Poster: Efficient Iterative Deepening for Bounded Exhaustive Generation of Complex Structures

Affan Rauf
LUMS, Lahore, Pakistan
14030022@lums.edu.pk

Muhammad Nawaz
LUMS, Lahore, Pakistan
15030025@lums.edu.pk

Junaid Haroon Siddiqui
LUMS, Lahore, Pakistan
junaid.siddiqui@lums.edu.pk

ABSTRACT

The time required to generate valid structurally complex inputs grows exponentially with the input size. It makes it hard to predict, for a given structure, the most feasible input size that is completely explorable within a time budget. Iterative deepening generates inputs of size *n* before those of size *n* + 1 and eliminates guesswork to find such size. We build on Korat algorithm for structural test generation and present *iKorat* – an incremental algorithm for efficient iterative deepening. It avoids redundant work as opposed to naïve Korat-based iterative deepening by using information from smaller sizes, which is kept in a highly compact format.

CCS CONCEPTS

• **Software and its engineering → Software reliability**; **Software testing and debugging**;

KEYWORDS

automated test generation, bounded exhaustive testing, complex structures, incremental test generation, korat, iterative deepening

1 INTRODUCTION AND BACKGROUND

Automated generation of test inputs for programs with structurally complex inputs requires constraints describing the desired tests and technology to solve them. Given a predicate that describes desired structural properties, Korat [1] performs bounded exhaustive search in the predicate's input space and enumerates inputs for which the predicate returns true. Being exponential in nature, a time budget is often used for bounded exhaustive exploration, which requires experimenting with different sizes until a size with enough generated structures and finishing within the given time bound is found. If it finishes too soon, we would like to try a larger size to better utilize the time budget. If it does not, the generated tests are not exhaustive within the set bounds. Thus, we would like to try a smaller size that can be exhaustively covered. Korat implements a backtracking search. So, if stopped before completion, the generated structures of even smaller sizes are not complete, and the generation is not resumable from that point either. A structured way to find the optimal size is *iterative deepening*, which starts from the smallest bound, gradually increases the bound and thus produces results in the order of size. It requires no a priori specification of bound but it results in redundant work as Korat regenerates smaller structures to grow them to form larger structures.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3195002

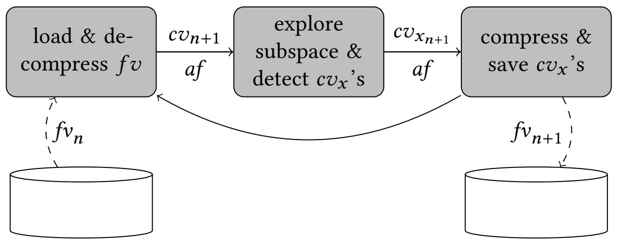

Fig. 1: Architecture diagram of iKorat

We consider Korat and present an incremental technique for efficient generation of complex structures for iterative deepening [3]. Korat represents structures as integer vectors called *candidate vectors* (*cv*). Each index of a *cv* corresponds to a field in the structure and each field is associated with a field domain, which contains all the values that a field can possibly assume. See a BinaryTree represented as *cv* in Fig. 2. All possible combinations of these values permitted by the field domain sizes form a search space, which Korat explores systematically. During exploration it constructs actual structures corresponding to the *cv*'s and invokes a predicate function (`repOk`) on them to see if they are valid instances or not. During validation `repOk` "touches" different fields. These fields are recorded in the order they are touched in accessed-fields vector (*af*). Korat proceeds by substituting the field touched last with the next value from its field domain. In some *cv*'s, doing so will exhaust the field domain. Such vectors can be *extended* to generate structures of larger size. We use them and call them *extendable candidate vectors* (cv_x's). We save all cv_x's found in a particular run and use them as starting points of search space exploration in the next incremental run.

2 TECHNIQUE

iKorat can be broadly divided into the steps illustrated in Fig. 1:

2.1 Load Compressed Vector

Storing both cv_x's and af's results in huge files increasing I/Os and execution time. Hence, we use techniques to reduce file sizes:

(1) *Fused Vectors*: The af vector contains indices of the cv in the order corresponding fields were accessed during `repOk` execution. The cv stores indices of field domains. See Fig. 2. In iKorat, we introduce *fused vector* (*fv*), which is a single vector constructed from these two vectors. It preserves order from af vector and values from cv. See Fig 2. This, however, is a lossy compression. We have lost the information that which fields were actually accessed. We just know the values they were assigned. Nevertheless, we use a novel algorithm to recover the lost information. The algorithm is based

on the insight that a deterministic `repOk` implementation accesses fields of a particular structure in deterministic order. Therefore, the i^{th} field access should correspond to i^{th} value in *fv*. So, we invoke `repOk` on an empty structure while trapping its field accesses. On i^{th} field access we determine the corresponding *cv* position and assign it i^{th} value from *fv*. We update the structure so that the `repOk` execution does not return prematurely. The process is repeated for subsequent field accesses until we have consumed all *fv* values. At the end we have recovered *cv*, af and the actual structure.

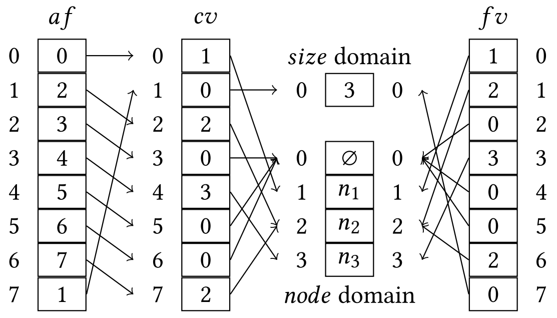

Fig. 2: *cv* representation of a binary tree. The actual mapping among af, cv and fields domain in Korat (left) and the new mapping between fv and field domain in iKorat (right)

(2) *Differential Compression*: Due to lexicographical exploration and backtracking, adjacent vectors frequently have common prefixes of varying length. We store only those values that differ from the preceding vector. We identify different patterns that occur frequently and encode each differently.

(3) *Bit Compression*: As there is a specific bound on the sizes of field domains and the integers we need to save are indices in them, we can just save $\lceil log_2(size) \rceil$ least significant bits.

We start by loading fv's for the preceding size. Then they go through the decompression steps mentioned above.

2.2 Search Space Exploration

Korat explores search space monolithically starting from a single point. iKorat, however, "resumes" exploration from cv_x's (along with corresponding af vectors). We recover *cv* and *af* from the *fv* read from file. Each *cv* serves as a starting point in exploring a subspace within which the exploration continues in the same way as it would in Korat. During the exploration we detect extendable vectors for the next size and save them (§ 2.3). The value at a particular index in the *cv* is incremented during the exploration. If it falls out of the field domain we know that we have run out of elements and cannot continue without adding a new element to the domain, thus increasing structure's *size* by one. This is exactly the kind of vectors we are interested in because in the subsequent incremental execution an extra element *will* be available which we can incorporate into the current structure we have. Therefore, we save such vectors. The *af* gets updated after every iteration. When it becomes a smaller prefix of the one we started with, we stop and conclude that the search in this subspace is over. If there are more *fv*'s in the file, we continue with recovering vectors and exploring other subspaces until we have consumed all *fv*'s.

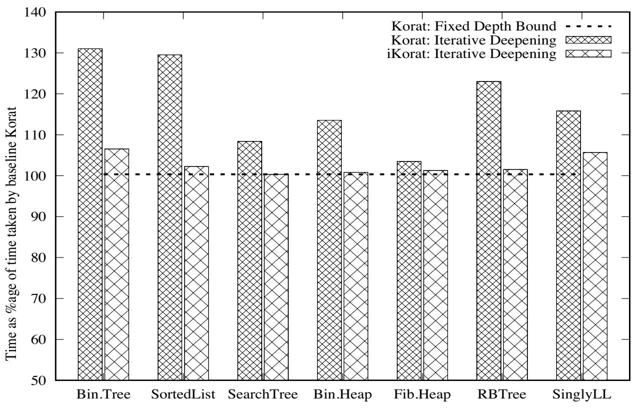

Fig. 3: Relative time of Korat and iKorat based iterative deepening with respect to the time required by baseline Korat

2.3 Save Compressed Extendable Vectors

The cv_x's found go through several compression steps i.e. conversion to fv's, bit compression and differential compression.

3 EVALUATION

The baseline in Fig. 3 represents the scaled time taken by Korat to run for the most feasible size for a given data structure. Of course, it cannot predict that size. It was found using iKorat and then we ran Korat for that size to compare performances. The left and right bars show the time required by Korat-based and iKorat-based iterative deepening, respectively, relative to the baseline. We see the iKorat-based solution is clearly the winner. Also the percentage overhead incurred by iKorat-based iterative deepening over the baseline is always close to the baseline itself. We may conclude that iKorat enables iterative deepening in Korat almost for free.

4 RELATED WORK AND CONCLUSION

The subspaces for which cv_x's act as entry points are similar to the *ranges* which have been used for ranged symbolic execution [4] and ranged model checking [2]. However, they have focused on making the exploration parallel and not on iterative deepening.

We presented iKorat algorithm in this paper to run iterative deepening in a manner that eliminates all redundant work. Experiments showed that iKorat-based iterative deepening performed better than naïve Korat-based iterative deepening and introduced a minimal overhead over baseline Korat for a given fixed size. This was achieved using a novel storage scheme that after losing information recovers it in the next incremental run due to the deterministic nature of Korat exploration.

REFERENCES

[1] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing based on Java Predicates. In *Proc. 2002 International Symposium on Software Testing and Analysis (ISSTA)*. 123–133.

[2] Diego Funes, Junaid Haroon Siddiqui, and Sarfraz Khurshid. 2012. Ranged Model Checking. *SIGSOFT Softw. Eng. Notes* 37, 6 (Nov. 2012), 1–5. https://doi.org/10.1145/2382756.2382799

[3] Richard E. Korf. 1985. Depth-first Iterative-deepening: An Optimal Admissible Tree Search. *Artif. Intell.* 27, 1 (Sept. 1985), 97–109. https://doi.org/10.1016/0004-3702(85)90084-0

[4] Junaid Haroon Siddiqui and Sarfraz Khurshid. 2012. Scaling Symbolic Execution using Ranged Analysis. In *Proc. 27th Annual Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*.