On the Evaluation of Automatic Program Repair Techniques and Tools

Alireza Khalilian
Dept. of Software Engineering
University of Isfahan
Isfahan, Iran
khalilian@eng.ui.ac.ir

Ahmad Baraani-Dastjerdi
Dept. of Software Engineering
University of Isfahan
Isfahan, Iran
ahmadb@eng.ui.ac.ir

Bahman Zamani
Dept. of Software Engineering
University of Isfahan
Isfahan, Iran
zamani@eng.ui.ac.ir

Abstract— **Since 2009 several scalable and promising techniques to automatic program repair have been proposed with each technique often accompanied with a prototype tool. These techniques work in different levels of code with various types of defects and designed for different programming languages. Now the subfield of automatic program repair is mature enough to merit evaluate existing techniques and tools. This evaluation helps us identify the strengths and weaknesses of current research and provides future direction. To this end, in this paper, we present a family of criteria grouped into seven sets for evaluating automatic program repair techniques and tools. Moreover, a five-level maturity model is proposed for the mentioned subfield. To the best of our knowledge, no research yet evaluates automatic program repair techniques and tools in a general, broad, and comprehensive manner and this is the first attempt towards this goal. We employed our criteria to three existing mutation-based techniques and their corresponding tools and reported the results of preliminary evaluation. The proof-of-concept results demonstrate different aspects, capabilities and shortcomings of each technique and provide evidence to the applicability and utility of our criteria.**

Keywords- program repair; patch; fault localization; fix localization; criteria

I. INTRODUCTION

Software is still released with known and unknown bugs [43] due to several reasons: limitations in time and resources of testing and debugging, manual effort on fixing bugs, which makes it expensive and time consuming, and higher rate of faults detected by testing as compared to those that are fixed [25]. Post-release fixing of faults is even more expensive and adversely affects the credit of the company in the competitive market [26]. Moreover, it may compromise the critical and private user data. The results of studying the data from 139 software companies in North America [5] represented that each one has spend about 22 million dollars per year to address software defects. Moreover, the results of investigations [46] delineate that among 2000 manual bug fixing, about 14 to 24 percent were incorrect and 43 percent of them resulted in software crash, corruption, security concerns, and hang in system's functionality.

In response to these limitations, the research community started developing automatic techniques [3, 8, 11, 13, 26, 31] to program repair in 2009 based on the proposal of Arcuri et al. [4]. Since then several scalable and promising techniques have been proposed that work in different levels of code [25, 37] with various types of defects [25, 19] and designed for different programming languages [8, 22, 26]. Often each technique has come with a prototype tool to show the applicability and utility of the technique and to provide a way to evaluate it. Now the subfield of automatic program repair is mature enough to merit evaluation of existing techniques and their corresponding tools to identify the benefits and shortcomings of them.

In ICSE'14 Monperrus reviewed [49] the PAR technique [22] and proposed three criteria to evaluate automatic program repair techniques. Then, in ISSTA'15, Qi et al. [50] evaluated the patches of three automatic program repair techniques using a few criteria. However, these publications used either small number of criteria which limits comprehensive evaluation and comparison, or the criteria are targeted towards evaluating the results of some few specific techniques which limits generalization. Moreover, comparing two given techniques implicates that they must share many common features to make the comparisons consistent and meaningful in terms of theoretical and empirical aspects [51]. In their evaluations, Smith et al. [51] selected GenProg [25] and TrpAutoRepair [52] for comparison because they share sufficient common characteristics. Trivially, this important requirement does not often hold for diverse set of existing techniques. A similar concern lies in the fact that many of repair techniques are actually randomized algorithms [53]. Evaluating and comparison of such algorithms are non-trivial. Then, how can we evaluate and compare existing repair techniques?

In order to address these issues and to formalize evaluation of current techniques and tools we present in this paper, a family of 30 (mostly qualitative) criteria with total of 62 first-level sub-criteria and total of five second-level sub-criteria. The criteria are grouped into *seven* sets of interrelated criteria. Most of these criteria can be measured *objectively*. For few of those criteria that need subjective measurements, we present guidelines to mitigate subjectivity. This is because the criteria are intended to yield the same results of investigation by every user. Moreover, the criteria are intended to consider characteristics of each technique and its tool associated with architectural properties, internal constructs, design decisions, constituting components, optimization aspects, and the like. Hence, the criteria can be measured with precise understanding of the technique itself. In case where standard common benchmarks are present, one can compare techniques in terms of performance metrics. However, benchmarks come for a specific language such as C [27] and other techniques that are designed for languages such as Java [3] or Eiffel [41] fail to

benefit from such benchmarks. As a result, a comparison of techniques in terms of performance and non-functional properties are unlikely to succeed. Our criteria, however, aims to overcome this issue by providing a way for evaluating techniques themselves and for comparison with other techniques without reliance on benchmarks and independent of implementation and repair scenario. This also justifies why some of them tend to be more general rather than specific.

We categorized our criteria into two broad classes: *intrinsic* and *extrinsic*. Most of the criteria belong to the first class and few of them belong to the second one. For each criterion, some possible values are enumerated which helps identify the future values of each criterion for newer techniques. In addition, potential benefits and/or drawbacks behind satisfaction/dissatisfaction of each criterion are presented that helps incorporate useful features in the design of new techniques. We also propose a five-level *maturity model* for the automatic program repair techniques. This model facilitates determining the status of an automatic program repair technique in the literature. To the best of our knowledge, no research yet evaluates automatic program repair techniques and tools in a general, broad, and comprehensive manner and this is the first attempt towards this goal. Overall, the criteria are designed to help investigate the current body of knowledge in the context of automatic program repair from different viewpoints. The results of such investigation provide insightful guidelines for future research to build sophisticated techniques that are more scalable and have potential to be used in industrial practice. In order to evaluate the criteria, we applied them to three existing mutation-based techniques. The results serve us as a proof-of-concept and demonstrate the effectiveness of the proposed criteria. Applying the criteria on the other techniques and an extensive and thorough evaluation remain as future work.

The main contributions of this paper are as follows:

- A family of criteria and sub-criteria that is specific to evaluate automatic program repair techniques and tools. The criteria are grouped into seven sets of interrelated criteria and each criterion belongs to either of two classes: intrinsic or extrinsic.
- Some values are enumerated for each criterion or sub-criterion according to the existing techniques and tools. In addition, potential benefits/drawbacks of each criterion are explained where possible.
- A five-level maturity model for automatic program repair techniques and tools.
- The results of studies on evaluating three existing techniques for automatic program repair using the proposed criteria.

The remaining of this paper is organized as follows: Section II briefly reviews automatic program repair concepts. We present the motivation and our criteria to evaluate program repair techniques in Section III. Section IV reports the results of employing criteria to some of existing methods. Discussion and limitations are given in Section V. Related work is discussed in Section VI. Finally, conclusions are mentioned in Section VII.

II. AUTOMATIC PROGRAM REPAIR

There exist two approaches to make correct programs [17, 26]: one class includes techniques that are applied in software development process such as *rigorous software engineering* [45]. This class cannot be applied to existing legacy software. The second class consists of techniques that can be applied to the software itself such as verification, testing, and debugging. Verification techniques provably check the correctness of programs. However, they need formal specifications which are rare in practice in addition to scalability issues to make them applicable in industrial practice [26]. Currently, software is tested against a *representative* set of test cases to detect faults. The faults are then localized to find the root cause or suspicious regions of the code. Finally the bug will be fixed. However, bug-fixing is still a manual effort which is time-consuming and may introduce new faults [25, 28]. Besides, the rate of fault detection by testing is higher than the rate with which faults are fixed. This motivates the research community to seek for automatic techniques to program repair.

Empirical investigations [10] showed that programmers do not produce programs at random and the buggy program is usually close to the correct state. Therefore, we can favor this property to find correct programs by few number and small modifications. A sequence of modifications to the buggy program to correct it is usually called a *patch*. The number of possible modifications over buggy program in infinite despite the fact that required modifications is mostly small and insignificant [3]. Hence, automatic program repair techniques need to employ ways to substantially reduce this space. One way is to map automatic program repair problem to a search problem such as those in search-based software engineering (SBSE) [14].

In order to establish automatic support on program repair, two approaches exist. One class of approaches is meant to generate comments or recommendations that serve as debugging assistance for developers [18, 21]. The second class actually fixes the bug by modifications on the code [25]. This second class consists of two broad categories of automatic program repair techniques [27], namely *correct-by-construction* and *generate-and-validate*. The former produces one or few number of correct programs using sound techniques [19, 31]. The latter generates multiple candidate patches that are then evaluated by heuristic methods and the most appropriate one is returned [29]. The input to an automatic program repair technique is a faulty program in source, binary, or assembly code along with required behavior and an evidence of the fault [28]. The latter pieces of information are typically provided via test cases or some kind of specifications. The automatic program repair technique often fixes the bug by generating a patch, runtime modifications, or a dynamic jump to the new code [26]. Note that current techniques to automatic program repair are roughly unable to replace human developer [25] due to many reasons. This is especially the case for test case-based techniques because they cannot consider any aspect or design goal beyond what can be manifested by test cases. Note also that absolute “automation” is by no means the case

for existing techniques due to oracle generation problem, the need to initialize and dispatch and other reasons [25, 28].

III. MOTIVATION AND THE PROPOSED CRITERIA

As the number of programming languages (PL) [38] and model transformation languages [1] (MTL) [7] increases, software developers need to leverage various evaluation criteria [24, 38] to identify their strengths and weaknesses. This would help them to decide on an appropriate language, among diverse set of languages, for the task at hand. The results of evaluation would also provide helpful insights for the design of more effective and efficient languages in future. Similarly, the research community has developed promising techniques and tools in recent years for the subfield of automatic program repair [3, 22, 26, 37]. Introducing these techniques led to an extent of maturity that merits to be considered as a *level* of maturity. At this stage, evaluation of existing techniques can expose the potentials and shortcomings of each technique in isolation. It can also provide a way to compare and contrast different techniques and tools.

Evaluation and comparison of current repair techniques face several impediments:

- We need to standard common benchmarks whose absence make comparison nonsensical. However, the current research community faces substantial limitations in this line of work.
- The only available systematically-constructed benchmark we are aware of, much recently created by Le Goues et al. [27], consists of C programs. Hence, other techniques designed for Java [3], Eiffel [41], or other languages fail to use it.
- Even in presence of standard benchmarks and in case where the target language of subject techniques are the same, the implementation of a certain technique may not be available, or the detailed settings of parameters used in the tool may not be exactly presented [27].
- Comparison of two given techniques implicate that they share many common characteristics to make consistent and meaningful results from both theoretical and empirical aspects. This requirement does not hold for diverse set of current techniques as they establish different approaches.
- Many of current repair techniques are based on randomized algorithms. Evaluating such algorithms is non-trivial since many issues must be carefully taken into account [51]: sample size, statistical tests, cross-validation, and bootstrapping.

This paper presents *domain-specific* evaluation criteria to be used in the subfield of automatic program repair. Putting a new technique for automatic program repair in these criteria allows for a fast yet accurate assessment of the technique itself. It also provides a framework to help compare a certain technique with other existing techniques. To the best of our knowledge, this is the first report to propose evaluating of automatic program repair techniques using a general, comprehensive, and mostly objective family of criteria dedicated for this task. The novelty of our work lies in the fact that in addition to present a broad family of criteria, we also present some of the possible values for each criterion along with the relevant characteristics. This helps extend the possible values by appearance of new techniques accordingly. Moreover, the potential benefits and/or drawbacks behind satisfaction or dissatisfaction of each criterion are also given to guide future research for developing more sophisticated and effective techniques.

We grouped our criteria in seven sets of interrelated criteria which are shown in Tables I to III. Except for the first group which contains independent criteria, each other group consists of the sub-criteria related to a major criterion. We propose to divide the criteria into two broad classes: *intrinsic* and *extrinsic*. Intrinsic criteria cover technical aspects that root in the technique itself in terms of design, development, and evaluation. By contrast, extrinsic criteria deal with environmental aspects and are influenced by external factors. Most of our criteria are intrinsic and few of them are extrinsic.

We have *included* those criteria that target automatic program repair techniques and tools, that are general enough to evaluate them independent of some (possibly unclear) details and without reliance on running the tools, that can be perceived and measured by understanding the technique and evaluations themselves, and that can be measured objectively as much as possible. We have also *excluded* each criterion that needs any form of execution of the tool. Since each technique and its corresponding tool map to each other, it is reasonable to consider them together when evaluation.

In addition to the criteria, we propose a *maturity model* as a separate criterion with five levels. This model takes as input an automatic program repair technique and places it in the fittest level. It would establish a metric to facilitate identifying where a certain technique fits in the increasing body of the automatic program repair literature.

In the remaining of this section, we give the mentioned criteria. Each criterion may have multiple levels (at this time only two levels) of sub-criteria. For each criterion or sub-criterion, some possible cases are also given. The number of criteria and their possible values can be further extended as the subfield grows. The criteria are shown in Table I to Table III. Upper part of Table I shows single intrinsic criteria; i.e. those with no sub-criteria. The lower part of Table I and other tables show the criteria that have first-level or second-level sub-criteria. Note that extrinsic criteria start with "E". We then give explanations on criteria starting from Table I to Table III sequentially.

The general approach: Some of the successful approaches work based on evolutionary computations (EC) such as genetic programming (GP) meta-heuristic [36]. The others work based on behavioral models [8] and so on. The particular approach of each category should be explicitly stated.

[1] MTLs are used in the context of model-driven software engineering (MDSE) [5]

Repair approach: Correct-by-construction approaches produce single or few repairs that are provably sound [27]. They work based on some formulation of program such as formal specifications. Generate-and-validate approaches produce multiple repairs heuristically and test them against a fitness function [42].

Type of algorithm: An algorithm with at least one randomized input or step is considered as stochastic algorithm such as GP [36]. By contrast, deterministic algorithms always produce the same output. Stochastic and deterministic search demonstrate different tradeoffs in the search space [27].

The type of search: If all elements of a repair space are enumerated, it is exhaustive; otherwise the search space may be reduced through some constraints in which the search is constraint-based [25, 37].

Type of execution: The algorithm may be distributed in essence to multiple machines [37]; otherwise it may be centralized to a single machine [43].

Maturity: We propose automatic program repair maturity model (APRMM) with five levels to assess the maturity level of an automatic program repair technique. These levels are as follows: Proof-of-concept (PoC), confidence or reliability (CoR), broad recognition (BR), optimization (OPT), and industrial practice (IP). The descriptions for these levels are shown in Table IV. Drawing strong conclusions based on the results provided by exact statistical tests, comparisons, human studies and similar means are required at each level. At this time, no technique is at level 5. Few techniques moved from level 3 toward 4; however, they are not yet placed at level IV.

Scalability: This criterion has been previously defined by Le Goues [26]. Here, we extend this definition. Particularly, three factors affect scalability: capability to work on real-world programs with real-world defects; ability to repair programs from thousands to millions lines of code; comparable to human repair in terms of time and monetary cost. Lack of the first factor makes an automatic program repair technique unscalable. If the technique satisfies the first factor, it is considered as partially-scalable. A technique with either the second or the third factors, in addition to the first one, is considered semi-scalable. Full-scalable technique satisfies all three factors.

Reliance on formal specification: Some of the techniques especially those that make sound patches [31, 41], rely on different kinds of formal specifications to evaluate or construct patches [8]. The exact type of specification should be explicitly specified. Currently, however, formal specifications are not common in practice, they need heavyweight tools, they are hard to use and time-consuming [26].

Expressivity: An expressive automatic program repair technique is one that can repair various types of programs (generic programs) with various types of defects (generic repair) [26].

Human competitive: An automatic program repair technique is said to be human competitive if it satisfies four properties: full-scalable, expressivity, minimum-quality repair (see overall patch quality), and comparable to human repair in terms of time and monetary cost [26]. Note that full-scalability implies satisfaction of the fourth property. Moreover, if the technique is multi-language (see language generality), then it can be called super human competitive.

Desired functionality and evidence to the fault: In order to repair a fault, a technique needs to know the desired behavior (to preserve) and evidence to the fault (to eliminate). Positive test cases typically are employed to show the desired behavior and negative test cases to expose the fault. This is effectively practical because test cases are always available or can be generated. However, test cases do not provide sufficient confidence to the quality of software [3, 28]. By contrast, formal specifications provably evaluate programs. However, they are rare in practice, especially for legacy software. Hybrid techniques may leverage test cases and formal specifications to benefit from the both worlds.

Instrumentation: Some techniques require adding extra code to the program to gather different information. This information is typically gathered against the execution of test cases and specify statements that are exercised when execution of positive and negative test cases. Fault localization and fix localization [28] (suitable code to be used for fixing) benefit from this information.

Most time-consuming part: Some parts of an algorithm may be bottlenecks and are most time-consuming parts. For example, in test case-based techniques [29, 37] fitness evaluation needs to run the whole positive and negative test cases on a program to measure its desirability. This would be very time-consuming as compared to other parts.

Applied fault localization techniques: Most techniques [3, 25, 37] apply simple fault localization techniques to reduce the infinite space of modifications. For example, statements executed on running positive and negative test cases are identified and weighted with heavier weight on statements that exclusively executed on negative test cases. Some techniques employ particular localization methods such as Tarantula [20]. Effective fault localization for effective program repair is difficult and remains an unsolved problem [25]. Fault localization is roughly impossible for some categories of faults such as nondeterministic ones [25].

Syntactically ill-formed programs: This refers to whether the technique outputs programs that do not compile due to syntactical issues such as imbalance parentheses or the like [13].

Semantically ill-formed programs: The output program of a technique may be semantically wrong due to for example using a variable out of score [13]. Applying strongly-typed GP [33] in evolutionary approaches prevents from this problem.

Target system or context of repair: A technique may be designed to work on, for example, legacy software in C, embedded program in assembly, and so on.

Input: This specifies the requirements of an automatic program repair technique to start. For example, a faulty program along with some positive test cases and a negative test case are typical inputs. Additionally, some of techniques especially evolutionary ones include lots of parameters that

need to be set before the repair task commences. There can be at least two choices: a default parameter setting (which has been shown to be optimal in experiments) is used for all repairs; or the user sets parameters at the beginning of each repair task.

Output: This is the outcome of an automatic program repair technique. A patch is the normal output. The repair technique may make runtime modifications or set a jump to the new code. The output patch can have at least two levels: an initial patch which is the immediate output of repair technique and the final patch which is minimized or optimized version of initial patch against redundant codes added during evolution.

Available tool support: Often, a prototype tool is constructed to support for evaluation and effectiveness of a proposed technique. This tool can be made public and available to download for other researchers to further investigate the technique, to reproduce the results, or to directly compare the obtained results with their own. The other case is to explain the characteristics and technical aspects of the tool and the obtained results.

Time and space complexity: Timing and space complexities are differently measured for each technique. For an evolutionary-based technique, the number of fitness evaluations may be a scenario-independent metric to estimate time complexity. Similarly, the memory needed to construct individuals and generations may be an indicative of the space complexity.

Industrial popularity and acceptance: This refers to the fact that whether the technique is used in industrial practice or not. It can be determined according to valid available reports.

Real-world share: This criterion measures the amount of real-world software produced by the language on which the technique works and is measured by three elements: the number of available code repositories for that language; the number of available jobs; and the number of web searches for that language. These three pieces of information can be simply obtained from online websites such as GitHut[2], sitepoint[3], and TIOBE Index[4] respectively.

Academic popularity and acceptance: There exist at least two cases where a technique becomes popular in academia: it is often used as baseline for comparison; its best practices and design decisions will be adapted in developing new techniques.

Type of studies: Systematic studies [27, 29] follow a certain methodology and are reproducible. They are conducted according to standard and well-defined procedures such that the results can be generalized and possible biases in datasets, techniques, and underlying tools are minimized. Some other studies may not be established systematically and thus are less reliable.

Type of evaluations: Longitudinal studies [27] are several defects in a program over the time. Each time a defect is repaired and in case of exposing another defect, the program undergoes another repair. Latitudinal studies [27] include various programs and defects and measure the success of repair in multiple programs.

Source of dataset: Datasets of experimental studies in the context of fault detection, localization, and program repair may be taken from three sources [27]: datasets in the wild are those that were taken from ad hoc case studies, manual search through databases, industrial partnerships, and the like. Datasets might be collected through systematic search to help prevent from biases. They may also come from existing repositories. For the last case, the design purposes and suitability of datasets should be considered to make sense for the current study. Existing datasets are either benchmark developed by other developers or a typical non-benchmark dataset just to establish an evaluation.

Using cloud environments: When evaluating a certain technique, it can be run on (public) cloud resources to save time.

Type of experiments: Controlled experiments are conducted against programs with a few reproducible defects. These programs are often of medium and small sizes with synthesized test suites generated to satisfy a certain coverage criteria. Besides, they often contain seeded faults that are not indicative of real-world faults. By contrast, case studies are typically conducted on real-world (larger) programs with real faults. The obtained results are more general provided that biases in dataset were reduced or minimized. Controlled experiments help perform proof-of-concept evaluations of a technique; however due to major biases often present in dataset, the results typically cannot be generalized.

Type of programs: A technique may be capable of handling programs of various arbitrary types or it may work only on application-specific programs such as web servers [43]. The repair technique may have potential to be employed to construct a hybrid system such as the closed-loop long running system comprising web server and an intrusion-detection system [25]. The first class of studies implies generic techniques of program repair or program generality property. To note that there exist techniques that claim to be generic though they lack of sufficient experimental evidence to support for their claim. Nevertheless, we consider them generic.

Size of programs evaluated: The programs used in experiments are of different sizes. We categorize them in four classes: small size stands for programs that have less than 100 lines of code; programs with more than 100 and less than 1000 lines of code are categorized as medium; programs that are larger than 1000 lines of code and less than 100,000 are large; and programs larger than 100,000 lines of code would belong to very large class.

Type of analysis: The results of experiments can be analyzed qualitatively or quantitatively. Moreover, evaluations can be subjective or objective. Objective analysis makes experiments and results reproducible and directly comparable to other techniques. A combination of these cases can also be made.

Metrics: A number of metrics have been used to evaluate automatic program repair techniques. Among them, some need

[2] www.githut.info
[3] www.sitepoint.com
[4] www.tiobe.com

further explanation. Patch readability is important due to maintenance activities and future evolutions of the software. Number of steps refers to the primary operations of a certain technique. Efficiency typically measured by the number of fitness evaluations for evolutionary approaches. The benefit of this choice is that it is independent of any specific scenario to measure the repair time. Average time to repair is typically measured based on wall-clock time. Patch size is usually the number of lines of code. Some of the techniques minimize the obtained patch to eliminate problems such as code bloat and redundant code and the like. These techniques produce an initial patch and a final patch. Measurements are repeated for both of these steps.

Comparison to other techniques: Studies in which available benchmarks were applied can be directly compared to other techniques. Besides, the defects should be reproducible. Experimental setup of comparing studies should be as much consistent as possible.

Defects are reproducible: Defect reproducibility is an essential property to make datasets reusable in other studies. They provide possibility of comparison with other techniques. In order to reproduce a defect and replicate the results, six elements should be available [27]: the source code of the version of program that includes the fault; the test suite as partial or specifications (by invariants, annotations, or supporting documents) as complete indicative of correct behavior and failing test case as evidence to fault; the exact steps to run test suite for a tool; the suitable compiler and its version and compiling scripts; the execution platform to expose the fault including operating system, libraries, particular hardware architecture, and supporting scripts; and the exact configuration and parameter values.

Defect scenarios are well-defined: In order to perform a comprehensive evaluation on a defect, we need defect scenario which requires reproducible defects and human-written patch as an optional element to provide a baseline comparison [27].

Results are replicable: To measure this metric, we should check whether the defects are well-defined and the configuration, experimental setup, and values of parameters are exactly detailed. Note that if the repair technique includes stochastic element, difficulties in replicating of results will arise [27]. An exact report of the experiments with sufficient details mitigates this issue.

The way of patch production: A technique may produce a patch by random modifications [29] or provably produce sound patches relying on formal specifications.

Source of repair code: A repair technique needs to find appropriate code to repair a defect which is called fix localization [28]. This code may come from existing library of program code or pre-defined templates. Another option is to use the code of the program under repair without any limitation. Using the code of the repairing program itself roots from the intuition that a programmer that makes a fault, is likely to address this fault in another location [25, 26]. An alternative way is to use a certain part of program to reduce the large space of fix code and resolve some of the issues. For example, fix space may be defined as statements that have executed by at least one positive test case and their variables are in the scope of target code in which fix code is added [29]. Similar approach can be employed when repairing assembly code [37].

Number of repair alphabets: Various statements of the programming language of the program under repair are alphabets or primitives for repair techniques. Assembly languages have higher number of alphabets than a high-level language such as C. However, they are simpler and often have fixed size.

The size of repair alphabets: As mentioned in previous criterion, the size of statements may be fixed or variable. For high-level languages such as C or Java, the size can be roughly of arbitrary size. For assembly language it can be considered as fixed size.

Complexity of repair alphabets: Complexity of statements is very diverse for high-level languages. More precisely, statements in high-level languages can be combined and written in many (complex) ways. This is not the case for low-level languages.

Human-validated and acceptability: In order to further investigate the quality of the repairs, some studies perform post facto manual code review by human developers. This will substantially increase the confidence to the reported results and effectiveness of the technique.

Comparison to human patches: In case where studies are conducted around standard benchmarks, comparison to human patches will manifest the effectiveness of the technique and gives stronger confidence in the results of evaluation. Note that the benchmark should contain human-written patches per each defect in programs. When patches resulted from automatic techniques are compared with manual human patches in terms of cost (monetary, wall-clock, etc), at least three potential complexities may arise [29]. For one, identifying defects require test cases and test case generation makes additional costs often in development of open-source software development. For two, patches produced by automatic techniques need further inspection and validation by human developers. Finally, human patches are shown in many cases to be imperfect [46] as was reported in Section I.

Runtime overhead: A repair technique may add extra code for runtime monitoring or other similar purposes. This code often leads to running overheads in the repaired program [11].

Large augmented code: The extra code added by repair technique may be so large that makes significant increase in the final code size [39].

Steps for further quality: There exist techniques that validate the quality of patches in multi-level manner. For example, a patch that passes all positive and negative test cases during evolution is considered to be *valid* [3]. After termination of repair algorithm, the output patch is executed against a larger set of test cases to provide more confidence in its quality. A patch that passes this stage is called *robust* [3]. This is essentially effective because validation of program under repair occurs many times during consecutive iterations of

repair algorithm and large sets of test cases make the technique heavily time-consuming.

Internal metrics used to validate: What are internal metrics used to validate the quality of the patch? Compilation of patched programs and testing are such (simple) metrics. Testing are typically accomplished using test cases and bug-inducing inputs. The type of testing differs according to the type of defects and programs. For instance, in the context of security applications, fuzz testing is used to measure the functionality of programs for quality purposes.

External metrics used to validate: A comparison to human-written patches in terms of size and time-to-repair are external metrics to validity. The severity of the defect in real-world software is another external metric. This information need standard benchmarks [27] that report them. However, at the time of writing this paper, they are not available. Validation using the mentioned metrics is an initial step to justify applying repair techniques in industrial practice.

Minimized or optimized: A repair task is very close in spirit to a plastic surgery [16]. After an initial patch is generated with respect to minimal requirements, it may contain redundant or dead code that does not contribute to the correct functionality of the program. A repair technique is expected to leverage additional post-processing step to eliminate any kind of redundancies. Delta-debugging and structural differencing are common techniques used [1, 48]. These cleanup processing may be done on the high-level, assembly code or abstract syntax tree (AST) of the program.

Fitness function: During the process of patch generation, intermediate programs are generated that should be measured for desirability to direct the process towards the correct patch. Particularly this is the case for evolutionary techniques via a fitness function [25]. This function may leverage test cases, formal specifications, or a combination for fitness evaluation. At any rate, measuring desirability by test cases is time-consuming and a major limitation in test case-based techniques. Most of the existing techniques are single-objective in that they target only one aspect of the program.

Overall patch quality: This criterion has three cases: minimum, medium, and high. A patch that passes all positive and negative test cases (i.e. fixes the defect and preserves the required behavior) and does not introduce new faults or vulnerability is a minimum-quality patch. A minimum-quality patch without running overhead or large augmented code (possibly redundant) is a medium-quality patch. Finally, a high-quality patch is a medium-quality patch that is also validated and accepted by human developer. Note that Le Goues defined high-quality patch [26] equivalent to our minimum-quality level. By contrast, our definition here is intended to be more general and comprehensive.

Reliance on test cases: A number of state-of-the-art techniques for program repair need test cases as a major element. Test cases are prevalent in practice or can be generated using huge number of test case generation techniques available [2]. However, exhaustive testing using the whole test suite is infeasible due to constraints in testing time and resources. Thus we need to select an indicative subset of test cases which is a challenging task [47]. Less confidence in the software is another concern.

Source of test cases: How should we find the required test cases? They may come from existing test suites, or can be generated automatically, or may be provided by human developers.

Co-evolution of test cases: As the program evolves and its structure mutates, test cases may no longer validate the evolving program. Arcuri proposed to co-evolve test cases with programs similar to prey and predator in order to help convergence rate [4]. We propose to use test suite augmentation (TSA) techniques [23] to co-evolve test cases.

Measuring quality of test cases: The quality of test cases is mainly measured through code coverage criteria or fault-exposing potential [26].

Method to select test case: Since exhaustive regression testing is infeasible, we need to select a representative set of regression test cases. Test case selection, test suite reduction, test case prioritization [47], or impact analysis [34] techniques are common.

Oracles: A critical requirement of test case-based techniques is an oracle which is the expected output of program against running a test case. Oracles are typically provided in studies. However, full automation of repair techniques necessitates automatic oracle generation [15], which remains a challenging task.

Type of evolutionary approach: GP has been the most applied meta-heuristic in current techniques. However, other methods such as hill climbing were also used.

Code bloat: This is a problem which mostly occurs in GP. When GP modifies the code during evolution, redundant code may be added that does not contribute to the code responsible for repair. There are a number of bloat control mechanisms in the literature [32] to resolve the problem. Existing research employed methods to control bloat in automatic program repair techniques [25].

Type of mutation operator: For the case of GP-based techniques, mutation is an essential operator on individuals. In the literature of common GP techniques, mutation is responsible for exploitation of current individual [36] by changing a single bit. However, mutation in GP-based program repair techniques accounts for both exploration and exploitation [36] by changing a statement. In fact, mutation is customized for program repair context. Three operations often occur for mutation: insert, delete, or replace. This modification may apply on AST [13], assembly code [37], list of edits [29], and so on.

Type of crossover operator: Since developers do not write programs at random, a faulty program is assumed to be close to correct one [3, 10]. Hence, crossover does not significantly contribute in the context of program repair and would have a low rate. One-point crossover and variations of unit crossover have been mostly used in current techniques [36]. Crossback is another type of crossover in which crossover is performed on an individual and the original faulty program [36].

TABLE I. SINGLE INTRINSIC CRITERIA (UPPER), EXTRINSIC CRITERIA, AND SUB-CRITERIA FOR 'EVALUATION' CRITERION

Criterion	Some possible values	T_1	T_2	T_3
The general approach	1) Evolutionary, 2) Non-evolutionary	1	1	1
Repair approach	1) Correct-by-construction, 2) Generate-and-validate	2	2	2
Type of algorithm	1) Deterministic, 2) Stochastic	2	2	2
The type of search	1) Exhaustive, 2) Constraint-based	2	2	2
Type of execution	1) Centralized, 2) Distributed	1	1	1, 2
Maturity	1) PoC, 2) CoR, 3) BR, 4) OPT, 5) IP	1	3	2
Scalability	1) Not scalable, 2) Partially-scalable, 3) Semi-scalable, 4) Full-scalable	2	4	4
Expressivity	1) Yes, 2) No	1	1	1
Human competitive	1) Yes, 2) No	2	1	1
Reliance on formal specification	1) Yes, 2) No	2	2	2
Desired functionality and evidence to the fault	1) Test case, 2) Formal specification, 3) Hybrid	1	1	1
Instrumentation	1) Yes, 2) No	1	1	2
Most time-consuming part	1) Fitness function	1	1	1
Applied fault localization techniques	1) Weighting statements by test cases, 2) Particular technique, 3) Existing technique	2, 3	1	2
Syntactically ill-formed programs	1) Yes, 2) No	2	2	1
Semantically ill-formed programs	1) Yes, 2) No	1	1	1
Target system or context of repair	1) Legacy software, 2) Embedded system, 3) Object-oriented software	3	1	2
Input	Faulty program in various forms (1: source, 2: binary, 3: intermediate), 4) Test cases (4: positive, 5: negative), 6) Formal specifications, 7) Parameter settings	1, 4,5, 7	1, 4, 5, 7	2, 4, 5, 6
Output	1) Textual patch or repaired program, 2) List of edits and changes, 3) Runtime modifications, 4) Jump to the new code	1	2	1
Available tool support	1) Yes, 2) No	2	1	2
Time and space complexity	1) Typical complexities to run evolutionary algorithms such as GP, but optimized due to several design decisions	1	1	1
E – Industrial popularity and acceptance	1) Yes, 2) No	2	2	2
E – Real-world share	#Code repositories	222,852	73,075	2,264
	#Current jobs	18%	9%	0.0%
	#Web searches	19.57%	15.62%	1.81%
E – Academic popularity and acceptance	Comparison? 1) Yes, 2) No	2	1	2
	Adaptation? 1) Yes, 2) No	2	1	2

Evaluation		T_1	T_2	T_3
Sub-Criteria	**Some possible values**			
Type of studies	1) Systematic, 2) Non-systematic	1	2	1
Type of evaluations	1) Longitudinal, 2) Latitudinal	2	2	2
Source of dataset	1) In the wild, 2) Systematic search, 3) Existing data	1	2, 3	3
Using cloud environments	1) Yes, 2) No	3	2	1
Type of experiments	1) Controlled, 2) Case studies	1	2	2
Type of programs	1) Generic, 2) Application-specific	1	1	1
Size of programs evaluated	1) Small, 2) Medium, 3) Large, 4) Very large	1	1, 2, 3, 4	1, 2, 3
Type of analysis	1) Qualitative or 2) Quantitative, 3) Subjective or 4) Objective	2, 4	1, 2, 4	1, 2, 4
Metrics	1) Number of patched defects, 2) Time to repair, 3) Monetary cost, 4) Patch size, 5) Patch complexity or readability, 6) Memory, 7) Success rate, 8) Number of steps required to obtain patch, 9) Efficiency, 10) Number or percent of positive and negative test cases, 11) Program size, 12) Compilation (absolute time or percent of total time), 13) Measurements for original and final patch, 14) Technique-specific metrics such as weighted path	8, 11	2, 4, 7, 9, 10, 11, 12, 13, 14	2, 6, 7, 9, 11, 14
Comparison to other techniques	1) Yes, 2) No	2	2	2
Defects are reproducible	1) Yes, 2) No	2	1	1
Defect scenarios are well-defined	1) Yes, 2) No	2	1	1
Results are replicable	1) Yes, 2) No	2	1	1

TABLE II. 4 CRITERIA ALONG WITH THEIR SUB-CRITERIA

Patch and patch quality			**T_1**	**T_2**	**T_3**
Sub-Criteria	**Some possible values**				
The way of patch production	1) Sound, 2) Stochastic		2	2	2
Source of repair code	1) Pre-defined library, 2) Clamp-variable-to-value, 3) Templates, 4) Existing program code, 5) Generating new code, 6) Fixloc space, 7) Constrained assembly code		4	6	4
Number of repair alphabets	1) Low, 2) High		1	1	2
The size of repair alphabets	1) Fixed, 2) Variable		2	2	1
Complexity of repair alphabets	1) Low, 2) High		2	2	1
Human-validated and acceptability	1) Yes, 2) No		2	1	2
Comparison to human-patches	1) Yes, 2) No		2	1	2
Running overhead	1) Yes, 2) No		2	2	0
Large augmented code	1) Yes, 2) No		2	2	0
Steps for further quality	1) Valid, 2) Robust		1, 2	1	1
Internal metrics used to validate	1) Testing, 2) Compilation		1, 2	1, 2	1, 2
External metrics used to validate	1) Human time-to-repair, 2) Size of human patch, 3) Severity of defect		0	0	0
Minimized or optimized?	1) Yes, 2) No		2	1	2
	When	1) During patch generation, 2) After patch generation	0	2	0
	Applied techniques	1) Delta-debugging, 2) structural difference	0	1, 2	0
	Target of modifications	1) AST, 2) Source code	0	1	0
Fitness function	1) Yes, 2) No		1	1	1
	Objective	1) Single-objective, 2) Multi-objective	1	1	1
	Method of measurement	1) Test case, 2) Formal specification, 3) Code review, 4) Hybrid	1	1	1
Overall patch quality	1) Minimum, 2) Medium, 3) High		1	1	1

Test suite		**T_1**	**T_2**	**T_3**
Sub-Criteria	**Some possible values**			
Reliance on test cases	1) Yes, 2) No	1	1	1
How many of each test case?	(#Positive, #Negative); s = small (< 10), m = medium (10 <= m <= 100), l = large (> 100), '+' = more than one	(+, +)	(s, 1)	(+, 1)
Source of test cases	1) Existing, 2) Generation, 3) Human developers	2	1	1
Co-evolution of test cases	1) Yes, 2) No	2	2	2
Measuring quality of test cases	1) Coverage, 2) Fault-exposing potential	1	0	0
Method to select test case	1) RTS, 2) TCP, 3) TSR, 4) IA	0	0	0
Oracles	1) Existing, 2) Generation	1	1	1

Evolutionary approach		**T_1**	**T_2**	**T_3**
Sub-Criteria	**Some possible values**			
Type of Evol. approach	1) Genetic programming, 2) Hill climbing, 3) Random	1, 2, 3	1	1
Code bloat	1) Yes, 2) No	1	1	1
Type of mutation operator	1) Specific, 2) Pre-customized by GP engine, 3) Rewriting rules based on grammar, 4) Genetic modification in AST, 5) Genetic modification in patch list, 6) Genetic modification in assembly	2	4	6
Type of crossover operator	1) Crossback, 2) One-point, 3) Variations of unit crossover	0	2	0
Code selected for mutation	1) A type of fault localization space	1	1	1
Fix code for mutation	1) Fixloc space, 2) Whole program, 3) A special search operator	3	1	2
Selection code for crossover	1) Cutoff point in fault localization space	0	1	0
Tournament selection	1) Yes, 2) No	1	1	1
Elitism	1) Yes, 2) No	1	0	0

Language		**T_1**	**T_2**	**T_3**
Sub-Criteria	**Some possible values**			
Target language	1) Language-specific, 2) General languages	1 - Java	1 - C	1 - ASM
Language generality	1) Yes, 2) No	2	2	2
Language evaluations	1) Single language, 2) Multiple languages	1	1	2
Level of code	1) Source, 2) Binary, 3) Intermediate (AST, ELF, ASM)	1	1	3
Level of modifications	1) Assembly, 2) High-level statements (Atomic, Sub-stmt.)	2	2	1
Conversion to other form	1) AST, 2) List of edits, 3) List of byte-codes, 4) List of assembly instructions, 5) Lisp code	1	1	0
Global and local variables, data structures, type definitions, and the like	1) Yes, 2) No	1	2	0

TABLE III. DEFECTS CRITERION AND ITS SUB-CRITERIA

Defects		T_1	T_2	T_3
Sub-Criteria	**Some possible values**			
Defect class	1) Generic repair, 2) Defect-specific	1	1	1
Defect type	Deterministic (1: single-threaded, 2: multi-threaded), 3) Concurrent and nondeterministic	1	1, 2	1
Number of defects	1) Single-fault, 2) Multi-fault	1	1	1
Locality of defects	1) Local, 2) Whole program	1	1	1
Priority of defects	1) High-priority, 2) Low-priority	2	1	1
E – Contemporary defects	1) Legacy, 2) Modern, 3) Common	0	1, 3	1, 3

TABLE IV. LEVELS OF APRMM AND DESCRIPTIONS

Level	Description
Level 1 – Proof of Concept (PoC)	The technique is evaluated against small programs that are either hand-coded or taken from other studies or the like. At any rate, they are not real-world programs. Typically a single or limited configuration and parameter set are used.
Level 2 – Confidence or Reliability (CoR)	This level points to the fact that whether a particular technique has been investigated on various types of real-world programs with various sizes and defects to study different tradeoffs and functionalities.
Level 3 – Broad Recognition (BR)	This level includes investigating different design decisions, configurations, and parameter settings of the technique.
Level 4 – Optimization (OPT)	At this level, the exact contexts and circumstances in which the technique best fits are identified.
Level 5 – Industrial Practice (IP)	At this level, the technique is extended and enhanced enough to be scalable, effective, and efficient in industrial practice.

Code selected for mutation: A GP-based technique often selects the target code for mutation from a limited space to overcome space complexity issues and targets more relevant code. Particularly, randomly selecting code from the fault localization space (statements that identified suspicious and further ranked by a fault localization technique) would be a reasonable design decision [43].

Fix code for mutation: In case where the mutation operator tends to insert or replace code, extra code is needed. Often this code is selected from the faulty program itself considering the intuition that the faulty code contains the seeds of its repair as well [26]. The simplest case is to choose the code at random from the whole program. A more intelligent selection would be to choose the statement from the fix localization space [28]: the statement whose variables are in the scope of target statements and at least one positive test case exercises that statement.

Selection code for crossover: Crossover is a major operator in common GP techniques but is less important for GP-based program techniques. Nevertheless, a technique may want to choose a cutoff point in fault localization space (weighted path) [43]. Many other techniques are possible.

Tournament selection: A method of selection at each generation [36].

Elitism: Selecting the fittest individuals to move to the next generation and affecting convergence rate [36].

Target language: Often program repair techniques are designed per each language. GenProg designed for legacy software in C [25]. Other techniques were designed for Eifel [8] and assembly [37] languages. Since program repair needs special treatment for the statements of each language, it seems that near future techniques will be language-specific.

Language generality: A technique that is designed to repair programs of one language may have potential to be easily adapted for repairing of programs in another language. This may occur due to for example close syntactical constructs of both languages. In this case, the technique could be called multi-language.

Language evaluations: Typically techniques designed for high-level languages are evaluated against a single language for which they were designed [22]. However, there exist techniques that designed for and evaluated against multiple low-level languages such as assembly and ELF binary [37].

Level of code: Automatic program repair techniques work on different levels of code, representing different tradeoffs in design and efficiency. Source code [25], binary code [35], and assembly code [37] are some examples.

Level of modifications: A technique may modify only major statements and inner sub-statements remain untouched; or else, it may modify a target statement and all of its sub-statements.

Conversion to other form: Typically, program repair techniques convert original source code to another form for modifications. For example, GenProg converts C source code to AST. In later versions, it stores the modification on a list of edits for efficiency.

Global and local variables, data structures, type definitions: Some of the existing research just work on executable statements and do not take global and local variable, data structures, or type definitions into consideration.

Defect class: Techniques that make generic repair are preferable since they work for multiple classes of defects (defect generality). By contrast, defect-specific techniques work on a single class of defects and thus are optimized for that class and typically produce sound patches. Most of existing

techniques are designed for generic repair for which specification of desired behavior and evidence to the fault should be provided somehow [3, 25]. Moreover, most of the defect-specific or correct-by-construction techniques employ implicit specifications [27]. A number of the properties among several of more formal techniques, especially correct-by-construction ones, are common with the advances in program synthesis such as component-based or test-driven program synthesis [27].

Defect type: At high-level, defects can divided into deterministic and concurrency categories. Most of existing techniques repair deterministic defects [25] and are unsound. This kind of defects can occur in single-threaded or multi-threaded programs. By contrast, the number of techniques that repair concurrency faults is less than the other category but they often produce sound patches. Concurrency faults are typically hard to reproduce [27].

Number of defects: This indicates the number of faults present in the program that can be simultaneously repaired by a technique.

Locality of defects: The root cause of a defect may be assumed to be local or spread through the whole program.

Priority of defects: The defects that are used in evaluating a certain techniques should be high priority; i.e. they are important enough for human developer to repair and influence test suite [29].

Contemporary defects: An automatic program repair technique may repair defects that are rarely found in current software being developed; Or else, it might be designed to deal with the defects that are common in modern software technologies, languages, and programming paradigms. For instance, buffer overflow is a kind of security vulnerability due to a semantic fault in legacy C/C++ programs which is not checked by the compiler. Modern languages such as C# or Java avoid such issues by verifying the index bound of an array. Some defects are specific to modern newer languages such as class cast exception. Finally, a wide range of (logical) faults are common in every language.

IV. EVALUATION

Due to strict limitation in space, we summarize the results and report a fraction of our evaluations just to serve as a proof-of-concept. The rightmost three columns of Table I to Table III are assigned to concise representation of the results. We report the results for three of existing mutation-based techniques and label them as T1 to T3 as follows: T1 stands for the technique presented by Arcuri [3]; T2 shows GenProg which was presented by Le Goues et al. [25]; T3 is related to the technique by Schulte et al. [37]. Then, the possible values of each criterion are numbered and further used in T_i columns. A zero inside a cell means that either the criterion or cases do not make sense or there is no evidence to measure the corresponding criterion. Investigation of other state-of-the-art techniques along with detailed explanation on the results remains for future publications. Next, we present a brief introduction to each of three techniques.

The technique proposed by Arcuri [3] uses three randomized search algorithms including random search as baseline, hill climbing, and GP to evolve faulty programs into correct ones. The technique employs new operators based on fault localization technique to narrow down the space of modifications in the code. This technique works on a large subset of Java language. A tool named JAFF implements the technique and evaluates it on five faulty versions of seven small Java programs with seeded faults. Passing test cases indicate the correct behavior and failing test case provide evidence to the fault.

GenProg [25] is a repair technique based on GP that works on C source programs. It targets eight classes of faults mostly security ones and was evaluated on 16 programs in terms of quantitative and qualitative metrics. The programs are transformed to AST representation in which specialized mutation and crossover operators modify the programs. Fault localization and fix localization are used to narrow down the corresponding space. GenProg employs failing and passing test cases to serve as the evidence to the fault as well as the correct behavior. They are also used to measure fitness for evaluating candidate patches.

The technique by Schulte et al. [37] works on arbitrary faults on embedded systems where limited resources are available. Using GP approach to repair programs, it works on assembly and ELF binary code. Besides, random fault localization was employed to provide performance and narrow the search space. The technique was evaluated on 12 C programs and a C++ faulty program. It also uses test cases to evaluate candidate fixes. While the techniques by Arcuri and Le Goues need instrumentation to collect initial information of the code, this technique is free from code instrumentation.

V. DISCUSSION AND LIMITATIONS

When comparing two techniques using the mentioned criteria, the significance of each criterion should strongly be considered. Some properties for an automatic program repair are critical and the other may be less important. In other words, at least two categories of significance exist among criteria namely major and minor. For instance, capability of a technique to handle various types of faults and programs and thus its expressive power could be more important than timing costs. Hence, this importance can introduce weights for criteria. In fact, if we want to combine the results of all criteria for a technique to measure its overall functionality and performance and compare to another technique, this weighting should be incorporated somehow. Another concern may arise by the fact that some criteria have many degrees and limiting them to some discrete cases leads to very coarse-grain measurements. The net effect is lack of proper discrimination among techniques. For instance, “generic repair” for “defect class” is roughly an approximation of reality. The technique might have potential only to handle several types of *security* defects (and not all of them) and classified as a generic repair technique against a technique that can handle only atomicity violation fault and thus is defect-specific. At the same time, another technique that can handle various types of defects in web application is again considered as generic repair. In fact such criteria are by nature very fuzzy and subjective.

A similar problem occurs for criteria that are influenced by several elements and are measured only in Binary manner such as “yes” and “no”. For example, “human competitive” criterion does well illustrate this situation. It consists of four sub-criteria and has two possible cases, “yes” and “no”. If the technique does not satisfy a sub-criterion such as full-scalability, we classify the technique as non-human competitive. However, it may actually be less human competitive compared to the ideal case not non-human competitive.

Finally, we gave some possible cases for each criterion as guidelines. We do not claim that the list of values is comprehensive and they are by no means complete. The list of values per each criterion may be extended accordingly.

VI. RELATED WORK

Evaluation Criteria. Programming languages and model transformation approaches have been evaluated in many studies using different criteria. Readability, writability, and reliability are major criteria with various sub-criteria used to evaluate PLs [38]. Similarly, bidirectionality, reusability, and rule inheritance are common criteria for the assessment of MTLs in the context of model-driven software engineering (MDSE) [7, 24]. However, most of these criteria are qualitatively measured and are too vague for research community to agree on. This is why subjective measurements are common in these contexts despite the fact that objective measurements were also employed to mitigate the subjectivity. In this paper, we introduced several criteria for objective measurement and evaluation of the automatic program repair techniques to direct this young research subfield.

Automatic Program Repair. Preliminary work on program repair involves substantial limitations such as heavy constraints on the type of bugs or the need to formal specifications. Wotawa and Stumptner made early attempts on bug-fixing by considering a single fault model [40]. It exhaustively searches the state space of program to find a correct one. Using static analysis and pattern matching, Deeprasertkul et al. presented a technique [9] to find and repair pre-defined bugs.

Arcuri proposed the use of GP for the task of general program repair for the first time [3]. This technique needs only positive and negative test cases without any assumption on the type of bugs. Furthermore, it does not have any pre-defined pattern for patches and the candidate patches are evaluated using a fitness function. Inspired by the proposal of Arcuri, Weimer et al. started to study the application of GP to automatic repair of real-world programs with real faults [25, 28, 43]. They investigated different representations [30] and fitness functions [12] on programs taken systematically from real software repositories [29]. Dallmeier et al. introduced PACHIKA [8] to generate fixes according to the difference between normal and anomalous behavior. These techniques and that of Arcuri [3] are dependent upon test cases to validate patches and have at least two drawbacks; lack of confidence to the results of test case-based evaluation and longer repair time due to numerous test case executions.

Weimer presented an automatic method for patch generation [44], which defined formal safety specifications in the form of finite state machine. Demsky et al. presented a technique [11] for data structure repair [24] based on formal specification of consistency for a data structure. Sound techniques to repair concurrency bugs were presented by Bradbury et al. [6], Jin et al. (AFix) [19], and Liu and Zhang (Axis) [31]. Most of such techniques soundly patch atomicity violation faults. Defect-specificity of such techniques and working based on formal specifications (which are rare in practice) are major shortcomings of these techniques. ClearView [35] which was proposed by Perkins et al. uses runtime monitoring in binary code level to specify failing executions and then generates the candidate patches. PAR is another recent technique [22], proposed by Kim et al., for patch generation of Java programs based on the knowledge gained by manual inspection of human-written patches. All of these techniques have just evaluated their own techniques in isolation by some metrics. However, none of them presented a general framework to evaluate and compare automatic program repair techniques against each other. This paper concerned itself to bridge this gap.

Automatic Program Repair Evaluation. Monperrus [49] and Qi et al. [50] presented some evaluation criteria for the assessment of automatic program repair techniques. However, they either used a small number of criteria [49] or their criteria [50] are limited and specifically targeted towards evaluating few of existing techniques. Moreover, they often need to run the corresponding tools for which some issues may preclude. Even in case of running the tools extensive investigations are required to draw conclusions. By contrast, our criteria are intended to be general, comprehensive, objective, easy to measure, and independent of running tools.

VII. CONCLUSION

This paper introduced a family of criteria for the assessment of automatic program techniques. Since 2009 a number of promising techniques to automatic program repair have been presented and this subfield is mature enough to evaluate existing techniques to direct future research. To formalize this evaluation and make it more systematic, we proposed criteria with objective measurement capability. One of the criteria is the amount of maturity for which we presented a maturity model. We also evaluated three of existing techniques using our criteria and compared them against each other and gave direction for future work. To our knowledge, this is the first report to present comprehensive evaluation criteria for automatic program repair context.

REFERENCES

[1] Al-Ekram, R., Adma, A., & Baysal, O. (2005, October). diffX: an algorithm to detect changes in multi-version XML documents. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research* (pp. 1-11). IBM Press.

[2] Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., ... & McMinn, P. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*,*86*(8), 1978-2001.

[3] Arcuri, A. (2011). Evolutionary repair of faulty software. *Applied Soft Computing*,*11*(4), 3494-3514.

[4] Arcuri, A., & Yao, X. (2008, June). A novel co-evolutionary approach to automatic software bug fixing. In *Evolutionary Computation, 2008. CEC*

2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on (pp. 162-168). IEEE.

[5] Ballou, M. C. (2008). Improving software quality to drive business agility. White paper. *International Data Corporation*.

[6] Bradbury, J. S., & Jalbert, K. (2010, September). Automatic repair of concurrency bugs. In *International symposium on search based software engineering—fast abstracts* (pp. 1-2).

[7] Brambilla, M., Cabot, J., & Wimmer, M. (2012). Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, *1*(1), 1-182.

[8] Dallmeier, V., Zeller, A., & Meyer, B. (2009, November). Generating fixes from object behavior anomalies. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering* (pp. 550-554). IEEE Computer Society.

[9] Deeprasertkul, P., Bhattarakosol, P., & O'Brien, F. (2005). Automatic detection and correction of programming faults for software applications. *Journal of Systems and Software*, *78*(2), 101-110.

[10] DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, (4), 34-41.

[11] Demsky, B., Ernst, M. D., Guo, P. J., McCamant, S., Perkins, J. H., & Rinard, M. (2006, July). Inference and enforcement of data structure consistency specifications. In *Proceedings of the 2006 international symposium on Software testing and analysis* (pp. 233-244). ACM.

[12] Fast, E., Le Goues, C., Forrest, S., & Weimer, W. (2010, July). Designing better fitness functions for automated program repair. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation* (pp. 965-972). ACM.

[13] Forrest, S., Nguyen, T., Weimer, W., & Le Goues, C. (2009, July). A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation* (pp. 947-954). ACM.

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

[15] Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2013). A comprehensive survey of trends in oracles for software testing. *University of Sheffield, Department of Computer Science, Tech. Rep. CS-13-01*.

[16] Harman, M. (2010). Technical Perspective Automated Patching Techniques: The Fix Is In. *Communications of the ACM*, *53*(5).

[17] Jeffrey, D. B. (2009). *Dynamic state alteration techniques for automatically locating software errors* (Doctoral dissertation, UNIVERSITY OF CALIFORNIA RIVERSIDE).

[18] Jeffrey, D., Feng, M., Gupta, N., & Gupta, N. (2009, May). BugFix: A learning-based tool to assist developers in fixing bugs. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on* (pp. 70-79). IEEE.

[19] Jin, G., Song, L., Zhang, W., Lu, S., & Liblit, B. (2011). Automated atomicity-violation fixing. *ACM SIGPLAN Notices*, *46*(6), 389-400.

[20] Jones, J. A., & Harrold, M. J. (2005, November). Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (pp. 273-282). ACM.

[21] Kaleeswaran, S., Tulsian, V., Kanade, A., & Orso, A. (2014, May). Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 266-276). ACM

[22] Kim, D., Nam, J., Song, J., & Kim, S. (2013, May). Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering* (pp. 802-811). IEEE Press.

[23] Kim, Y., Zu, Z., Kim, M., Cohen, M. B., & Rothermel, G. (2014, March). Hybrid directed test suite augmentation: An interleaving framework. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on* (pp. 263-272). IEEE.

[24] Kusel, A., Schönböck, J., Wimmer, M., Kappel, G., Retschitzegger, W., & Schwinger, W. (2013). Reuse in model-to-model transformation languages: are we there yet?. *Software & Systems Modeling*, *14*(2), 537-572.

[25] Le Goues, C., Nguyen, T., Forrest, S., & Weimer, W. (2012). GenProg: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, *38*(1), 54-72.

[26] Le Goues, C. (2013). *Automatic program repair using genetic programming* (Doctoral dissertation, University of Virginia).

[27] Le Goues, C., Holtschulte, N., Smith, E. K., Brun, Y., Devanbu, P., Forrest, S., & Weimer, W. The ManyBugs and IntroClass benchmarks for automated program repair. *IEEE Transactions on Software Engineering. In Press*.

[28] Le Goues, C., Forrest, S., & Weimer, W. (2013). Current challenges in automatic software repair. *Software Quality Journal*, *21*(3), 421-443.

[29] Le Goues, C., Dewey-Vogt, M., Forrest, S., & Weimer, W. (2012, June). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*(pp. 3-13). IEEE.

[30] Le Goues, C., Weimer, W., & Forrest, S. (2012, July). Representations and operators for improving evolutionary software repair. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation* (pp. 959-966). ACM.

[31] Liu, P., & Zhang, C. (2012, June). Axis: Automatically fixing atomicity violations through solving control constraints. In *Proceedings of the 34th International Conference on Software Engineering* (pp. 299-309). IEEE Press.

[32] Luke, S., & Panait, L. (2006). A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, *14*(3), 309-344.

[33] Montana, D. J. (1995). Strongly typed genetic programming. *Evolutionary computation*, *3*(2), 199-230.

[34] Orso, A., Apiwattanapong, T., Law, J., Rothermel, G., & Harrold, M. J. (2004, May). An empirical comparison of dynamic impact analysis algorithms. In*Proceedings of the 26th International Conference on Software Engineering* (pp. 491-500). IEEE Computer Society.

[35] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., ... & Rinard, M. (2009, October). Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (pp. 87-102). ACM.

[36] Poli, R., Langdon, W. B., McPhee, N. F., & Koza, J. R. (2008). *A field guide to genetic programming*. Lulu. com.

[37] Schulte, E., DiLorenzo, J., Weimer, W., & Forrest, S. (2013). Automated repair of binary and assembly programs for cooperating embedded devices. *ACM SIGARCH Computer Architecture News*, *41*(1), 317-328.

[38] Sebesta, R. W. (2012). *Concepts of Programming Languages*. 10th Ed., Pearson.

[39] Sidiroglou, S., Giovanidis, G., & Keromytis, A. D. (2005). A dynamic mechanism for recovering from buffer overflow attacks. In *Information security*(pp. 1-15). Springer Berlin Heidelberg.

[40] Stumptner, M., & Wotawa, F. (1997, January). Model-based program debugging and repair. In *Proc. Ninth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE'96)* (pp. 155-160).

[41] Wei, Y., Pei, Y., Furia, C. A., Silva, L. S., Buchholz, S., Meyer, B., & Zeller, A. (2010, July). Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis* (pp. 61-72). ACM.

[42] Weimer, W., Fry, Z. P., & Forrest, S. (2013, November). Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*(pp. 356-366). IEEE.

[43] Weimer, W., Nguyen, T., Le Goues, C., & Forrest, S. (2009, May). Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering* (pp. 364-374). IEEE Computer Society.

[44] Weimer, W. (2006, October). Patches as better bug reports. In *Proceedings of the 5th international conference on Generative programming and component engineering* (pp. 181-190). ACM.

[45] Wirsing, M., & Hoelzl, M. (2011). *Rigorous Software Engineering for Service-oriented Systems: Results of the SENSORIA Project on Software Engineering for Service-oriented Computing* (Vol. 6582). Springer Science & Business Media.

[46] Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., & Bairavasundaram, L. (2011, September). How do fixes become bugs?. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (pp. 26-36). ACM.

[47] Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, *22*(2), 67-120.

[48] Zeller, A. (1999, January). Yesterday, my program worked. Today, it does not. Why?. In *Software Engineering—ESEC/FSE'99* (pp. 253-267). Springer Berlin Heidelberg.

[49] Monperrus, M. (2014, May). A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 234-242). ACM.

[50] Qi, Z., Long, F., Achour, S., & Rinard, M. (2015). An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Software Testing and Analysis (ISSTA), 2015 International Symposium on* (pp. 24-36). IEEE.

[51] Smith, E. K., Barr, E. T., Le Goues, C., & Brun, Y. (2015, August). Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (pp. 532-543). ACM.

[52] Qi, Y., Mao, X., & Lei, Y. (2013, September). Efficient automated program repair through fault-recorded testing prioritization. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on* (pp. 180-189). IEEE.

[53] Motwani, R., & Raghavan, P. (1995). *Randomized Algorithms*. Cambridge University Press.