays e.g. $CA(3, 6, 6)$ and $CA(3, 10, 10)$. A hybrid approach is also given in Bryce and Colbourn [51] for constructing covering array. The study focussed on covering as many t-tuples as early

Table 6
Variants of approaches used for constructing covering arrays using metaheuristics.

Approach used	Articles
Independent application of metaheuristics	Cohen et al. [46,47], Stardom [45], Nurmela [49], Shiba et al. [50]
Use of an integrated approach	Cohen et al. [48], Bryce et al. [51]

as possible. So rather than minimizing the number of tests to achieve t-way coverage, the initial rate of coverage was the primary concern. The hybrid approach applied a one-test-at-a-time greedy algorithm to initialize tests and then applied heuristic search to increase the number of t-tuples in a test. The heuristic search techniques applied were hill-climbing, SA, TS and great flood. With different inputs, SA in general produced the quickest rate of coverage with 10 or 100 search iterations. The study also concluded that smaller test suites do not relate to greater rate of coverage; hence two different and sometimes inconsistent goals when applied in a real world setting.

As mentioned earlier, there is less evidence on construction of variable strength arrays. One such study used SA to find variable strength arrays and provided initial bounds [47]. In his work using TS, Nurmela improved on previously known upper bounds on the sizes of optimal covering arrays [49]. Experimenting with number of factors and number of values for each factor, good upper bounds were tabulated for strength-two covering arrays. In addition, the study improved upper bounds for strength three covering arrays. The TS algorithm was found to work best for strength two covering array with number of levels for each factor equal to three. According to [49], it was difficult to be sure about the upper bounds to be optimal or not because TS being a stochastic algorithm could improve upon the new bounds if given more computing time.

Toshiaki et al. used genetic algorithms (GA) and ant colony algorithm (ACA) for constructing covering arrays [50]. The results were compared with AETG, in-parameter order (IPO) [62] algorithm and SA. SA outperformed their results of using GA and ACA with respect to the size of resulting test sets for two-way and three-way testing. Their results however outperformed AETG for two-way and three-way testing. It was interesting to find that using GA, the results did not match with those produced by Stardom's study [45] which indicated that GA did not perform well in generating covering arrays, even though several attempts were made to modify the structure of the algorithm.

A summary of results of applying metaheuristics for covering array construction is given in Table 7.

3.5. Safety

Safety testing is an important component of the testing strategy of safety critical systems where the systems are required to meet safety constraints. In terms of metaheuristic search techniques, SA and GA are applied for safety testing.

In Tracey et al. [52], the authors proposed an approach using GAs and SA to generate test data violating a safety property. This approach extended the authors' previous work in developing a general framework for dynamically generating test data. The violation of a safety property meant a hazard or a failure condition, which was initially identified using some form of hazard analysis technique e.g. functional hazard analysis. The fitness function used evaluates different branch predicates and evaluates to zero if the safety property evaluates to false and will be positive otherwise. The search stopped when a test data with a zero cost was found. The cost function calculation is presented in Table 8. where K represents the penalty which was added for undesirable data [52].

The paper provided a simple example where either SA or GAs can be applied to automatically search for test data violating safety properties that must hold 'after' the execution of the SUT. The same given cost function can also be used to generate test data to violate safety conditions at specific points 'during' the execution of the SUT. In this case, the SUT needed to be instrumented such that the branch predicates were replaced by procedures which served two purposes of returning the boolean value of the predicate they replaced and adding to the overall cost the contribution made by each individual branch predicate that was executed. An example was given with an original program and the instrumented program with examples of how the fitness function was able to guide the search.

Table 8
Cost function calculation.

Element	Value
Boolean	`if TRUE then 0 else K`
$a = b$	`if abs(a - b)=0 then 0 else abs(a - b) + K`
$a \neq b$	`if abs(a - b) ≠ 0 then 0 else K`
$a < b$	`if a - b < 0 then 0 else (a - b) + K`
$a \leq b$	`if a - b ≤ 0 then 0 else (a - b) + K`
$a > b$	`if b - a < 0 then 0 else (b - a) + K`
$a \geq b$	`if b - a ≤ 0 then 0 else (b - a) + K`
$a \vee b$	`min(cost(a), cost(b))`
$a \wedge b$	`cost(a) + cost(b)`
$\neg a$	Negation propagated over a

Table 7
Summary of results applying metaheuristics for covering array construction. The last column on the right covers any issues such as constraints, limitations and highlights. (TS is short for Tabu Search, SA is short for Simulated Annealing, HC is short for Hill Climbing, ACA is short for Ant Colony Algorithm and GA is short for Genetic Algorithm.)

Article	Applied metaheuristic	Fitness function used	Limitations and highlights
Stardom [45]	TS, SA and GA	Number of uncovered t-subsets	**GA** takes more time and more moves to find a good covering array. **Larger** parameter sets require greater memory to store information.
Cohen et al. [46]	SA and HC	Number of uncovered t-subsets	**There** is still no best method for building variable strength test suite. **Combining** algebraic constructions with metaheuristic search is promising.
Cohen et al. [47]	SA	Number of uncovered t-subsets	**Variable** strength arrays guarantee a minimum strength of overall coverage and allow varying the strength among disjoint subsets of components.
Nurmela [49]	TS	Number of uncovered t-subsets	**If** more computing time is given, many of the new bounds can be improved slightly.
Cohen et al. [48]	SA	Number of uncovered t-subsets	**A** tool can be designed to take advantage of combining combinatorial construction along with heuristic search.
Bryce et al. [51]	TS, SA and HC	Number of uncovered t-subsets	**SA** provides the fastest rate of t-tuple coverage while TS is slowest.
Toshiaki et al. [50]	ACA, GA	Number of uncovered t-subsets	**The** test sets generated are small but they are not always optimal.

The approach presented by Tracey et al. [52] has been extended by Abdellatif-Kaddour et al. [53] for sequential problems. In Abdellatif-Kaddour et al. [53], SA was used for step-wise construction of test scenarios (progressive exploration of longer test scenarios) to test safety properties in cyclic real-time control systems. The stepwise construction was required due to the sequential behavior of the control systems as it was expected that safety property violation will occur after execution of a particular trajectory or sequence of data in the input domain. The test strategy was applied to a steam boiler case study where the target safety property was the non-explosion of the boiler. Along with the objective of violating a safety property, there was a set of dangerous situations of interest when exploring progressive evolution towards property violation. So the objective was not only violation of a target property but also to reach a dangerous situation. For the steam boiler case study, there could be ten possible safety property violations and three dangerous situations. The solution space in this case was divided into several subsets of smaller sizes. So different classes of test sequences were independently searched. For each class, the objective was defined into sub-objectives corresponding to either safety property violation or the achievement of a dangerous situation. The overall cost was the minimum value of the different sub-objective cost functions. The efficiency of using SA was analyzed in comparison with random sampling. The first experiment showed that random sampling found test sequences that fulfilled main objective more quickly than the approach using SA. Therefore a revised SA was used in which the acceptance probability was adjusted to allow for significant moves in case of no cost improvement. The revised version of SA offered significant improvement over the basic version of SA, while in comparison with random sampling a slight improvement was observed both in terms of total number of iterations and successful search. The results of the study confirmed the usefulness of stepwise construction of test scenarios, but in terms of efficiency of SA algorithm, the cost effectiveness as compared with random sampling remains questionable.

In Baresel et al. [15,54], an evolutionary testing approach using genetic algorithms was presented for structural and functional sequence testing. For complex dynamic system like car control systems, long input sequences were necessary to simulate these systems. At the same time, one of the most important aims was to generate a compact description for the input sequences, containing as few elements as possible but having enough variety to stimulate the system under test as much as necessary. In order to have a compact description of input sequences for a car control system, the long input sequence was divided into sections. Each section had a base signal having signal type, amplitude and length of section as variables. These variables had bounds within which the optimization generated solutions. The output sequences generated by simulating the car control system were to be evaluated against a fitness function, which was defined according to the problem at hand. For example, in case of a car control system, the fitness function checked for violations of signal amplitude boundaries. The applied fitness function consisted of two levels, which differentiated quality between multiple output signals violating the defined boundaries. An objective value of -1 indicated a severe violation while for less severe violations, the closeness of the maximal value to the defined boundary was calculated. The results of the experiment performed on a car control system showed that the optimization continually found better values and ultimately a serious violation was detected.

A summary of results of applying metaheuristics for safety testing is given in Table 9.

4. Discussion and areas for future research

The body of knowledge into the use of metaheuristic search techniques for verifying the temporal correctness is geared towards real-time embedded systems. For these systems, temporal correctness must be verified along with the logical correctness. The fact that there is a lack of support for dynamic testing of real-time system for temporal correctness caused the research community to take advantage of metaheuristic search techniques. It is possible to differentiate the temporal testing research into two dimensions. One of them focuses on violation of timing constraints due to input values and most of the temporal testing research follows this dimension. The other dimension, which is the one taken by Briand et al. [12] analyses task architectures and consider seeding times of events triggering tasks and tasks' synchronization, i.e. Briand's et al. study does not consider tasks in isolation. Both approaches to temporal verification are complementary.

The performance outcome information from studies related to execution time are given in Table 10. It is fairly evident from the table that GA consistently outperforms random and statistical testing in wide variety of situations, producing comparatively longer execution times faster and also finding new bounds on BCET. GAs were also able to perform better than human testers and on occasions where it failed to do so may be attributed to the complexity of the test objects inhibiting evolutionary testability. With respect to comparison with static analysis, GA performed comparably well and both techniques are shown to bound the actual execution times from opposite ends.

For execution time, genetic algorithms are used as the metaheuristic in vast majority of cases (14 out of 15 papers), while SA finds application in one of the studies. The preference of using genetic algorithms over SA can be attributed to the very nature of search mechanism inherent to genetic algorithms. Since a genetic algorithm maintains a population of possible solutions, it has a better chance of locating global optimum as compared to SA which proceed one solution at a time. Also due to the fact that temporal

Table 9
Summary of results applying metaheuristics for safety testing. The last column on the right covers any issues such as constraints, limitations and highlights. (GA is short for Genetic Algorithm and SA is short for Simulated Annealing.)

Article	Applied metaheuristic	Fitness function used	Limitations and highlights
Tracey et al. [52]	GA and SA	Evaluation of different branch predicates with zero cost if the safety property evaluates to false and positive cost otherwise	**Experimentation** on small scale problems, thus results are preliminary. **Instrumentation** of the test objects is a challenge in the scalability of the technique.
Kaddour et al. [53]	SA	Cost related to the violation of the safety property and achievement of dangerous situation	**Using** SA, many trials are necessary for investigating alternative design choices and calibrating the corresponding parameters. **More** experimentation is required to confirm the efficiency applying revised SA algorithm.
Baresel et al. [15], and Pohlheim et al. [54]	GA	Problem specific fitness function measuring different properties e.g. signal amplitude boundaries	**The** input sequences must be long enough and should have the right attributes to stimulate the system. **The** output sequence must be evaluated according to the problem under investigation.

Table 10
Key evaluation information and outcomes of execution time studies. (GA is short for Genetic Algorithm, EGA is short for Extended GA, while SA is short for Simulated Annealing.)

Article and metaheuristic	Method of evaluation	Test objects	Performance factor evaluated	Outcomes of the experiment
Wegener et al. [25], GA	Comparison with statistical and systematic testing	Simple C function	Finding WCET/BCET and number of generations	The longest execution time was found very fast and a new shortest execution time was found, not previously discovered by statistical and systematic testing
Alander et al. [26], GA	Comparison with random testing	Relay software simulator	Finding processing time extremes	GA generated input data with longer response times
Wegener et al. [27], GA	Comparison with statistical and systematic testing	5 programs with up to 1511 LoC and 843 integer input parameters	Finding WCET/BCET and the number of tests required	GA found more extreme times although on occasions required more tests to do so
Wegener and Grochtmann [10], GA	Comparison with random testing	8 programs with up to 1511 LoC and 5000 input parameters	Finding WCET/BCET and the number of tests required	GA always obtained better results as compared with random testing
O'Sullivan et al. [28], GA	Comparison with different termination criteria i.e. limiting the number of generations, time spent on test and examining the fitness evolution	An algorithm from automotive electronics	Convergence analysis of various termination criteria	Cluster analysis turned out to be a more powerful termination criterion which allowed quick location of local optima
Tracey et al. [29], SA	Comparison in terms of size of parameter space to search	4 programs (conditional blocks, simple loop, binary integer square root and insertion sort)	Finding WCET	SA successfully executed a worst case path and was more effective with larger, complex parameter space
Pohlheim and Wegener [32], EGA	Comparison with systematic testing	Modules from a motor control system	Finding maximum execution times	GAs found longer execution times for all the given modules
Mueller and Wegener [30], GA	Comparison with static analysis	5 experiments with industrial and reference applications	Finding WCET/BCET	Use of evolutionary testing and static analysis bounded the actual execution times from opposite ends and were complementary
Puschner and Nossal [31], GA	Comparison with static analysis and random testing	7 programs with diverse execution time characteristics	Finding WCET	For large input data space, GA outperformed random method. In comparison with static analysis, the performance of GA is comparable
Wegener et al. [33], GA	Comparison with execution time found with developers test	An engine control system with 6 time critical tasks	Finding WCET	Longer execution times were found with the evolutionary test than with the developer tests
Groß [36], GA	Random test case generation and performance of an experienced human tester	15 example test programs	Finding WCET	Evolutionary testing generated more worst-case times as compared with random testing. Also for only 4 out of 15 test objects the human tester was more successful
Groß [35], GA	None	21 test objects of varying input size	Evolutionary testability	The prediction system forecasted evolutionary testability with almost 90
Groß et al. [34], GA	None	22 test objects with varying input sizes	Evolutionary testability	Evolutionary testability and complexity of test objects was found to be interrelated
Tlili et al. [11], EGA	Standard evolutionary real-time testing with random initial population and no search space restriction	12 test objects with varying cyclomatic complexity	Finding longer execution times	Using range restriction and seeding initial population with data achieving high structural branch coverage, longer execution times were found for most of the test objects except for two
Briand et al. [12], GA	None	Two case studies, one of researchers own scenarios and the second consisted of an actual real time system	Maximizing critical deadline misses	It was possible to identify seeding times such that small errors in the execution time estimates could lead to missed deadlines

behavior of real-time systems always results in complicated multi-dimensional search space with many plateaus and discontinuities, genetic algorithms are suitable since they perform well for problems involving large number of variables and complex input domains.

There is another way of analyzing the body of knowledge into the use of metaheuristics for testing temporal correctness; which is in terms of (1) properties associated with the metaheuristic itself and (2) properties related to the SUT (test object). In terms of metaheuristic, the research focuses on improving multiple issues: reduction in search space, comparative studies with static analysis, selection schemes for initial population of genetic algorithm, evaluation of suitable termination criterion for search, choice of fitness function, mutation and crossover operators and search for robust and appropriate parameters. In terms of test objects, the research focuses on properties of test objects inhibiting evolutionary testability and formulation of complexity measures for predicting evolutionary testability (Fig. 4).

The fact that seeding the initial population of an evolutionary algorithm with test data having high structural coverage has had better results, it would be interesting to design a fitness function that takes into account structural properties of individuals. Then it will be possible not only to reward individuals on the basis of execution time but also on their ability to execute complex parts of the source code. Similarly, there are different types of structural coverage criteria, which can be used to seed initial populations and

might prove helpful in the design of a fitness function that takes into account such a structural coverage criterion.

In terms of reliable termination criterion for evolutionary testing, use of cluster analysis is found to be useful over other termination criteria, e.g. number of generations and examination of fitness evolution. However, to the authors' knowledge, the use of cluster analysis as a termination criterion is used in only one study [28]. Cluster analysis information can be used to change the search strategy in a way that escapes local optima and helps exploring more feasible areas of the search space. Also the performance of evolutionary algorithms can be made much better by using robust parameters. The search for these parameters is still on. Similarly, variations of existing algorithms e.g. like using extended evolutionary algorithms might give interesting insights into the performance of evolutionary testing.

We also gathered studies regarding application of metaheuristic search techniques for quality of service aware composition and violation of service level agreements (SLAs) between the integrator and the end user. Genetic algorithms have been applied to tackle the QoS-aware composition problem as well as generation of inputs causing SLA violations. Table 11 show that in comparison with linear integer programming and random search, genetic algorithms were more successful in meeting QoS constraints. We also infer that the testing of service-oriented systems has inherently several issues. These include testability problems due to lack of observability of service code and structure, integration testing issues due to the use of late-binding mechanisms, lack of control and involved cost of testing [38]. These issues raises the need for adequate testing strategies and approaches tailored for service-oriented systems. In terms of QoS, different attributes are of interest and are competing e.g. cost, response time, availability and reliability. These attributes need to be computed for workflow constructs. Empirical evidence has to be gathered for a thorough comparison of genetic algorithms with non-linear integer programming for QoS-aware service composition. Also network configurations and server load, being one of the factors causing SLA violations, is to be accounted for generating test data violating the SLA.

Buffer overflow attacks compromises the security of applications. The attacker needs three requirements for a successful exploit, (i) a vulnerable program within the target system (ii) information of the size of memory reference necessary to cause the overflow and (iii) the correct placement of a suitable exploit to make use of the overflow when it occurs [43]. In order to guide the interpretation of findings, performance outcomes are summarized in Table 12, in addition to Table 5. The range of studies offers variations with respect to the main theme, although all have the common goal of addressing security testing. Therefore, we see studies making use of metaheuristic search techniques to create a range of successful attacks to evade common intrusion detection systems as well as to identify buffer overflows. For detecting buffer overflows, the attacker's arbitrary code needs modification to increase the success chances of creating a malicious buffer. In case of hacker scripts generation, the goals for the hacker script generation needs to be defined. The use of metaheuristic search techniques includes grammatical evolution, linear genetic programming, genetic algorithm and particle swarm optimization. Devising a useful fitness function is the focus of majority of the studies which highlights the difficulty in instrumenting security is-

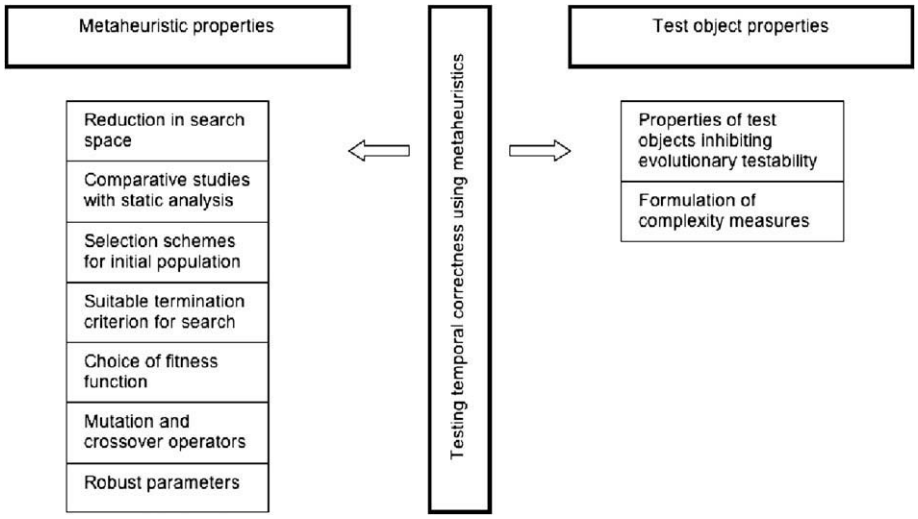

Fig. 4. Two ways to analyze temporal testing research using metaheuristics.

Table 11
Key evaluation information and outcomes of QoS studies. (GA is short for Genetic Algorithm.)

Article and metaheuristic	Method of evaluation	Test objects	Performance factor evaluated	Outcomes of the experiment
Canfora et al. [37], GA	Linear integer programming	A workflow containing 18 invocations of 8 distinct abstract services	Convergence times of integer programming and GA for the same achieved solution	When the number of concrete services available for each abstract service is large, GA should be preferred instead of integer programming. On the other hand, whenever the number of concrete services available is limited, integer programming is to be preferred
Di Penta et al. [38], GA	Random search	Audio processing workflow and a service for chart generation	Violation of QoS constraint and time required to converge to a solution	The new approach outperformed random search and successfully violated QoS constraints

Table 12
Key evaluation information and outcomes of security studies. (GA is short for Genetic Algorithm, GE is short for Grammatical Evolution, while PSO is short for Particle Swarm Optimization.)

Article and metaheuristic	Method of evaluation	Test objects	Performance factor evaluated	Outcomes of the experiment
Dozier et al. [39], GA and PSO	Comparison of steady state GA and six variants of PSO	1998 MIT Lincoln Lab Data	To discover holes (Type II errors/false negatives)	GA outperformed all of the swarms with respect to number of distinct vulnerabilities discovered
Kayacik et al. [40], GE (basic, niching and niching & NoOP minimization)	Detection (or not) of each exploit through the Snort misuse detection system	A simple (generic) vulnerable application performing a data copy without checking the internal buffer size	The number of alerts that Snort generates when attacks are executed	The results from three variants of GE were comparable
Budynek et al. [41], GA	Results from a log analyzer	Automatic generation of computer hacker scripts (a sequence of Unix commands)	Evidence collection from the logs	Various top scoring scripts were obtained
Grosso et al. [42], GA	Comparison of three different fitness functions namely vulnerable coverage fitness, nesting fitness and buffer boundary fitness	A white-noise generator and a function contained in the ftp client	Modified *t*-test to compare fitness values of three fitness functions across the two test objects	A fitness function accounting for the distance from the buffer boundaries outperformed fitness function not using the same factor
Grosso et al. [44], GA	The fitness that does not use dynamic weighing	Two different sets of C applications	A comparison of fitness values of competing fitness cases in terms of number of generations	The new fitness function outperformed the comparable ones
Kayacik et al. [43], Linear GP	Evaluation of a new approach	Three experiments with different data sets	Avoidance of attack detection by Snort, the network based intrusion detection system	The evolved attacks discovered different ways of attaining sub-goals associated with building buffer overflow attacks
Kayacik et al. [23], Linear GP	Compares two fitness functions (incremental and concurrent) in detecting anomaly rate	Instruction set consisting of most frequently occurring system calls	Measurement of alarm rate indicating evasion of Stride, the anomaly host based detection system	Reduction of anomaly rate from $\sim 65\%$ to $\sim 2.7\%$

sues as an appropriate search metric. Most of the fitness functions are based on the ability of the attack to fulfill the conditions necessary for a successful exploit. This has resulted in studies comparing the performances of different fitness functions rather than comparing with traditional approaches to security testing. This indicates that most of the studies into use of metaheuristics for this domain is largely exploratory and no trends are observable that can be generalized. Due to this we see authors experimenting with simple and small applications on a limited scale. The scalability of these approaches, with larger data sets and greater number of trials, is an interesting area of future research. The work of Kayacik et al. [40,43,23] is notable as they move towards a general framework for attack generation based on the evolution of system call sequences. Also the co-evolution of attacker-detector pairs (as pointed out by Kayacik et al. [23]) that provides the opportunity to actually preempt new attack behaviors before they are encountered, formulation of techniques (like static analysis and program slicing) to reduce the search space for the evolutionary algorithm, hybridization of GA and PSO algorithm for effective searching and creation of an interactive tool for vulnerability assessment based on evolutionary computation for the end users provides further future extensions.

In the area of usability, we found the application of metaheuristic search techniques for constructing covering arrays. Being a qualitative attribute, usability possesses different interpretations. However, we classify studies reporting construction of covering arrays under usability as they relate to a form of interaction testing covering *t*-way user interactions whereby each test case exposes different areas of a user interface. The research for construction of covering arrays for software testing have dual focus of finding new techniques to produce smaller covering arrays and to produce them in reasonable amount of time. As expected, a trade-off must be achieved between computational power and size of resulting test suites. The extent of evidence related to applying metaheuristics for finding better bounds on covering arrays suggest that metaheuristics are very successful for smaller parameter sets. For larger parameter sets, the heuristic algorithms run slowly due to the large amount of memory required (to store information). Therefore, execution time is a known barrier in finding more results using metaheuristic search algorithms. One obvious way to achieve efficient memory management is to reuse the previously calculated *t*-combinations by storing them in some form of a temporary memory. Also as mentioned earlier, the size of the test suite is not known a priori, many sizes must be tested for obtaining a good bound on a *t*-way test set of given size.

It is also worth mentioning that finding optimal parameters for effectively using metaheuristics require many trials, moreover it is difficult to be sure whether the upper bounds produced are optimal or not because further improvements on bounds can take place if given more computing time. In addition to Table 7, we summarize key evaluation information and outcomes in Table 13. We gather that a range of metaheuristics have been applied for constructing covering arrays. This includes SA, TS, GA and ant colony optimization. We find SA and TS to be widely applicable search techniques, particularly SA being applied to generate smaller sized test suites. Out of seven primary studies, five report using SA while three use TS, either being used independently or in combination with other search techniques and algebraic constructions. We infer from Table 13, SA consistently performs better than GA, HC, ACA and TS in terms of size of resulting test sets. The performance of SA can further be improved by integrating it with the use of algebraic constructions. Another possibility is to begin with a greedy algorithm (like TCG) and then make a transition to heuristic search after meeting a certain condition [46]. GA has been applied in two studies with contradictory results and hence requires further experimentation. An interesting area is to explore the use of ant colony optimization to generalize initial results given by [50]. In terms of construction of variable strength covering arrays, there is a potential for further research for finding the best approach for variable strength test suite.

Safety testing is used to test safety critical systems that have to satisfy the safety constraints in addition to satisfying the functional

Table 13
Key evaluation information and outcomes of usability studies. (GA is short for Genetic Algorithm, TS is short for Tabu Search, SA is short for Simulated Annealing, HC is short for Hill Climbing, PSO is short for particle swarm optimization while, ACA is short for Ant Colony Algorithm.)

Article and metaheuristic	Method of evaluation	Test objects	Performance factor evaluated	Outcomes of the experiment
Stardom [45], SA, TS and GA	Comparison of SA, TS and GA	Different sizes of covering arrays, CA(13,11:1), CA(9,7:1), CA(7,6:1)	Three tests to find the best arrays possible in the shortest amount of time	GA was ineffective at finding quality arrays when compared with TS and SA. SA in general was found to be very useful for finding covering arrays of various sizes while when the size of an arrays neighborhood was smaller, TS was able to find much better arrays
Cohen et al. [46], SA and HC	Comparison of SA, HC and greedy methods (AETG, TCG)	Different sizes of mixed covering arrays and fixed covering arrays	Number of test cases in a test suite and time required to obtain them	HC and SA improved on bounds given by AETG and TCG, SA consistently performed well or better than HC
Cohen et al. [47], SA	Presentation of results for a new combinatorial object (variable strength covering array)	Minimum, maximum and average sizes of different variable strength covering arrays	Minimum, maximum and average sizes of variable strength covering arrays after 10 runs of SA	Variable strength covering arrays of different sizes
Nurmela [49], TS	Comparison with best known upper bounds	Upper bounds on $g_2(Z_n^q)$ for small q and n	Size of the covering array	The search algorithm worked best for t=2 and q=3
Cohen et al. [48], SA	Comparisons with strength three covering arrays	Several bounds for strength three covering arrays	Size of strength three covering array	Combination of combinatorial construction and SA presented new bounds for some strength 3 covering arrays
Bryce et al. [51], TS, SA, HC	Comparisons among TS, SA and HC	Two inputs with factors having equal number of levels and two inputs having mixed number of levels	Rate of t-tuple coverage	SA had the fastest rate of t-tuple coverage
Toshiaki et al. [50], ACA, GA	Comparisons with AETG, IPO and SA algorithms for the cases $t = 2$ and $t = 3$	Covering arrays and mixed covering arrays of strength 2 and 3	Size of resulting test sets and amount of time required for generation	For t=2, GA and ACA performed comparable to AETG. For t=3, GA and ACA outperformed AETG. SA outperformed GA and ACA with respect to size of the resulting test sets

specification. We take safety in terms of dangerous conditions, which may contribute to an accident. There are two approaches for achieving verification of safety properties: dynamic testing and static analysis. Static analysis does not require execution of the safety critical system while dynamic testing executes the system in a suitable environment with test data generated to test safety properties. Both static analysis and dynamic testing are complementary approaches; in many cases the results of static analysis are used to give criteria for dynamic testing (as in Tracey et al. [52]). The available primary studies discussing testing of safety properties can be differentiated into two themes. One is the case where generation of separate inputs is discussed to test the safety property while the other case discusses generation of sequence of inputs.

The performance outcomes for studies related to safety testing are given in Table 14. The studies show that SA and GAs are applied in the context of safety testing, GA being more successfully applied. However, the results suggest a need for further experimentation in terms of investigating alternative design choices and calibration of algorithmic parameters. This is desirable especially when the extensions to the basic approaches of safety testing can be applied to fault injection, testing for exception conditions and testing for safe component reuse and integration [52]. In terms of alternate design choices, we see in Abdellatif-Kaddour et al. [53] that the efficiency of SA is dependent on the initial solution because when no cost improvement is observed, the search does not allow moves larger than those authorized by the neighborhood function. Therefore, improvement in the efficiency of the SA algorithm can be achieved by searching elsewhere then the neighborhood of the current solution if no cost improvement is made. Further experimentation is also desirable in terms of safety testing real world applications, however, the instrumentation required for testing of safety conditions is a challenge with respect to the scalability of the technique. Also, since the design choices are highly dependent on the nature of the safety critical system, it is consequently important to include more problem specific knowledge into the representation of solutions and design of fitness function.

Table 14
Key evaluation information and outcomes of safety studies. (GA is short for Genetic Algorithm, while SA is short for Simulated Annealing.)

Article and metaheuristic	Method of evaluation	Test objects	Performance factor evaluated	Outcomes of the experiment
Tracey et al. [52], GA and SA	Comparison of safety conditions obtained from software fault tree analysis and functional specification	Small size functions used in the pre-proof step	Finding test data that causes an implementation to violate a safety property	The approach might be useful not only for safety verification but also for integration with fault injection, testing for exception conditions and testing for safe component reuse
Kaddour et al. [53], SA	Effectiveness in terms of finding appropriate test sequences and efficiency in terms of comparing the speed of SA with random sampling	Non-explosion of the steam boiler	Finding the test sequence that lead to either an explosion or a dangerous situation	Both random search and SA were effective while random search was more efficient
Baresel et al. [15], Pohlheim et al. [54], GA	None	Dynamic car control system	Violation of defined requirements for output sequence generated by the simulation of the dynamic system	It was possible to generate real world input sequences causing violations of a given safety requirement

We can infer from this review that search-based testing is poorly represented in the testing of non-functional properties. While search-based software engineering might be transitioning from early optimistic results to more in-depth understanding of the associated problems [2], search-based testing of non-functional properties is still ad-hoc and largely exploratory. The main reasons that can be attributed to this trend are the difficulties associated with instrumenting non-functional properties as fitness functions and also difficulties in generalizing search-based testing of non-functional properties on a broader scale due to strictly domain specific nature of existing studies. With the majority of studies in search-based software testing applied to functional testing, the use of metaheuristic search techniques for testing non-functional properties is rather limited and, with exception to execution time studies, are very problem specific. We believe that it is important, in order to develop the currently emerging field of search-based software testing, to analyze the applicability of search techniques in testing of diverse non-functional properties which can then trigger the second phase of exploration requiring a deeper understanding of problem and solution characteristics [2].

5. Validity threats

There can be different threats to the validity of study results.

Conclusion validity refers to the statistically significant relationship between the treatment and the outcome [63]. One possible threat to conclusion validity is biasness in applying quality assessment and data extraction. In order to minimize this threat, we explicitly define the inclusion and exclusion criteria, which we believe is detailed enough to provide an assessment of how we reached the final set of papers for analysis. With respect to the quality assessment, we wanted to be as inclusive as possible, so we resorted to a binary 'yes' or 'no' scale rather than assigning any scores. We made sure to a large extent *include* instead of *exclude* references, hence making sure not to place, by mistake, any relevant contributions in the 'no' category. To assess the consistency of data extraction, a small sample of primary studies were used to extract data for the second time.

Internal validity refers to a causal relationship between treatment and outcome [63]. One threat to internal validity arises from unpublished research that had undesired outcomes or proprietary literature that is not made available. It is difficult to find such grey literature; however we acknowledge that inclusion of such literature would have contributed in increasing internal validity.

Construct validity is concerned with the relationship between the theory and application [63]. One possible threat to construct validity is exclusion of relevant studies. In order to minimize this threat, we defined a rigorous search strategy (Section 2.2), which included two phases, to ultimately protect us against threats to construct validity.

External validity is concerned with the generalization of results outside the scope of the study [63]. We can relate it to the degree to which the primary studies are representative of the overall goal of the review. We believe that our review protocol helped us achieve a representative set of studies to a greater extent. During the course of scanning references, the authors also came across two studies by Schultz et al. [64,65], applying evolutionary algorithms for the robustness testing of autonomous vehicle controllers. We do not include these two studies in our analysis since these studies were published in 1992 and 1995, which are outside the time span (1996–2007) of this review. Furthermore, we did not anticipate finding other relevant studies outside the time span of 1996–2007 as previous relevant surveys supports such a choice of time span.

6. Conclusions

This systematic review investigated the use of metaheuristic search techniques for testing non-functional properties of the SUT. The 35 primary studies are distributed among execution time (15 papers), quality of service (2 papers), safety (4 papers), security (7 papers) and usability (7 papers). While scanning references, we also found two papers relating to robustness testing of autonomous vehicle controllers [64,65] but we do not include these two papers in our review as they were outside the time span of our search (1996–2007).

Within execution time testing, genetic algorithms finds application in 14 out of 15 papers. The research trend within execution time testing is more towards violation of timing constraints due to input values; however, the paper by Briand et al. [12] provides another research approach that analyzes the task architectures and consider seeding times of events triggering tasks and tasks' synchronization. In terms of use of fitness function, we find three variations; the execution time measured in CPU clock cycles, coverage of code annotations inserted along shortest and longest algorithmic execution paths and the fitness function based on the difference between execution's deadline and execution's actual completion. The challenges identified include dealing with potential probe effects due to instrumentation and uncertainty about global optimum, finding appropriate and robust search parameters and a having a suitable termination criteria of search.

Within Quality of Service (QoS), the two papers apply genetic algorithms to determine the set of service concretizations that lead to QoS constraint satisfaction and to generate combinations of bindings and inputs causing violations of service level agreements. One of the papers uses a fitness function based on the maximization of desired QoS attributes while minimizing others and includes the possibility of having static or dynamic fitness function. The other paper uses a fitness function that combines distance-based fitness with a fitness guiding the coverage of target statements. The challenges include the need to compute QoS attributes of component services for workflow constructs and to deal with the possibility of deviation of fitness calculation due to workflow instrumentation.

In security testing; genetic algorithms, linear genetic programming, grammatical evolution and particle swarm optimization have been applied. The applied fitness functions used different representations for the completion of conditions leading to successful exploits. Modifications to the attacker's arbitrary code and finding appropriate goals for hacker script generation are identified as the challenges for security testing.

Within usability testing, metaheuristic search techniques are applied to find better bounds on covering arrays. A variety of metaheuristic search techniques are applied including SA, TS, GA and ant colony algorithms. The fitness function used is the number of uncovered t-subsets. Execution time is a major challenge in this case as for large parameter sets, the metaheuristic algorithms run slowly due to the large amount of memory required to store information.

In safety testing, we find two research directions to test safety properties of the SUT. One makes use of generation of separate inputs to test the safety property, while the other uses a sequence of inputs. Simulated annealing and genetic algorithms are the used metaheuristics and the fitness function takes into account the violation of various safety properties. The incorporation of problem specific knowledge into the representation of solution and design of fitness function presents a challenge for the application of metaheuristic search techniques to test safety properties.

We believe that there is still plenty of potential for automating non-functional testing using search-based techniques and we expect that studies involving NFSBST will increase in the following

Table 15
Study quality assessment.

	Execution time												QoS		Security				Usability						Safety			
	[25]	[26]	[27]	[10]	[28]	[29]	[30]	[31]	[32]	[33]	[34]	[35]	[36]	[12]	[11]	[37]	[38]	[39]	[40]	[41]	[42]	[44]	[43]	[23]	[45]	[46]	[47]	[49]	[48]	[51]	[50]	[52]	[53]	[15]	[54]
A	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
B	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓
D	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
F	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
G	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
H	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
I	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
J	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

A: Is the reader able to understand the aims of the research?
B: Is the context of study clearly stated?
C: Was there a comparison or control group?
D: Are the measures used in the study fully defined [5]?
E: Is there an adequate description of the data collection methods?
F: Does the data collection methods relate to the aims of the research?
G: Is there a description of the method used to analyze data?
H: Are the findings clearly stated and relate to the aims of research?
I: Is the research useful for software industry and research community?
J: Do the conclusions relate to the aim and purpose of research defined?

years. The results of our systematic review also indicate that the current body of knowledge concerning search-based software testing does not report studies on many of the other non-functional properties. On the other hand, there is a need to extend the early optimistic results of applying NFSBST to larger real world systems, thus moving towards a generalization of results.

Acknowledgement

The authors would like to thank Barbara Ann Kitchenham and the anonymous referees for their comments on the earlier drafts of the paper.

Appendix A. Search strings and study quality assessment

- *Abstracts:* (evolutionary OR heuristic OR search-based OR metaheuristic OR optimization OR hill-climbing OR simulated annealing OR tabu search OR genetic algorithms OR genetic programming) AND (“software testing” OR “testing software” OR “test data generation” OR “automated testing” OR “automatic testing”) AND (non-functional OR safety OR robustness OR stress OR security OR usability OR integrity OR efficiency OR reliability OR maintainability OR testability OR flexibility OR reusability OR portability OR interoperability OR performance OR availability OR scalability).
- *Titles:* (evolutionary OR heuristic OR search-based OR metaheuristic OR optimization OR hill-climbing OR simulated annealing OR tabu search OR genetic algorithms OR genetic programming) AND (testing) AND (non-functional OR safety OR robustness OR stress OR security OR usability OR integrity OR efficiency OR reliability OR maintainability OR testability OR flexibility OR reusability OR portability OR interoperability OR performance OR availability OR scalability).
- *Keywords:* (evolutionary OR heuristic OR search-based OR metaheuristic OR optimization OR hill-climbing OR simulated annealing OR tabu search OR genetic algorithms OR genetic programming) AND (“software testing” OR “testing software” OR “test data generation” OR “automated testing” OR “automatic testing”) AND (non-functional OR safety OR robustness OR stress OR security OR usability OR integrity OR efficiency OR reliability OR maintainability OR testability OR flexibility OR reusability OR portability OR interoperability OR performance OR availability OR scalability).

References

[1] M. Harman, B.F. Jones, Search-based software engineering, Information Software Technology 43 (14) (2001) 833–839.
[2] M. Harman, The current state and future of search based software engineering, in: FOSE'07: 2007 Future of Software Engineering, IEEE Computer Society, Washington, DC, USA, 2007, pp. 342–357.
[3] P. McMinn, Search-based software test data generation: a survey, Software Testing, Verification and Reliability 14 (2) (2004) 105–156.
[4] T. Mantere, J.T. Alander, Evolutionary software engineering, a review, Applied Soft Computing 5 (2005) 315–331.
[5] B.A. Kitchenham, Guidelines for performing systematic literature reviews in software engineering, Tech. Rep., EBSE-2007-001, UK (July 2007). URL <http://www.dur.ac.uk/ebse/>.
[6] ISO, International standard iso/iec 9126, information technology—product quality – Part 1: Quality model, Tech. Rep., International Standard Organization, 2001.
[7] N.E. Fenton, S.L. Pfleeger, Software metrics: A rigorous and practical approach, revised, second ed., Course Technology, Boston, MA, USA, 1998.
[8] IEEE, IEEE recommended practice for software requirements specifications, Tech. Rep. 830–1998, Institute of Electrical and Electronics Engineers, Inc., 1998.
[9] D.G. Firesmith, Common concepts underlying safety, security, and survivability engineering, Tech. Rep., CMU/SEI-2003-TN-033, Software Engineering Institute, Pittsburgh, PA, December 2003.

[10] J. Wegener, M. Grochtmann, Verifying timing constraints of real-time systems by means of evolutionary testing, Real-Time Systems 15 (3) (1998) 275–298.
[11] M. Tlili, S. Wappler, H. Sthamer, Improving evolutionary real-time testing, in: GECCO'06: Proceedings of the Eighth Annual Conference on Genetic and Evolutionary Computation, ACM Press, 2006, pp. 1917–1924.
[12] L.C. Briand, Y. Labiche, M. Shousha, Stress testing real-time systems with genetic algorithms, in: GECCO'05: Proceedings of the Seventh Annual Conference on Genetic and Evolutionary Computation, ACM Press, 2005, pp. 1021–1028.
[13] B.A. Kitchenham, E. Mendes, G.H. Travassos, Cross versus within-company cost estimation studies: A systematic review, IEEE Transactions on Software Engineering 33 (5) (2007) 316–329.
[14] E.K. Burke, G. Kendall (Eds.), Introductory tutorials in optimisation, decision support and search methodology, Springer, 2005.
[15] A. Baresel, H. Pohlheim, S. Sadeghipour, Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms, in: Genetic and Evolutionary Computation – GECCO 2003, Lecture Notes in Computer Science, vol. 2724, Springer, 2003, pp. 2428–2441.
[16] Y. Zhan, J.A. Clark, Search based automatic test-data generation at an architectural level, in: Genetic and Evolutionary Computation – GECCO 2004, Lecture Notes in Computer Science, vol. 3103, Springer, 2004, pp. 1413–1424.
[17] Y.S. Dai, M. Xie, K.L. Poh, B. Yang, Optimal testing-resource allocation with genetic algorithm for modular software systems, Journal of Systems and Software 66 (1) (2003) 47–55.
[18] K. Derderian, R.M. Hierons, M. Harman, Q. Guo, Input sequence generation for testing of communicating finite state machines (CFSMS), in: GECCO'04: Proceedings of the Sixth Annual Conference on Genetic and Evolutionary Computation, Lecture Notes in Computer Science, vol. 3103, Springer, 2004, pp. 1429–1430.
[19] E. Alba, F. Chicano, Finding safety errors with aco, in: GECCO'07: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, ACM, New York, NY, USA, 2007, pp. 1066–1073.
[20] J. Koljonen, M. Mannila, M. Wanne, Testing the performance of a 2D nearest point algorithm with genetic algorithm generated gaussian distributions, Expert Systems with Applications: An International Journal 32 (3) (2007) 879–889.
[21] K.R. Walcott, M.L. Soffa, G.M. Kapfhammer, R.S. Roos, Timeaware test suite prioritization, in: ISSTA'06: Proceedings of the 2006 International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, 2006, pp. 1–12.
[22] S. Bouktif, H. Sahraoui, G. Antoniol, Simulated annealing for improving software quality prediction, in: GECCO'06: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, ACM, New York, NY, USA, 2006, pp. 1893–1900.
[23] H.G. Kayacik, A.N. Zincir-Heywood, M. Heywood, Automatically evading IDS using GP authored attacks, in: CISDA'07: Proceedings of the IEEE Symposium on Computational Intelligence in Security and Defense Applications, IEE Computer Society, New York, NY, USA, 2007, pp. 153–160.
[24] T. Dybå, T. Dingsøyr, G.K. Hanssen, Applying systematic reviews to diverse study types: An experience report, in: ESEM 2007: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, 2007, pp. 225–234.
[25] J. Wegener, K. Grimm, M. Grochtmann, H. Sthamer, B. Jones, Systematic testing of real-time systems, in: EuroSTAR'96: Proceedings of the Fourth International Conference on Software Testing Analysis and Review, 1996.
[26] J.T. Alander, T. Mantere, G. Moghadampour, J. Matila, Searching protection relay response time extremes using genetic algorithm-software quality by optimization, in: APSCOM'97: Fourth International Conference on Advances in Power System Control, Operation and Management, vol. 1, 1997, pp. 95–99.
[27] J. Wegener, H. Sthamer, B.F. Jones, D.E. Eyres, Testing real-time systems using genetic algorithms, Software Quality Control 6 (2) (1997) 127–135.
[28] M. O'Sullivan, S. Vössner, J. Wegener, Testing temporal correctness of real-time systems, in: EuroSTAR'98: Proceedings of the Sixth International Conference on Software Testing Analysis and Review, 1998.
[29] N. Tracey, J. Clark, K. Mander, The way forward for unifying dynamic test case generation: The optimisation-based approach, in: International Workshop on Dependable Computing and Its Applications (DCIA), IFIP, 1998, pp. 169–180.
[30] F. Mueller, J. Wegener, A comparison of static analysis and evolutionary testing for the verification of timing constraints, Real-Time Systems 21 (3) (1998) 241–268.
[31] P. Puschner, R. Nossal, Testing the results of static worst-case execution-time analysis, in: RTSS'98: Proceedings of the IEEE Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, 1998, p. 134.
[32] H. Pohlheim, J. Wegener, Testing the temporal behavior of real-time software modules using extended evolutionary algorithms, in: W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, R.E. Smith (Eds.), Genetic and Evolutionary Computation – GECCO 1999, vol. 2, Orlando, FL, USA, 1999, p. 1795.
[33] J. Wegener, P. Pitschinetz, H. Sthamer, Automated testing of real-time tasks, in: WAPATV'00: The First International Workshop on Automated Program Analysis, Testing and Verification, Limerick, Ireland, 2000.
[34] H.-G. Groß, B.F. Jones, D.E. Eyres, Structural performance measure of evolutionary testing applied to worst-case timing of real-time systems, IEE Proceedings – Software 147 (2) (2000) 25–30.
[35] H.-G. Groß, A prediction system for dynamic optimization-based execution time analysis, in: SEMINAL 01: Proceedings of the First International Workshop on Software Engineering using Metaheuristic Innovative Algorithms, Toronto, Canada, 2001.
[36] H.-G. Groß, An evaluation of dynamic, optimisation-based worst-case execution time analysis, in: ITPC'03: Proceedings of the International Conference on Information Technology: Prospects and Challenges in the 21st Century, Kathmandu, Nepal, 2003.
[37] G. Canfora, M. Di Penta, R. Esposito, M.L. Villani, An approach for QoS-aware service composition based on genetic algorithms, in: GECCO'05: Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, ACM, 2005, pp. 1069–1075.
[38] M. Di Penta, G. Canfora, G. Esposito, V. Mazza, M. Bruno, Search-based testing of service level agreements, in: GECCO'07: Proceedings of the Nineth Annual Conference on Genetic and Evolutionary Computation, ACM Press, 2007, pp. 1090–1097.
[39] G.V. Dozier, D. Brown, J. Hurley, K. Cain, Vulnerability analysis of immunity-based intrusion detection systems using evolutionary hackers, GECCO'04: Proceedings of the Sixth Annual Conference on Genetic and Evolutionary Computation, vol. 3102, Springer, 2004, pp. 263–274.
[40] H.G. Kayacik, A.N. Zincir-Heywood, M. Heywood, Evolving successful stack overflow attacks for vulnerability testing, in: ACSAC'05: Proceedings of the 21st Annual Computer Security Applications Conference, IEEE Computer Society, Washington, DC, USA, 2005, pp. 225–234.
[41] J. Budynek, E. Bonabeau, B. Shargel, Evolving computer intrusion scripts for vulnerability assessment and log analysis, in: GECCO'05: Proceedings of the Seventh Annual Conference on Genetic and Evolutionary Computation, ACM, New York, NY, USA, 2005, pp. 1905–1912.
[42] C.D. Grosso, G. Antoniol, M.D. Penta, P. Galinier, E. Merlo, Improving network applications security: A new heuristic to generate stress testing data, in: GECCO'05: Proceedings of the Seventh Annual Conference on Genetic and Evolutionary Computation, ACM, New York, NY, USA, 2005, pp. 1037–1043.
[43] H.G. Kayacik, M. Heywood, A.N. Zincir-Heywood, On evolving buffer overflow attacks using genetic programming, in: GECCO '06: Proceedings of the Eighth Annual Conference on Genetic and Evolutionary Computation, ACM, New York, NY, USA, 2006, pp. 1667–1674.
[44] C. Del Grosso, G. Antoniol, E. Merlo, P. Galinier, Detecting buffer overflow via automatic test input data generation, computers and operations research (COR) focused issue on search based software engineering 35 (10), 3125–3143.
[45] J. Stardom, Metaheuristics and the search for covering and packing arrays, Master's Thesis, Simon Fraser University, B.C., Canada.
[46] M.B. Cohen, P.B. Gibbons, W.B. Mugridge, C.J. Colbourn, Constructing test suites for interaction testing, in: ICSE '03: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society, 2003, pp. 38–48.
[47] M.B. Cohen, P.B. Gibbons, W.B. Mugridge, C.J. Colbourn, J.S. Collofello, Variable strength interaction testing of components, in: COMPSAC'03: Proceedings of the 27th Annual International Conference on Computer Software and Applications, IEEE Computer Society, 2003, p. 413.
[48] C.J.C.M.B. Cohen, A.C.H. Ling, Constructing strength three covering arrays with augmented annealing, in: ICSE'03: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society, 2003, pp. 38–48.
[49] K.J. Nurmela, Upper bounds for covering arrays by tabu search, Discrete Applied Mathematics 138 (1-2) (2004) 143–152.
[50] T. Shiba, T. Tsuchiya, T. Kikuno, Using artificial life techniques to generate test cases for combinatorial testing, in: COMPSAC'04: Proceedings of the 28th Annual International Computer Software and Applications Conference, IEEE Computer Society, Washington, DC, USA, 2004, pp. 72–77.
[51] R.C. Bryce, C.J. Colbourn, One-test-at-a-time heuristic search for interaction test suites, in: GECCO'07: Proceedings of the Nineth Annual Conference on Genetic and Evolutionary Computation, ACM, New York, NY, USA, 2007, pp. 1082–1089.
[52] N. Tracey, J. Clark, J. McDermid, K. Mander, Integrating safety analysis with automatic test data generation for software safety verification, in: Proceedings of 17th International System Safety Conference, System Safety Society, 1999, pp. 128–137.
[53] O. Abdellatif-Kaddour, P. Thévenod-Fosse, H. Waeselynck, Property-oriented testing based on simulated annealing. URL <http://www.laas.fr/francois/SVF/seminaires/inputs/02/olfapaper.pdf>.
[54] H. Pohlheim, M. Conrad, A. Griep, Evolutionary safety testing of embedded control software by automatically generating compact test data sequences, in: SAE 2005 World Congress and Exhibition, 2005.
[55] W. Afzal, R. Torkar, R. Feldt, A systematic mapping study on non-functional search-based software testing, in: 20th International Conference on Software Engineering and Knowledge Engineering (SEKE), 2008.
[56] H.G. Groß, B. Jones, D. Eyres, Evolutionary algorithms for the verification of execution time bounds for real-time software, in: IEE Colloquium on Applicable Modelling, Verification and Analysis Techniques for Real-Time Systems, 1999, pp. 8/1–8/8.
[57] L.C. Briand, Y. Labiche, M. Shousha, Using genetic algorithms for early schedulability analysis and stress testing in real-time systems, Genetic Programming and Evolvable Machines 7 (2) (2006) 145–170.
[58] R.C. Bryce, Automatic generation of high coverage usability tests, in: CHI'05: Extended Abstracts on Human Factors in Computing Systems, ACM, New York, NY, USA, 2005, pp. 1108–1109.

[59] D. Hoskins, R.C. Turban, C.J. Colbourn, Experimental designs in software engineering: d-optimal designs and covering arrays, in: WISER'04: Proceedings of the 2004 ACM workshop on Interdisciplinary Software Engineering Research, ACM, New York, NY, USA, 2004, pp. 55–66.
[60] Y.-W. Tung, W. Aldiwan, Automating test case generation for the new generation mission software system, Aerospace Conference Proceedings, vol. 1, IEEE 1 (2000) 431–437.
[61] D.M. Cohen, S.R. Dalal, M.L. Fredman, G.C. Patton, The AETG system: An approach to testing based on combinatorial design, IEEE Transactions Software Engineering 23 (7) (1997) 437–444.
[62] Y. Lei, K.-C. Tai, In-parameter-order: A test generation strategy for pairwise testing, in: HASE'98: The Third IEEE International Symposium on High-Assurance Systems Engineering, IEEE Computer Society, Washington, DC, USA, 1998, pp. 254–261.
[63] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, Norwell, MA, USA, 2000.
[64] A.C. Schultz, J.J. Grefenstette, K.A.D. Jong, Adaptive testing of controllers for autonomous vehicles, in: AUV'92: Proceedings of the 1992 Symposium on Autonomous Underwater Vehicle Technology, 1992, pp. 158–164.
[65] A.C. Schultz, J.J. Grefenstette, K.A.D. Jong, Learning to Break Things: Adaptive Testing of Intelligent Controllers, IOP Publishing Ltd., Oxford Press, 1997 (Chapter G3.5).