A Graphical Parameter-Based *Finitization* Generator for the Korat Algorithm

T. Williams, C.H.Hopper, M. Stanton

Abstract— We present a novel, Java-based add-on to the Korat exhaustive testing framework. Our utility dynamically determines the structure and sub-structures of a class under test and then presents an interactive visual representation. This representation permits a software tester to graphically constrain the number of member variables created by Korat's *finitization* process. Finally, our extension textually presents the generated valid input structures. We believe our implementation to be novel and beneficial in that it (1) uses the Java reflection API to generate a treeview of the decomposed test structure (including contained sub-structures), (2) allows a tester to visually constrain the number, *nullability,* and value of objects generated and (3) visually depicts the finitization space created in a hierarchical manner. Most significantly, the ability we provide to refine the input parameters to the *finitization* method enables the tester to readily control test scope. In the last section of this paper we use our graphical adaptation to test a data structure similar to that in use at a large financial institution.

I. Introduction

THE Korat algorithm is used to exhaustively, but parsimoniously, generate *all* legal object configurations within a user-defined size for linked complex data structures [1][2]. Korat generates valid test inputs by iterating through a state space of candidates. Only those candidates returning affirmatively from a class invariant method (the *repOK*) are considered valid inputs for use in testing.

Korat is singular in its ability to quickly and exhaustively generate test inputs given a user-specified bound on the size of inputs, *the scope*. Korat conservatively generates test inputs by avoiding the duplication of entities whose members form identical structural graphs (*isomorphs*) but differ only in terms of identity. It does this by imposing a natural partial order on fields, and tracking accesses made thereto by the class invariant. In the following sections of this paper we first address the motivations of our project. Next we discuss our understanding -- and the importance of -- the object field and class domain data structures in Korat. We follow this coverage by addressing the details of our implementation of Korat and our visual finitization editor. Finally, we test our efforts against a real-world data structure used in a national financial services firm.

II. Project Motivations

In consideration of Korat's power and speed, we chose to develop a graphical, Java-based, add-on tool to facilitate a tester's ability to quickly (1) identify through a visual decomposition (*treeview*) the structure and sub-structures of a class to be tested by Korat (2) specify using the very same treeview the number of contained objects in the test structure to create (3) specify a numeric range of values that each field containing primitive data types can assume (4) specify whether contained object(s) can be null and (5) generate a resultant text-based list of Korat-created candidates passing the repOK method. Our primary motivation for constructing this extension was to permit the tester to quickly change those inputs that govern the size of the state space our Korat *finitization* method constructs. Moreover, by providing an interactive visual depiction of the structure under test, we hoped a novice tester might more easily decipher its composition. We textually output valid candidates to aid further the tester's comprehension by means of visual feedback.

III. A SELECTIVE LOOK AT KORAT

As our investigation depended upon our thorough understanding and successful implementation of the Korat algorithm, and given that its source code is not publicly available, we spent considerable time authoring our interpretation of the algorithm. We used the pseudo-code presented in [2] as a starting point for this endeavor. Readers should note that we do not provide a comprehensive analysis of Korat, instead referring readers to the Korat author's texts in [1] and [2] to elucidate completely the algorithm's details.

We began our investigation by stepping through our conceptual interpretation of the algorithm using pencil and paper. After hand-generating more than one hundred candidates in the iterative/backtracking-based fashion described in the paper, we next turned our attentions to understanding the mechanics of creating/initializing and then moving through the state space.

Fundamental to Korat is the concept of an "`Object Field`." There is an `ObjectField` for each member variable in the test structure and the sub-structures it contains. The state space from which valid candidates emerge is a mapping of each and every `ObjectField` to an array of `ClassDomainIndex`(es). Korat-generated candidates passed to the repOK method are partially ordered streams of `ObjectFields`, wherein each position represents a particular `ObjectField`, and the value thereat is an indirect pointer (*that is, an index to an index to a value*) to a single object from one or more `ClassDomains`.

To understand how to generate class domains we began by deconstructing the data structure `ClassDomainIndex`. A `ClassDomainIndex` is, in essence, a mechanism through which to reach a particular instance of a class in a particular class domain. We discovered that it is a quasi-repetitive structure wherein each member of a class domain carries an identical copy of the domain objects specific to a particular domain. Differentiation occurs between class domain indexes with identical class domains because every class domain index has a successive numerical index into the class domain. This arrangement imposes a lexicographical ordering within each of the domains. In the case of the binary search tree we used in development (*a 3-node binary search tree*), the class domain(s) for the *Root* `ObjectField` consisted of {*null*}, {*Node0*, *Node1*, *Node2*}. Null always forms a domain by itself. As an example of this configuration, we present a graphical representation of the class domain indexes for the field tree::root in Figure 1.

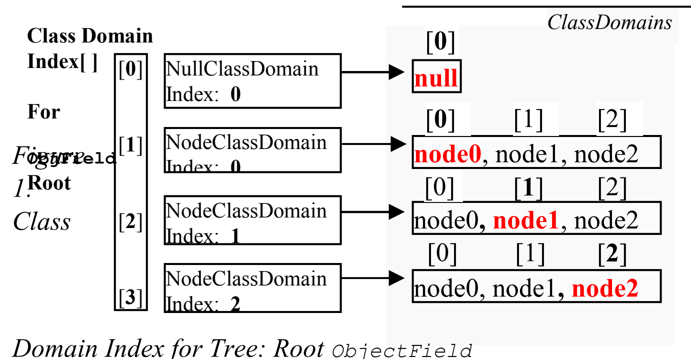

Figure 1. Class Domain Index for Tree: Root `ObjectField`

After understanding how to map object fields to class domains, we next turned our attention to implementing Korat in code.

IV. IMPLEMENTING KORAT AND THE VISUAL FINITIZATION EDITOR

We chose to author our implementation of Korat and the visual finitization construction editor in Sun Microsystems Java J2SE 5.0 (*a.k.a. 1.5.)* We chose this version because of its inherent support for generic data types. The visual portion of our project we authored also in 5.0, using both the JFaces and SWT toolkits from the Eclipse Project. We selected these libraries because their performance is vastly superior to that of Java JFC/Swing in our target environment, Windows. Interestingly, we encountered a fare amount of difficulty in our attempts to make our treeview nodes editable. Direct support for editing nodes in the JFace API proved so troublesome that it became a non-trivial task to force this functionality by merging SWT routines with the JFaces library.

Our first coding responsibility entailed the creation of a mechanism that takes as input a user-specified class name. It then investigates it to discern its underlying structure. In our code, we employ extensively the Java Reflection API to walk the structure of the class under test, creating as we go a list of the sub-structures contained therein. As our

program encounters each new sub-structure, the algorithm spans into that new type and begins decomposing it. Upon completion of this nested crawl, control returns to the parent structure. If a previously encountered type reappears, our program knows not to re-visit it since there already exists an entry for it in our type catalogue.

The results of this reflective exploration of the test structure populate the left pane of our GUI. The discovered structure is rendered as a collapsible treeview [*see Appendix A*]. The outermost level displays all of the *data types* discovered during the crawl. In the case of our binary search tree this includes: the binary search tree class, tree size definition (*string*), node class, node info definition, node description definition (*string*), and the node comparable definition. For each encountered data type, the tester sees the class name. In the case of types that describe fields, the tester sees a short name that is composed of the object's name and the name of the field. Data types that are numeric in nature present min/max value boxes, into which the tester may specify the minimum and maximum values of objects to be create created. A cardinality property exists for non-numeric data types. The tester can set the cardinality to control the number of objects to be created.

Descending inward from the first node in the treeview (in this particular case, the binary search tree class), the user views each *field* discovered in the test class. Under each field box there appears the field's name and type. Furthermore, there is a box indicating whether or not the field contains a primitive data type, of which there are eight in the Java language. Finally, the tester has the option to choose whether the field permits null as a valid value. Toggling on the null entry enables the tester to expand his state space to include checking for nulls. If such checking is not deemed necessary, then he can generate a smaller scope by disabling the inclusion of a null class domain for the selected field's class domain index array. These user specifications will be consumed in the next phase by the `ObjectFieldFactory` to constrain the creation of each `ObjectField` and the objects in the class domains they point to.

Clicking on the **Build** button initiates a three-step process. In Phase 1, we look at every object type appearing in the far left level of the treeview and determine the class domains to be built. The tester-defined cardinality values constrain the number of objects to create for the appropriate class domain. So, in the case of Node::Comparable, if the tester sets cardinality to 3, then our implementation knows to create three objects of the Comparable interface type. Note that we will only instantiate interfaces for which our crawler was able to find a concrete implementation. If no such concrete implementation is discovered, we simply set the class domain to null. In the case of primitives, such as *int*, we create objects using the appropriate object wrappers. Actual values for these wrapper types come from the range specified by the user in the GUI. To avoid class domain collisions (that is, when there are two or more *unrelated* fields that contain the same datatype), we append the fieldname to the class domain. Therefore, in the event that we were to have a structure with two unrelated *int* fields, we'd know that the values for each field came from different, discrete `ClassDomains`. We treat strings in a similar fashion – mapping each string field to a unique `ClassDomain` of strings.

In Phase 2 of the build process, we create an `ObjectField` for each and every field described by an `ObjectFieldFactory`. In the case of the tree object, we know to create two `ObjectFieldFactor(ies)` one for root and one for size. In the case of Nodes, we know to create Node0Left, Node0Right, Node1Left, Node1Right, Node2Left, Node2Right, etc. We name our fields by concatenating the class name, a sequential number and the field name. Hence, as seen above, we derive Node0Right.

In Phase 3, we use the metadata defined in the `ObjectFieldFactory` to determine whether a null class domain should be added to the existing `Class Domains` for each object field. Combining the information gleaned in Phase 1, we then create our `ClassDomainIndexes` mapping each and every `ObjectField` to a set of class domain indexes. We add the class domain indexes one at a time, starting with the null class domain index (if appropriate) and then adding the object class domains, using as our index the order of object creation. In the case of the Node0Left object field, we map it to one NullClassDomainIndex with an index of 0 and then three NodeClassDomainIndex(es) with indexes of 0

through 2. The result of this process is a finitization space. This is two-dimensional mapping of object fields to class domain indexes. We draw it in hierarchical fashion in the right hand pane of the GUI. In essence, this pane is a structured realization of the finitization creation effort. The outermost level of the tree depicts each and every object field we created. Underneath each object field the user sees every class domain index to which it is mapped. The class domain index is depicted in the following fashion: the pointed-to object is represented using its Java *.toString()* method and its numerical index into the class domain is shown as an integer. This visualization shows clearly the space to be fed to the koratSearch method. We believe this illustration to be a novel and useful aid to enhance the tester's comprehension of the *finitization* space constructed.

After generating the finitization space, we feed it to the koratSearch method. To successfully run the algorithm we adapted the pseudo-code in [2]. As it was imperative to track fields accessed during firing of the repOK method, we modified our repOK method to use accessor methods in our binary search tree implementation. Our accessor methods popped a copy of the object field on to the stack in the event it is not already there. Although using accessor methods was not our preferred implementation choice, we discovered that the only way to instrument Field accesses without using accessors and mutators was to modify the byte code or to use some form of proxy framework. Both of these options were rejected due to time constraints.

In order to ensure that the entire data structure under test is considered when checking for isomorphs, all reachable object fields must be added to the stack regardless of whether they were used to determine if the class invariants were satisfied. If the candidate is determined to be valid (or if pruning of the state space is disabled), the data structure is traversed to each basis field; any fields that are not already in the stack are added.

The bottom pane in the GUI presents the tester with a list of all valid candidate structures meeting the conditions of the class invariant. In the case of the binary search tree we implemented, the results pane displays all valid trees using the toString() representation of each object for each of the object fields. With isomorphism breaking turned on (*it is turned on by default*) and tree size value of exactly three, our results pane correctly display five non-isomorphic structures. In the case where we request all valid structures ranging from size 0 to size 3, the results pane displays 15 valid candidates.

V. An Analysis from the Financial services Domain

One of our central objectives in implementing Korat is to apply it to a 'real-world' data structure. We have therefore defined a simplified version of a loan application, similar to that in use at a large financial institution. This class consists of an id, a set of applicants, a state (either complete or incomplete), a status (either pre-submitted, submitted, decisioned, or fulfilled), and an approved amount ($0.00 for declined/undecisioned applications, otherwise the amount of the loan).

For the purposes of this exercise, external checks on the status of activities such as underwriting, sending a decline letter, or extending a loan offer we represent with the Boolean flags isApproved and isFulfilled. The services which compare the internal state and status of the loan application with other business systems are more elaborate, but these flags serve to illustrate how the class invariants for a loan application reflect the business rules that define what the correct values of an application may be at different points in its lifecycle.

The repOk() method for the LoanApplication class is seen below in *Figure 2*:

```
//enforce the class invariants of LoanApplication
   private boolean repOk(){
      if (state == LoanApplicationState.PreSubmitted){
         return true;
      }
      else if (status == LoanApplicationStatus.Incomplete){
         //only complete applications may be submitted
         return false;
      }

      //for decisioned loans - approval requires a loan amount
      //decline requires a $0.00 amount
      if (state == LoanApplicationState.Decisioned){
         if (isApproved){
            if(approvedAmount.getAmount() <= 0.00){
               return false;
            }
         }
         else{
            if (approvedAmount.getAmount() ==  0.00){
               return false;
            }
```

```
        }
    }

    if (state == LoanApplicationState.Fulfilled){
        if (!isFulfilled){
            return false;
        }
    }

    return true;
}
```

Figure 2: repOK for LoanApplication

The repOk() method shows that before an application is in a submitted state there are no constraints on the valid values of the fields. At this stage of processing, applications may have missing or incomplete data elements. Moreover, other fields may have initial values that are potentially illegal later on. Note that for any subsequent status, it is illegal for an application to be incomplete; if a decision has been made, the approved amount must agree with the decision. Declined loans must have an approved amount of $0.00, and approved loans must have some positive loan value. Finally, if an application is marked as fulfilled, the isFulfilled indicator must agree that fulfillment activities are complete.

VI. Conclusion

We have shown that Korat can be a powerful tool in the validation of data structures. The quality of the predicates and the finitization are critical factors in the success of the Korat-driven testing effort, however; and our interactive GUI finitization technique added to the Korat toolkit provides a significant enhancement to the finitization creation process.

This business domain example serves to illustrate a point from [1] that if a well designed object already contains a mechanism for enforcing the class invariants, like a repOk() method, then the generation of valid candidates with the Korat algorithm requires little additional effort on the part of the programmer. If no such enforcement mechanism has been implemented, there is a substantial amount of work in preparing the predicate method so that valid candidates can be identified.

While this may be a trivial exercise for a well defined abstract data type like a tree or a queue, for a business domain object, it is likely that the class invariants are poorly defined – if they are defined at all. This is not to say that this makes Korat an undue burden on the programmer; much rather, Korat becomes useful when the business requirements are understood well enough to define what the class invariants are (as well as the preconditions and postconditions for the methods on the class). The effort required to generate tests with Korat is essential in determining what to test in the first place.

It is often the case in building a large system that all of the details of the specification are not known up front. This is where the power of our GUI Finitization Editor becomes clear. A programmer can begin with a somewhat useful finitization generated from a weak class invariant and begin to refine it over time to become more useful and more well defined. Ultimately, the construction of better predicates can be driven by the types of objects the programmer selects in the interactive finitization process; then the well-defined predicates can be employed to better enforce the intended behavior of the class. This serves to illustrate how GUI-driven finitization improves the utility of Korat as an iterative development tool.

VII. REFERENCES

[1] C. Boyapati, S. Khurshid and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 123-133, July 2002.
[2] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid and M. Rinard. *An Evaluation of Exhaustive Testing for Data Structures*. Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, September 2003.
[3] R. Harris and R, R. Warner. *The Definitive Guide to SWT and JFace*. Apress. 2004.
[4] I. Forman and N. Forman. *Java Reflection in Action.* Manning Publications. 2004.

APPENDIX A: VISUAL FINITIZATION EDITOR

VizFin ver. 1.0

Reflect on...

net.toddgwilliams.vv.datastructure.Tree Reflect

net.toddgwilliams.vv.datastructure.Tree
Name: net.toddgwilliams.vv.datastructure.Tree
Type: class net.toddgwilliams.vv.datastructure.Tree
Cardinality
Tree::root
Tree::size
FieldName: size
FieldType: int
IsPrimitive: true
FieldNullable: false
false
Node::info
Name: Node::info
Type: int
Min Value
Max Value
Tree::size
Name: Tree::size
Type: int
Min Value
Max Value
Node::comparable
Name: Node::comparable
Type: interface java.lang.Comparable
Cardinality
net.toddgwilliams.vv.datastructure.Node
Name: net.toddgwilliams.vv.datastructure.Node
Type: class net.toddgwilliams.vv.datastructure.Node

ObjectFieldFactories

Finitization Space

Build

Treesize0
Class Domain: Tree::size
1
Index: 0
Class Domain: Tree::size
2
Index: 1
Class Domain: Tree::size
3
Index: 2
Nodedescription1
Class Domain: Node::description
String0
Index: 0
Class Domain: Node::description
String1
Index: 1
Class Domain: Node::description
String2
Index: 2
Nodeleft1
Class Domain: NULL
null
Index: 0
Class Domain: Node
net.toddgwilliams.vv.datastructure.Node@cac268
Index: 0
Class Domain: Node

ClassDomainIndexes for TreeSize field

ClassDomainIndexes for Node1Left with Null ClassDomainIndex at head

Korat

Find Solutions

Results Here

Data types that form the class domains

ObjectFields