Verification Runtime Analysis: Get the Most Out of Partial Verification

Martin Ring*, Fritjof Bornebusch*, Christoph Lüth*†, Robert Wille*‡, Rolf Drechsler*†
*Cyber-Physical Systems, DFKI, Bremen, Germany
†Mathematics and Computer Science, University of Bremen, Germany
‡Integrated Circuit and System Design, Johannes Kepler University Linz, Austria

***Abstract*—The design of modern systems has reached a complexity which makes it inevitable to apply verification methods in order to guarantee its correct and safe execution. The verification methods frequently produce proof obligations that can not be solved any more due to the huge search space. However, by setting enough variables to fixed values, the search space is obviously reduced and solving engines eventually may be able to complete the verification task. Although this results in a *partial verification*, the results may still be valuable — in particular as opposed to the alternative of no verification at all. However, so far no systematic investigation has been conducted on which variables to fix in order to reduce verification runtime as much as possible while, at the same time, still getting most coverage. This paper addresses this question by proposing a corresponding *verification runtime analysis*. Experimental evaluations confirm the potential of this approach.**

I. Introduction

While capturing almost all aspects of our daily life, embedded and cyber-physical systems reach a complexity which poses significant challenges to their design. As those systems are usually expected to be free from errors, particularly if they are safety-critical, verification becomes inevitable in order to guarantee their correct and secure execution. To this end, numerous methods such as simulation [1], [2], [3], emulation [4], [5], or formal verification [6], [7] have been proposed and are utilized in modern design flows.

The corresponding verification processes frequently produce proof obligations which pose serious challenges to the reasoning engine. The ever-increasing complexity yields huge search spaces which often cannot be covered within the given time limits. Together with pressing time-to-market constraints, this eventually forces designers to release the system even if 100% functional correctness has not been ensured. Consequently, bugs are frequently allowed to escape into the final product — a problem which is usually referred to as the *verification gap*.

A simple albeit effective way to reduce the search space and, by this, reduce the runtime of the reasoning engine is to set a certain amount of the given variables to a fixed value. Although this obviously results in less than 100% coverage, such a result is potentially offering more value than no result at all. In fact, rather than having to abort the verification process entirely because of time limitations, fixing some variables and accepting that the verification process covers a strictly defined subset at least provides a partial (but hopefully substantial) verification result. This is particularly the case in systems containing a large number of free variables representing configuration variables, parameters, sensor inputs, or user inputs, which will be substituted with concrete values after deployment anyway. Then, each partial verification result provides correctness guarantees at least for some scenarios in which the embedded system is eventually deployed.

While the general methodology has been explored in earlier work [8], the question of which variables to fix in order to achieve the largest reduction of verification runtime has not been addressed at all. While in theory fixing one Boolean variable would reduce the search space and runtime by half, actual instances show a much smaller and less uniform reduction due to the optimizations by the proof engine. Some variables may hardly have an effect at all, while others may immediately cut down a day-long verification process to a few moments. Because of that, it is essential for verification engineers to have a clear understanding about the impact of fixing a particular variable to the verification runtime, so they can follow the general idea of fixing some variables in order to get a partial result out of the verification process covering as many cases as possible. However, no systematic investigation on this effect has been conducted so far.

In this paper, we introduce a methodology to analyze verification runtime, and to measure it practically in a meaningful way. The main problem is *how many* and *which* variables are fixed. For this, we first state a formal criterion describing an optimal solution to this problem. Based on that, a cost function is defined which can be used to employ stochastic and heuristic methods in order to eventually determine solutions optimized for this goal. Using a proof-of-concept implementation based on evolutionary algorithms, we were able to confirm the potential of the proposed methodology. In fact, experimental evaluations confirmed that this methodology indeed determines a set of variables to be fixed which keeps the verification runtime within reasonable limits while still covering as much as possible of the search space.

In general, the methodology works for any other heuristic and is independent from both the reasoning engine and the underlying logical language, *i.e.* we treat the reasoning engine as a opaque black box which either proves a proposition or not. This offers a complementary verification approach which addresses the verification gap, not by incremental improvements (excessively investigated in the past and still struggling with the exponential growth of the underlying complexity), but an alternative scheme that accepts that 100% completeness might not always be possible but still aims for getting the most out of a verification task.

Research supported by BMBF grant SELFIE (grant no. 01IW16001) as well as the LIT Secure and Correct System Lab (funded by the State of Upper Austria).

978-3-9819263-4-7/DATE20/©2020 EDAA

The remainder of this work is structured as follows: Sect. II first defines the problem, both informally and formally. Based on that, Sect. III sketches the main ideas of the proposed methodology – including a definition for an optimal solution. A proof-of-concept implementation of the methodology based on evolutionary algorithms is described in Sect. IV. Finally, results obtained by experimental evaluations are summarized in Sect. V before the paper is concluded in Sect. VI.

II. Reducing Verification Runtime

In this section, we describe the main idea of the proposed verification runtime analysis. Recall that our goal is to determine how many and which variables to fix in order to achieve as much of a reduction in verification runtime as possible (and, by this, getting as much out of a partial verification as possible, rather than no result at all).

A. Fixing Free Variables

In the following, we consider a verification problem as a single proposition[1] ϕ that shall be proven with contemporary reasoning engines such as SAT solvers [9], [10], SMT solvers [11], [12], [13], or similar. The particular logic and reasoning engine used do not matter, as long as the proof procedure is fully automatic. We are interested in problems that cannot be solved using the given resources, where verification process would be aborted and the verification engineers would get no result at all.

In contrast, when enough variables are set to fixed values (we say the variables are *fixed*), the search space is reduced and the reasoning engine eventually yields a verification result. Even if such a result would not cover all instances of the verification problem, proving an instance of ϕ may still be of potential value.

This yields the questions *how many* and *which* variables should be fixed. In an idealized scenario, answers to these questions would be as sketched in the following example:

Example 1. *Consider a verification problem ϕ whose complete verification takes a certain time T_ϕ. Setting* all *variables of ϕ to a fixed value will allow for a more or less instantaneous completion of the verification task.*[2] *Moreover, in an idealized scenario, the proof time would be reduced exponentially with respect to the number of fixed variables. This is sketched by the* green solid line *in Fig. 1, showing an idealized graph plotting the (presumed) average proof time (in logarithmic scale) over the number of fixed variables. In this idealized scenario, answers to the two questions raised above are trivial: It does not matter* which *variables are fixed (any differences are averaged out) and the* number *is basically determined by the available resources; i.e. the available time (on the y-axis) determines the corresponding number of variables (on the x-axis).*

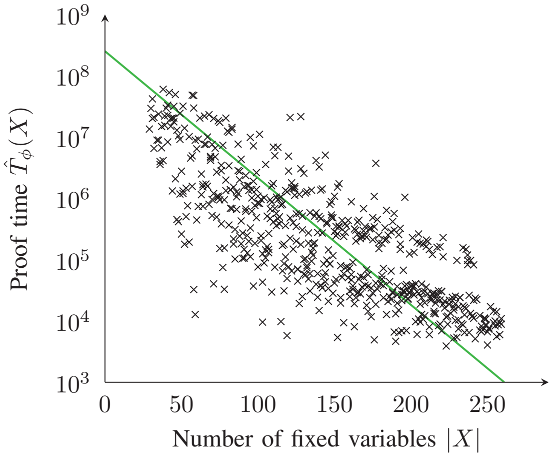

Fig. 1. Runtime of a representative verification problem.

However, such an idealized scenario almost never occurs. In fact, it quickly becomes clear that the relation between the number of fixed variables and the proof time is rather erratic. Again, this is illustrated by means of an example:

Example 2. *Consider a representative benchmark taken from the SMT-LIB benchmark library [14]*[3] *for which the relation between proof time and sets of fixed variables have been evaluated. The obtained results are shown in Fig. 1. Here, each data point at (n, t) corresponds to the average proof time t of ϕ with n different variables fixed. As can clearly be seen, there is no obvious relation between proof time and the number of fixed variables. Instead, there are a number of data points which are better than the idealized scenario discussed before in Example 1, i.e. points which lie below the diagonal in Fig. 1.*

As illustrated by these observations, simply fixing a certain number of variables of ϕ often does not yield the desired result. Moreover, a straightforward enumeration is not suitable because of the following issues:

- *Complexity*: Even if the number of variables to be fixed is given as say m, there still would be 2^m possible combinations left to try out.
- *Quality*: Proving ϕ with all variable fixed except for one certainly will be very fast, but will hardly give more insight than an aborted verification process. Hence, verification engineers are interested in restricting only as little variables as needed.
- *Effectiveness*: We are particularly interested in verification problems which cannot be completed due to a time-out; here, we are looking for the data points which lie as much to the left in Fig. 1 as possible, but are still below the time-out. These are hard to find by enumeration as one would run into time-outs a lot.

In summary, we are interested in finding the data points in the lower left corner of Fig. 1, which represent instances where only a small number of variables are fixed (*i.e.* the instance is as little restricted as possible), while at the same time runtime is kept small.

[1] Note that a number of verification conditions can of course always be combined into a single proposition by conjoining them. Furthermore, we consider all variables to be Boolean. This does not restrict the methodology (because other types such as integer variables can be encoded as bit vectors) but significantly simplifies the exposition in the following.

[2] In some logics (*e.g.* with nested quantifiers), this might not be the case, but the general principle that proving ground term propositions is much faster is still valid.

[3] The SMT-LIB library is composed of various benchmarks to challenge reasoning engines — including many problems from the verification of circuits and systems — and, hence, provides a representative source of problems to be considered within the scope of this work.

B. Verification Runtime Analysis

The observations and discussions from above motivate an analysis of the verification runtime in order to determine the best possible data points. This poses an optimization problem which has not been clearly defined so far. In the following, a definition is provided which is used as a basis for the remainder of this work.

The inputs of the optimization problem are as follows:

- A reasoning engine (such as a SAT solver, SMT solver, or similar) which, given a proposition, either returns true, false or does not terminate.
- A proposition ϕ which takes the time T_ϕ to prove using the reasoning engine.

Note that the actual time units are irrelevant, but we assume that the time is deterministic and reproducible.[4] Furthermore, the proof procedure may not terminate; in that case, T_ϕ is a time unit which is larger than any finite one.

Let $\mathit{FV}(\phi)$ denote the set of free variables occurring in ϕ. Given a subset $X \subseteq \mathit{FV}(\phi)$ of the free variables of ϕ, we define the *average verification runtime* $\hat{T}_\phi(X)$ as the average time it takes to prove ϕ with the variables in X set to ground terms, and the rest in $\mathit{FV}(\phi) \setminus X$ kept free. That is, $\hat{T}_\phi(X)$ is the *expected* verification runtime if the variables in X are set to an arbitrary fixed value. We have found that, for a given X, we can approximate $\hat{T}_\phi(X)$ with a small number (five) of representative samples.

Example 3. *Fig. 1 also provides an illustration of this notation. The figure plots the average verification runtime $\hat{T}_\phi(X)$ at the y-axis over the number of fixed variables (i.e. the cardinality $|X|$ of X) at the x-axis for the representative benchmark discussed in Example 2. Each data point in the diagram corresponds to $(|X|, \hat{T}_\phi(X))$ for a particular set X of variables to be fixed.*

The aim of the analysis is to determine a set X which is as small as possible while still corresponding to a reasonable average verification time. To this end, we need to investigate how the function mapping X to $\hat{T}_\phi(X)$ behaves. With $\emptyset \subseteq X \subseteq \mathit{FV}(\phi)$, we can state that

- $\hat{T}_\phi(\mathit{FV}(\phi))$ is the minimum, because it proves a ground term (no free variables), and
- $\hat{T}_\phi(\emptyset) = T_\phi$ is the maximum, because we prove the original proposition ϕ.

However, in between, the behaviour is not so well defined. From the above, we might guess that the smaller the set X, the larger the average verification runtime (*i.e.* $\hat{T}_\phi(X)$ is anti-monotone over the size of the variable set), but this turns out not be true. Given two different subsets $X, Y \in \mathit{FV}(\phi)$, we have

$$|X| \leq |Y| \not\Longrightarrow \hat{T}_\phi(X) \geq \hat{T}_\phi(Y). \tag{1}$$

In other words, increasing the number of fixed variables does not necessarily decrease the average verification runtime. Hence, the problem remains how to determine an *optimal subset* X of variables such that the average verification time $\hat{T}_\phi(X)$ is still acceptable.

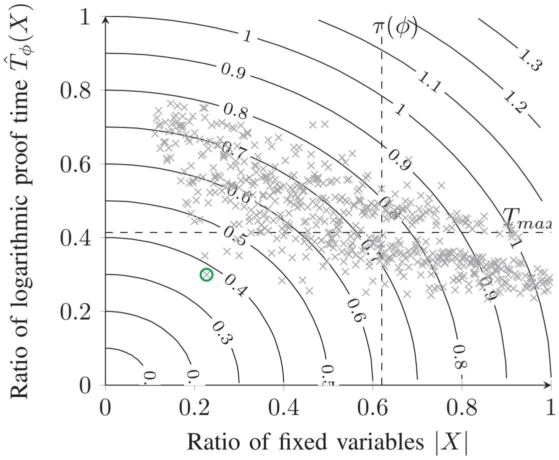

Fig. 2. Contour of the cost function.

III. Proposed Solution

The problem motivated and introduced above can be addressed in a number of different ways. However, a straightforward enumerative approach does not work here, as the optimal solution for say two variables is not necessarily a subset of the optimal solution for three variables (they are not even guaranteed to intersect at all). So we lack an order structure on the space of possible solutions — all subsets of $\mathit{FV}(\phi)$ — which can guide a search process to the optimal solution. To determine a solution in a rather unstructured space of solutions (such as this one), a number of probabilistic and heuristic approaches are available (*e.g.* simulated annealing, evolutionary algorithms, *etc.*). However, all of these need a dedicated *cost function* to guide the search.

To get this cost function, we propose a geometric interpretation of the data points in Fig. 1. We are looking for the one which is closest to the bottom left corner, *i.e.* which has the least distance to the origin. Geometrically, if we consider our data points as vectors, we are looking for the vector with the smallest length. In order to make the cost function behave uniformly for different propositions ϕ, we scale both axes with the maximum, *i.e.* the size of the set of fixed variable $|X|$ with the total number of free variables $|\mathit{FV}(\phi)|$ and the average verification runtime $\hat{T}_\phi(X)$ with the proof time of the original proposition T_ϕ. Thus, for a set X of variables to be fixed, our cost function is

$$q(X) \triangleq \sqrt{\left(\frac{|X|}{|\mathit{FV}(\phi)|}\right)^2 + \left(\frac{\log(\hat{T}_\phi(X))}{\log(T_\phi)}\right)^2}. \tag{2}$$

Example 4. *Fig. 2 visualizes the contours of the cost function q from Eq. (2). The theoretical optimum lies at $q(0,0) = 0$. When applied to the results of Fig. 1, a ranking of the data points becomes apparent, ordering the data points by the distance to the origin (highlighted by solid lines in Fig. 2). Considering this as cost metric, the optimal solution is the point marked with a green circle in Fig. 2.*

Our cost function requires a concrete value for T_ϕ which can only be approximated, as we are considering propositions where T_ϕ is very large (in practice, a time-out). Hence, we need an upper

[4] In the experiments summarized below, we use the number of elementary operations of the SMT solver *Z3* [11] as time unit (`rlimit` count), since this is deterministic and independent of architecture or memory.

limit for the solutions to consider during the analysis, otherwise we would constantly run into time-outs. Given an upper limit T_{max} which is considered acceptable for the analysis, the *threshold* $\tau(\phi)$ is defined as the number of variables to be fixed such that the average verification runtime is still below T_{max}. The value of $\tau(\phi)$ can be efficiently approximated *e.g.* through a binary search. This confines the number of data points to be considered to the ones which can be analyzed within acceptable runtime.

Example 5. *For the example considered above, assume an acceptable time limit T_{max}. Based on this, approximate $\tau(\phi)$ as illustrated by the dotted lines in Fig. 2. Now, only the data points between the left bottom corner and these lines lines are considered during analysis. This way, it is ensured that a good solution is derived while, at the same time, the analysis time remains efficient.*

Using this cost function and the threshold, any heuristic method of choice can be applied to determine a set X such that $q(X)$ is minimized – this will be our desired solution.

IV. Implementation

In this section, we describe one possible implementation of the proposed solution described above. As a heuristic, we decided to use *Evolutionary Algorithms* (EAs, [15], [16]) which represent an established method to solve optimization problems, with applications in hardware design [17], [18] or multi-objective optimization [19], [20]. In the following, we briefly review the basic concepts of EAs in general, before discussing how those concepts are utilized in order to address the problem.[5]

A. Evolutionary Algorithms

Evolutionary algorithms are stochastic search methods inspired by the natural evolution process. The goal is to find a group of individuals (representing solutions) which have the best fitness according to a requested property (in our case, which best satisfy the cost function stated in Eq. (2)).

In order to use EAs for an optimization problem, the following aspects need to be formulated:

- *Individuals*: An individual represents a possible solution for a considered problem, and a set of individuals constitute a population representing a set of solutions. The idea of EAs is that these populations (and hence solutions) are improved over generations.
- *Mutation operation*: Each individual of a population is subjected to mutations which change the solution each individual represents, and hence explore new parts of the search space.
- *Recombination operation*: Recombinations combine the characteristics of more than one individual, hoping to get the best out of them towards a better solution. Recombinations explore new parts of the search space as well, but also converge on existing individuals.
- *Fitness function*: After each mutation and recombination, new individuals are generated. To decide which individuals shall be considered further, a fitness function selects the best individuals and promising candidates for the next generation.

Overall, the typical flow of EAs starts with the generation of an initial population. Afterwards, a sequence of mutation and recombination operations is conducted which yield new generations of populations. The fitness function selects the individuals for the next generation. This process is continued until the process converges (*i.e.* no real improvements are observed any more) or until a time limit terminates the process.

B. EA-based Verification Runtime Analysis

In the following, we describe how EAs can be utilized for the optimization problem defined above. Recall that we are interested in keeping the set X of variables to be fixed as small as possible while the average verification runtime $\hat{T}_\phi(X)$ remains feasible for the reasoning engine. With this as a basis, we can formulate the different EA aspects with respect to the considered problem as follows:

Individuals: An individual represents a potential solution X as a bit vector $I = \langle I_i \rangle_{i=1,\dots,|\mathit{FV}(\phi)|}$ of size $|\mathit{FV}(\phi)|$ such that for every variable $x_i \in \mathit{FV}(\phi)$ there is a corresponding bit I_i in I which indicates whether $x_i \in X$.

Mutation: Based on the description of an individual, mutations are performed as follows: Given an individual I and a mutation rate p_m, every bit of its vector is flipped with probability p_M. This leads to a new individual J, representing the new solution.

$$m(I, p_m) = \langle J_i \rangle_{i=1,\dots,|\mathit{FV}(\phi)|}$$

$$J_i = \begin{cases} \text{flip } I_i & \text{with } p_m \\ I_i & \text{otherwise} \end{cases}$$

Recombination: Recombinations are performed as follows: Given two individuals I, J and a recombination bias p_c, we combine the two bit vectors by retaining bits which have equal values in both vectors, but randomly choose bits from I or J at positions where they differ. The recombination bias is applied here to prefer one of the individuals. The recombination leads to a new offspring K.

$$r(I, J, p_r) = \langle K_i \rangle_{i=1,\dots,|\mathit{FV}(\phi)|}$$

$$K_i = \begin{cases} I_i, & \text{if } I_i = J_i \\ I_i, & \text{with } p_c \\ J_i, & \text{otherwise} \end{cases}$$

Fitness Function: We employ q from Eq. (2) as the fitness function, with T_ϕ approximated as $T_{max} \cdot 2^{\tau(\phi)}$. $\hat{T}_\phi(X)$ is approximated by averaging the results of a small number of concrete measured times.

Implementation Aspects[6]*:* The *initial population* is obtained by first approximating the threshold $\tau(\phi)$ with a binary search and then instantiating random individuals with $\tau(\phi)$ positive bits. We employ the algorithm with a very low *mutation rate* p_m, since this yields better recombination results. Individuals to *recombine* are randomly chosen using a normal distribution which prefers the best individuals. In addition, we apply a recombination bias p_r towards

[5] However, note that any other optimization methodology can be applied as well, and that the usage of EAs only constitutes a representative.

[6] The implementation is available on GitHub at https://github.com/DFKI-CPS/verification-runtime-analysis.

the individual with the better score. We monitor the progress of the optimization and spawn increasingly many independent individuals as the optimization slows down. In the beginning, these random individuals are of cardinality $|X|$ where X is the best solution found so far. With every generation which does not yield an improvement over the last, we increase the deviation from $|X|$ in order to avoid getting stuck in a local optimum.

Even though the strategies described here constitute only one possible implementation, it yields very promising results, as the experimental evaluations summarized in the next section will show.

V. Experiments and Results

The solution proposed above has been implemented and evaluated using a large corpus of verification instances. This section summarizes the most important results obtained by this evaluation. To this end, we first briefly provide details on the actual set-up as well as the considered benchmarks. Afterwards, the obtained results are presented and discussed.

A. Set-up

As input for the considered problem, we used verification benchmarks provided by the SMT-LIB benchmark library [14] (in the bit vector logic QF_BV), and the SMT solver Z3 [11] as the reasoning engine. The EA has been implemented in the programming language Scala using the Java bindings of Z3.

In order to have a deterministic and reproducible notation of time for the analysis, we used Z3's `rlimit` count as a time unit, which provides the number of elementary operations required to solve an instance. This way, the time measurements (required for the fitness function of the EA) remain independent from the actual platform and hardware. The target time-out T_{max} was set to an `rlimit` count of 500 000 which is roughly equivalent to $0.5s$ of computation on the utilized compute server[7].

Using this set-up, the verification runtime analysis determines the desired set X out of which the variables to be set to fixed values can be obtained. Afterwards, the originally given proposition ϕ as well as the proposition with the variables in X set to ground terms is solved by Z3 again — showing the impact of the obtained analysis results. For the evaluations, solving times have been measured on an Intel Xeon (E3-1270 v3) compute server with 8 cores and 16 GB of memory running Linux.

B. Considered Benchmarks

Our methodology is meant for hard verification tasks which do not terminate before a given time-out. The SMT-LIB library provides a huge, representative corpus of such problems from the verification of circuits and systems. We considered non-iterative quantifier-free bit vector logic (QF_BV) benchmarks from the category "industrial" which are marked as "unsat", where $\tau(\phi)$ (determined by binary search as described above) is larger than 10; the latter ensures that trivial benchmarks which complete in less than roughly $T_{max} \cdot 2^{10} \approx 512s$ are omitted.

With the remaining set of hard benchmarks, the proposed method has been evaluated on a total of 333 propositions. The mean runtime t_A of the analysis was 86 seconds. 34% (114) of the benchmarks were analyzed in under 60 seconds, 93% (309) finished in less than 10 minutes, and the longest took 23 minutes 37 seconds. There was no significant relation between the runtime of the analysis and the original proof time.

C. Obtained Results

Since due to space limitations not all results can be listed and discussed, a representative subset of results is summarized in Table I. Here, the first columns denote the problem size: the number of SMT variables, and the number of bits ($|FV(\phi)|$) representing those SMT variables. The next group of columns shows the results of the analysis: $\tau(\phi)$ is the initially approximated number of variables that has to be fixed, $|X|$ is the size (in bits) of the found solution X; and t_A is the runtime of the analysis itself. The last column group shows the reduction in verification runtime: $T(\phi)$ is the runtime with state-of-the-art verification (which results in a time-out for all problems because we explicitly consider hard ones). $\hat{T}_\phi^\text{rnd}(|X|)$ denotes the runtime when just an arbitrary selection of variables $Y \subset FV(\phi)$ with the same size $|Y| = |X|$ is set to a fixed value, while $\hat{T}_\phi(X)$ denotes the runtime when exactly the variables in X are set to a fixed value.

The results clearly confirm the benefits of our approach. While it is in general not surprising that fixing a number of variables reduces the verification runtime, our analysis yields a small number $|X|$ of variables to fix for maximum effect. By this, verification engineers get much more out of partial verification since it allows them to only set a small portion of the variables to a fixed value. *E.g.* for *calypto/problem_22.smt2*, a naive method would have led them to set $\tau(\phi) = 128$ variables to a fixed value; with the sophisticated analysis method proposed in this work, just fixing $|X| = 13$ is sufficient — yielding substantially larger coverage.

Moreover, the results confirm that not only the number $|X|$ of variables is important (*how many?*), but also which variables should be set to a fixed value (*which?*). This can clearly be seen in the last two columns of Table I: randomly fixing $|X|$ variables often leads to a time-out ($600s$). In contrast, fixing exactly those variables X obtained by the proposed analysis allows solving *all* benchmarks in negligible runtime.

D. Further Discussions

The obtained results show how many and which variables to fix to get as much as possible out of partial verification. In this regard, note that there may be external reasons to fix (or not fix) a variable. For example, it makes no sense to fix sensor input which changes rapidly, but it makes a lot of sense to fix configuration parameters which rarely change (see [8]). Obviously such considerations can easily be integrated into the proposed analysis *e.g.* by adding a *weight* to the variables such that instantiating some variables (which do not change often) is favourable to instantiating others (which do change often).

With regard to related work, the term "partial verification" is also used with model checking, in particular software model checking (see *e.g.* [21], [22]), referring to techniques to reduce the search space in order to find counterexamples (and, hence, bugs), or referring to the exchange of results between different automatic tools (model checkers, static analyzers, theorem provers) such that the combination of partial results makes the whole

[7] Note that there is no exact relation between Z3's `rlimit` count and real time since `rlimit` also considers memory operations.

TABLE I
OBTAINED RESULTS

	Problem Size		Analysis			Verification Runtime Reduction		
Benchmark	SMT Variables	$\|FV(\phi)\|$	$\tau(\phi)$	$\|X\|$	t_{A}	$T(\phi)$	$\hat{T}_{\phi}^{\mathrm{rnd}}(\|X\|)$	$\hat{T}_{\phi}(X)$
...ia32...Add32.load32.Mul32.Mulh_u32.0005.smt2	38	831	43	5	$29s$	$> 3600s^*$	$> 600s^*$	$< 0.01s$
calypto/problem_22.smt2	33	205	128	13	$173s$	$> 3600s^*$	$> 600s^*$	$0.02s$
float/newton.1.3.i.smt2	427	8498	135	33	$92s$	$> 3600s^*$	$> 600s^*$	$0.12s$
float/test_v5_r10_vr10_c1_s7608.smt2	855	17860	91	35	$298s$	$> 3600s^*$	$64.15s$	$0.16s$
float/test_v5_r15_vr5_c1_s23844.smt2	1280	26710	235	48	$492s$	$> 3600s^*$	$301.85s$	$0.23s$
float/test_v7_r12_vr1_c1_s10576.smt2	1431	29853	234	50	$525s$	$> 3600s^*$	$113.98s$	$0.26s$
float/test_v7_r17_vr5_c1_s25451.smt2	2024	42194	157	65	$326s$	$> 3600s^*$	$94.39s$	$0.36s$
mcm/23.smt2	33	363	10	11	$45s$	$> 3600s^*$	$62.04s$	$0.03s$
mcm/63.smt2	36	432	29	12	$49s$	$> 3600s^*$	$155.09s$	$0.04s$
mcm/69.smt2	33	396	12	12	$47s$	$> 3600s^*$	$108.72s$	$0.04s$
tacas07/Y86_std.smt2	246	5795	700	109	$437s$	$> 3600s^*$	$> 600s^*$	$0.07s$
uum/uum16.smt2	190	3428	29	16	$27s$	$> 3600s^*$	$> 600s^*$	$0.01s$
uum/uum20.smt2	234	5244	36	20	$29s$	$> 3600s^*$	$> 600s^*$	$0.02s$

* = time-out

verification succeed (see *e.g.* [23], [24]). This is also referred to as conditional model checking [25]. Furthermore, the term is also used in the context of agents [26], [27], but refers to verification of truthfulness. However, the methodology proposed in this work here is not related to any of these previous work and, hence, is novel to the best of our knowledge.

VI. CONCLUSIONS

In this work, we proposed a novel and complementary approach to tackle the verification gap: instead of aborting the entire verification process and getting no result at all, we propose to set some variables to a fixed value in order to get at least a partial verification result. While the idea itself is rather obvious, we have proposed a systematic verification runtime analysis which show *how many* and *which* variables to fix for maximum verification runtime reduction. Experimental evaluations based on a proof-of-concept implementation confirmed the potential and demonstrated that the proposed analysis method does not only yield a partial verification result, but also gets the most out of it. Considering that further analysis methods can be implemented on top of this methodology, this work provides a promising basis for future work in this direction as an alternative to existing verification methods.

REFERENCES

[1] J. Yuan, C. Pixley, and A. Aziz, *Constraint-Based Verification*. Springer, 2006.

[2] T. Zhang, D. G. Saab, and J. A. Abraham, "Automatic assertion generation for simulation, formal verification and emulation," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE Computer Society, 2017, pp. 471–476.

[3] R. Wille, D. Große, F. Haedicke, and R. Drechsler, "SMT-based stimuli generation in the SystemC verification library," in *Forum on Specification and Design Languages (FDL)*. IEEE, 2009, pp. 1–6.

[4] A. Koczor, L. Matoga, P. Penkala, and A. Pawlak, "Verification approach based on emulation technology," in *International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE, 2016, pp. 169–174.

[5] Y. Kuwabara, T. Yokotani, and H. Mukai, "Hardware emulation of IoT devices and verification of application behavior," in *Asia-Pacific Conference on Communications (APCC)*, 2017, pp. 1–6.

[6] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[7] R. Drechsler, M. Soeken, and R. Wille, "Formal Specification Level: Towards verification-driven design based on natural language processing," in *Forum on Specfication and Design Languages*, 2012, pp. 53–58.

[8] M. Ring, F. Bornebusch, C. Lüth, R. Wille, and R. Drechsler, "Better late than never: Verification of embedded systems after deployment," in *Design, Automation & Test in Europe (DATE)*, 2019.

[9] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing (SAT)*, ser. Lecture Notes in Computer Science (LNCS), vol. 2919. Springer, 2003, pp. 502–518.

[10] A. Biere, "PicoSAT essentials," *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 4, pp. 75–97, 2008.

[11] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science (LNCS), vol. 4963. Springer, 2008.

[12] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science (LNCS), vol. 5505. Springer, 2009.

[13] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler, "SWORD: A SAT like prover using word level information," in *IFIP International Conference on Very Large Scale Integration (IFIP VLSI-SOC)*. IEEE, 2007.

[14] SMT-LIB-benchmarks, "Official SMT-LIB repository," https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks, 2018, accessed: 2018-11-27.

[15] D. B. Fogel, "Evolutionary algorithms in theory and practice," *Complexity*, vol. 2, no. 4, pp. 26–27, 1997.

[16] Z. Michalewicz and M. Schoenauer, "Evolutionary algorithms for constrained parameter optimization problems," *Evolutionary Computation*, vol. 4, no. 1, 1996.

[17] B. Korousic-Seljak, J. Silc, and G. Papa, "An evolutionary approach to problems in electrical engineering design," in *Handbook of Bioinspired Algorithms and Applications*. Chapman and Hall/CRC, 2005.

[18] Z. Vasícek and L. Sekanina, "Evolutionary approach to approximate digital circuits design," *IEEE Transactions on Evolutionary Computation*, vol. 19, 2015.

[19] R. Chen, K. Li, and X. Yao, "Dynamic multiobjectives optimization with a changing number of objectives," *IEEE Transactions on Evolutionary Computation*, vol. 22, pp. 157–171, 2018.

[20] K. Deb, "Multi-objective evolutionary algorithms," in *Handbook of Computational Intelligence*. Springer, 2015.

[21] P. Parizek and F. Plasil, "Partial verification of software components: Heuristics for environment construction," in *Euromicro Conference on Software Engineering and Advanced Applications (EUROMICRO)*. Lübeck, Germany: IEEE Computer Society, Aug 2007, pp. 75–82.

[22] A. Groce and W. Visser, "Heuristics for model checking Java programs," *International Journal on Software Tools for Technology Transfer*, vol. 6, no. 4, pp. 260–276, Aug 2004.

[23] V. Wüstholz, "Partial verification results," Ph.D. dissertation, ETH Zürich, 2015.

[24] D. Beyer, "Partial verification and intermediate results as a solution to combine automatic and interactive verification techniques," in *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, ser. Lecture Notes in Computer Science (LNCS), vol. 9952. Springer, 2016.

[25] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler, "Conditional model checking: A technique to pass information between verifiers," in *International Symposium on the Foundations of Software Engineering*. ACM, 2012, pp. 57:1 – 57:11.

[26] I. Caragiannis, E. Elkind, M. Szegedy, and L. Yu, "Mechanism design: From partial to probabilistic verification," in *ACM Conference on Electronic Commerce*, ser. Conference on Electronic Commerce (EC). ACM, 2012, pp. 266–283.

[27] L. Yu, "Mechanism design with partial verification and revelation principle," *Autonomous Agents and Multi-Agent Systems*, vol. 22, no. 1, pp. 217–223, 2011.