Automated Data Structure Generation: Refuting Common Wisdom

Kyle Dewey Lawton Nichols Ben Hardekopf
University of California, Santa Barbara
{kyledewey, lawtonnichols, benh}@cs.ucsb.edu

***Abstract*—Common wisdom in the automated data structure generation community states that declarative techniques have better usability than imperative techniques, while imperative techniques have better performance. We show that this reasoning is fundamentally flawed: if we go to the declarative limit and employ constraint logic programming (CLP), the CLP data structure generation has orders of magnitude better performance than comparable imperative techniques. Conversely, we observe and argue that when it comes to realistically complex data structures and properties, the CLP specifications become more obscure, indirect, and difficult to implement and understand than their imperative counterparts. We empirically evaluate three competing generation techniques, CLP, Korat, and UDITA, to validate these observations on more complex and interesting data structures than any prior work in this area. We explain why these observations are true, and discuss possible techniques for attaining the best of both worlds.**

I. Introduction

Automated data structure generation helps to test code that interacts with or operates on data structures. Techniques for automated data structure generation have been studied for over a decade, from TestEra [1] to Korat [2], ASTGen [3], and UDITA [4]. TestEra was based on the declarative language Alloy [5] (essentially an interface to a lower-level SAT solver); its declarative nature made simple data structures easy to specify, but its performance was disappointing. Succeeding techniques are based on the imperative language Java, explicitly in order to gain better performance. These latter techniques also extend Java semantics in various ways to include declarative features such as nondeterminism, explicitly in order to preserve the usability[1] benefits of declarative languages. This progression of research has led to the common wisdom in the automated generation community that *imperative implies better performance* and that *declarative implies better usability*.[2]

In this paper, we demonstrate that the common wisdom is flawed at best and backwards at worst—we show that declarative techniques can be much faster than imperative ones, and argue that imperative techniques can have much better usability than declarative ones. We arrive at this conclusion in two steps. Recall that the current state of the art (Korat and UDITA) modify the Java language semantics to include various declarative features. We argue first that extrapolating these features to their logical conclusion leads us to *constraint logic programming* (CLP), which can be used to subsume and extend the current state of the art in data structure generation. We show that an off-the-shelf CLP engine allows for declarative specification and generation of data structures that easily outperforms existing techniques such as Korat and UDITA. CLP is an inherently declarative approach, and thus we show that declarative techniques have superior performance to imperative ones, and we explain why this fact is true.

The current state of the art data structure generation techniques have been evaluated on relatively simple data structures such as sorted linked lists and red-black trees. For these data structures the CLP specification is concise and elegant, being at least on par with specifications in Korat and UDITA in terms of usability. Thus, when looking at the kinds of data structures that have historically been used to evaluate automated generation, CLP is the clear winner. However, our work goes beyond these simple data structures to encompass more complex structures and properties. For example, we specify and generate red-black trees with the additional property that inserting a given element guarantees the tree must be rebalanced. This property takes into account not just the invariants that define the data structure, but also models the imperative operations on that data structure (i.e., insertion and rebalancing). We also specify and generate more complex data structures such as skip lists, splay trees, and B-trees; these structures have cycles, probabilistic properties, and other features that make them more challenging than the simpler structures that previous work has evaluated. Based on our subjective experience, we observe that for these kinds of structures and properties the CLP specifications become complex and cumbersome; they are no longer concise and elegant and closely follow the definition of the data structure, but instead are obscure and indirect and difficult to define, understand, and debug. We support this observation via metrics which act as proxies for complexity, and argue why this declarative difficulty arises.

These two insights lead us to the conclusion stated earlier: generating data structures using CLP (an extreme point on the declarative side of the declarative/imperative spectrum) is much faster than using techniques such as Korat and UDITA that are based on Java and that use its imperative features; however in our experience Korat and UDITA are generally

This work was supported by NSF CCF-1319060.

[1]"Usability" refers to the ease of use and learnability of a human-made object, in this case how easily data structures can be specified.

[2]The exact meanings of the terms "imperative" and "declarative" are a constant source of debate, though we define "imperative" to mean the presence of mutable state and loops, and "declarative" to mean the lack of these features.

more usable than CLP for specifying complex data structures and properties. This insight, and the reasoning behind it, leads to some interesting ideas about future work that may eventually allow us to combine the benefits of both approaches. In summary, the contributions of this paper are:

- We describe the progression of past research on automated data structure generation and its tension between declarative and imperative techniques. (Section II)
- We argue that the logical extrapolation of the trends contained in that research is constraint logic programming, and explore why CLP can be a good fit for this problem for simple data structures. (Section III)
- We examine more complex properties that data structures might have and explain why CLP can be lacking in terms of usability when dealing with these properties. (Section IV)
- We describe seven specific data structures that exhibit a range of the above properties: three structures that were evaluated in previous work and four that have never been automatically generated before this work. Also, we describe for each data structure an additional property that yields a particularly interesting part of the search space for that data structure. (Section V)
- We specify and generate all of the data structures described above using CLP, Korat, and UDITA, objectively comparing these approaches in terms of performance and subjectively comparing them in terms of usability. (Section VI)
- We use our insights into the advantages and disadvantages of the different generation strategies to describe some possible interesting avenues for future work in combining their benefits. (Section VII)

II. Related Work

We focus on general black-box data structure generation techniques, as opposed to more specialized white-box techniques (e.g., [6], [7], [8], [9]). We observe that prior work in the space of black-box data structure generation has been moving between the imperative and declarative realms for over a decade in an attempt to find a "sweet spot" that makes the proper tradeoffs. We present the relevant prior work in chronological order in order to make this phenomenon clear.

TestEra [1] is the starting point for related work and represents a highly declarative mindset. TestEra allows users to specify data structures (and operations on said data structures) via invariants encoded in Alloy [5], a specification language intended for bounded model checking. Alloy translates these specifications into a SAT instance which is ultimately passed to a SAT solver. With TestEra, satisfying SAT assignments correspond to data structures which are valid according to the provided specification.

While TestEra allows for concise and precise definitions, it has two major drawbacks: (1) it is disconnected from downstream testing; and (2) it has poor and unpredictable performance. "Disconnected" means that the data structures for automated testing are specified and generated in a language which is entirely separate from what is actually being tested. That is, while we may be interested in testing a red-black tree implementation in Java, we must specify it in Alloy first and then translate the resulting structure into a Java datatype. In terms of performance, SAT is an NP-complete problem and SAT solvers are ill-suited for generating *many* satisfying solutions, as opposed to a single satisfying solution. In addition, it can be challenging to diagnose performance issues surrounding a SAT solver [10].

A notable step after TestEra is Korat [2], which addresses the disconnect and performance issues with TestEra. Korat allows users to write predicates directly in Java, represented as pure functions that take a data structure and return a boolean value; these functions are written to return `true` iff the provided structure meets the necessary invariants. Korat uses these predicates effectively in reverse to *generate* interesting data structures. The predicates themselves use imperative features such as loops and assignment, but the overall specification strategy (i.e., providing predicates that accept interesting structures and using those predicates to generate said structures) is declarative in nature. Assuming the downstream program under test is in Java, Korat's strategy allows for data structures to be passed directly to the test harness without any sort of translation. Additionally, Korat sees much better performance than TestEra does, with experiments typically running several orders of magnitude faster.

While Korat addresses the major drawbacks of TestEra, a slew of problems remain which were inherited from TestEra. Most important of these to our discussion is the fact that Korat's search strategy is fixed and outside of control of the user. This means that users cannot adjust the order in which data structures are enumerated, which is a desirable property to have [11].

In response specifically to the search strategy issues, ASTGen [3] was developed. ASTGen is aimed towards data structure generation problems which are best phrased in terms of how construction should occur, as opposed to what is considered interesting. This is a radical departure from prior work, since it fundamentally promotes an imperative approach to data structure generation as opposed to a declarative approach. The authors argue that certain problems, such as enumerating grammars, are well-suited to this approach. Most importantly, the authors make strong claims about performance, best illustrated through the following quote: "The imperative approach makes the generation faster since no search is necessary."

To allow for imperative generation, ASTGen offers an iterator-like interface in Java, where the key methods are:

- `hasNext`: Does the generator have more elements.
- `next`: Get the next available element.
- `reset`: Reset the generator to the beginning.

Via the above interface, users can directly generate data of interest with explicit control over the order in which such data is produced. The downside of this interface is that it makes operations painfully explicit, and generating any sort of nested data structure requires carefully constructing multiple iterator classes and passing instances along to where they are needed.

While ASTGen theoretically overcomes the issues surrounding Korat's search strategy, the authors do not demonstrate any sort of utility for this ability. More importantly, while performance is frequently used to justify the clear loss of usability, ASTGen is never evaluated against Korat (or any other prior techniques). As such, ASTGen's apparent performance benefits are never demonstrated, so it is unclear if they exist at all.

In response to the severe loss of usability in ASTGen, UDITA [4] was developed, which attempts to recover some of the usability of Korat in an ASTGen-style framework. A key observation made in the UDITA work is that the sort of imperative generators used in ASTGen can be seen as a means of nondeterministically enumerating a stream, done in a very explicit way. Building from this insight, the authors simplify the generator interface in ASTGen down to a single method `generate()`, which returns a single data structure in a nondeterministic fashion. This is made possible by executing the generators on Java PathFinder [12], a special Java Virtual Machine which imparts a nondeterministic semantics to Java. An additional benefit of using Java PathFinder is that users can take advantage of `assume` statements (pedantically, `ignoreIf` statements), which semantically allow execution to proceed only if a given Boolean condition evaluates to `true`. Via `assume`, programmers can generate data structures of interest given only a predicate that tests validity, as with Korat. In this way, UDITA harmonizes the imperative generators of ASTGen with the declarative style of Korat.

In addition to its usability benefits, UDITA saw performance benefits as well. UDITA was around 3× faster than ASTGen on large benchmarks, though around 30× slower on smaller benchmarks that ASTGen completed in under 3 seconds. This performance benefit was unexpected considering the fact that UDITA diverges from the imperative model, which was assumed in ASTGen to be unquestionably high-performance. Our own evaluation in Section VI shows that UDITA is substantially slower than Korat, and thus transitively this means ASTGen is even slower still; this observation implies that much the the work done after Korat has actually *regressed* the state of the art.

While UDITA addresses the sort of usability issues seen in ASTGen, and even sees performance improvements, this all comes at a substantial cost: significantly increased complexity, both from the user's and the implementer's standpoint. Users must understand both ASTGen and Java PathFinder just to pick up UDITA, and they must also be comfortable with nondeterministic execution. As for the implementation, in addition to the use of Java PathFinder, the authors found it necessary to implement some key optimizations such as lazy instantiation of data structures in order to make generation feasible. For these reasons, UDITA is is unappealing for data structure generation; there are simply too many moving parts, and the barrier to entry is high.

The most recent related work is Senni et al. [13], who propose using CLP as an alternative to Korat and perform a preliminary evaluation that shows CLP's promise. This work is not in the mainstream, but it anticipates our argument for CLP to a certain extent. However, Senni et al. are primarily interested in research on source-to-source Prolog optimizations rather than examining CLP for automated data structure generation. Thus, their paper does not attempt to explore the connections between prior work (such as Korat and UDITA) and CLP, it does not fully explore the utility of CLP and its capabilities, and it leaves a number of issues unaddressed that previous work has shown are important. First, they do not investigate CLP's ability to customize search strategies, which was a major impetus for ASTGen and which is a strength of CLP that Senni et al. do not exploit at all. Second, they trivialize the problem of being disconnected from downstream testing (something that CLP shares with TestEra), which was a major impetus for Korat. Finally, Senni et al. only evaluate a few low-complexity data structures with small bounds, all of which were seen in prior work; these structures do little to fully showcase and exercise the abilities of CLP. Because they did not push the abilities of CLP, they did not observe any of the shortcomings of CLP that we enumerate in this paper and do not have our central insight that refutes the common wisdom about data structure generation.

III. Advantage of CLP: Performance

Constraint logic programming (CLP) languages have features such as built-in nondeterministic search, value unification, and arithmetic constraint solvers. We argue that not only are these features useful for data structure generation, but that many of the features in prior work such as Korat, ASTGen, and UDITA are ad-hoc approximations of a subset of these features. Because CLP engines have been heavily optimized over decades to make these features efficient (e.g., [14], [15], [16], [17]), CLP has a natural performance advantage over the prior work. In addition, the prior work requires extensive changes and/or additions to the Java semantics, making their implementation complex; CLP can take advantage of off-the-shelf engines without any modification.

As an example of how CLP can be used for data structure generation, consider a sorted linked list. Here is the CLP program that can generate an infinite number of such structures:

```
sorted([]).
sorted([_]).
sorted([A,B|Rest]) :- A #=< B, sorted([B|Rest]).
:- sorted(L), write(L), fail.
```

This example uses standard Prolog syntax, where clauses are delimited with periods, `:-` can be understood as reverse implication, comma stands for logical conjunction, and labels starting with capital letters are logical variables. There are three clauses followed by a query; the first clause states that an empty list is a sorted list; the second states that a single-element list is a sorted list; the third states that a multi-element list is sorted if the first two elements in are ascending order and the rest of the list is sorted. Execution of this program starts at the query in the last line. The semantics of CLP dictate that the expression `sorted(L)` will find some structure that satisfies the predicate `sorted` and bind it to the variable `L` using unification. It will then write that structure to output and finally fail. Failure automatically triggers backtracking for nondeterministic search, and so the engine will backtrack to the `sorted(L)` expression and find a different satisfying structure, bind it to `L`, write it to output, and fail again. This process will continue indefinitely, outputting an infinite stream of satisfying structures.

We now go over the various features of CLP that are useful for data structure generation and compare them with features that are present in prior work.

Nondeterministic Search. From a high level, all prior work can be seen as techniques to nondeterministically generate data structures of interest in a given space. In practice, they all employ various form of backtracking algorithms. In TestEra [1], ultimately the nondeterministic generation is done by SAT solvers, which use backtracking algorithms for search [10]. Unlike CLP, however, traditional SAT solvers (e.g, those described in [10]) are ill-suited for generating *many* satisfying solutions, necessitating modifications to the core solving algorithm for the sake of efficiency [18], [19]. In Korat [2] a backtracking algorithm is added on top of the JVM and used to search the space of all structures to find ones that match a predicate defined by the user. UDITA [4] actually modifies the semantics of Java using Java PathFinder [12], which uses backtracking to make Java execution nondeterministic. These techniques all build nondeterministic search into the infrastructure, hiding it from the user; ASTGen [3], in contrast, forces the user to explicitly encode the nondeterministic search into the specification of the data structure being generated.

Nondeterministic execution is a core feature of CLP semantics, and has been discussed in the literature since very early on [20]. As such, there have been literally decades of work on making this feature efficient in CLP engines. The declarative nature of CLP (i.e., without assignments or mutation) helps to make these engines as efficient as they are; nondeterminism in an imperative setting as in the prior work is both less efficient and more complex.

Search Strategy Control. All prior work has employed bounded-exhaustive generation, i.e., defining a finite space and generating all structures within that space. While bounded-exhaustive search has merit [21], [22], there are other search strategies that can be useful. Random search, iterative deepening, and various hybrid approaches have been used in the past to good effect for other types of automated generation [11], [23]. As such, restricting search to a single strategy is overly limiting.

ASTGen and UDITA allow for some coarse-grained control over the order of data structure generation, but CLP can easily exceed this low bar. CLP naturally employs a depth-first search strategy, and the order in which structures are generated can be adjusted by modifying the order of clause definitions. Bounded-exhaustive search can be achieved by common built-in routines such as SWI-Prolog's [24] `call_with_depth_limit/3` function, which adds a user-defined recursion bound to the search. Random testing is possible again via common built-in routines such as `maybe(P)`, where `1 - P` is the probability that the `maybe` expression will fail and trigger backtracking. All of these strategies are composable with each other; for example, we can easily define an iteratively deepened random search within some maximum bound. This ability to vary the search strategy with such ease is a major advantage of CLP for data structure generation compared to related work.

Equality Constraint Propagation. Senni et al. [13] observe that UDITA's [4] lazy data structure instantiation optimization "can be seen as a particular CP (constraint propagation) solution strategy". We observe that this UDITA optimization in fact behaves just like the logical variables available in typical CLP engines, which allow for the propagation of *equality constraints*. Semantically, logical variables start in a special *uninstantiated* state, wherein the variable has no specific value. Uninstantiated variables can be aliased with each other, essentially putting variables into the same equivalence class. Logical variables can later become *instantiated* with particular values, and all aliased variables will automatically have that same value. To better illustrate this phenomenon, consider the following code: `X = Y, Y = 1`. This code aliases the logical variables `X` and `Y` with the expression `X = Y`, then sets both of them to the value `1` with the expression `Y = 1`. This behavior bears striking similarity to the lazy instantiation optimization in UDITA, which: (1) only instantiates variables when operations specific to a given data structure are performed on them, and (2) allows for uninstantiated variables to be aliased. In this way, UDITA is attempting to emulate the logical variables already available in CLP engines, though in an ad-hoc manner.

Disequality Constraint Propagation. In Korat [2], blind search is avoided by observing what sort of sub-structure caused a data structure to be rejected by the user-defined predicate. This information is retained in a way that prevents further data structures with identical sub-structure from being generated. We observe that, in effect, this strategy allows Korat to propagate *disequality constraints*, which prevent the generation of invalid sub-structures. In UDITA [4], certain optimizations related to isomorphism-breaking are implemented in a manner which is similar to disequality constraint propagation. This observation is made directly by the authors in demonstrating the correctness of the details of their generation algorithm.

While disequality constraints are somewhat non-standard in CLP languages, it is still compatible with CLP [25]. We do not make use of this fact in our evaluation, but our results show that CLP's standard equality constraints significantly outperform Korat's disequality constraints.

Arithmetic Constraint Propagation. The hallmark of CLP engines is the ability to reason about symbolic arithmetic via high-performance arithmetic constraint solvers. While all of the prior work allows for generation of data structures with arithmetic invariants, with the exception of Senni et al. [13] this capability is handled via a generate-and-filter approach. That is, instead of asking a constraint solver to deliver numbers which satisfy some given arithmetic constaints, one must instead try all numbers in a range and filter out those which did not satisfy applicable arithmetic invariants. Not only is this inefficient, it forces the data structure generator to reason about data structure shape and contents simultaneously, which can be problematic. For example, consider the problem of generating sorted lists of length 0 to N. In general, there are only $N+1$ unique list shapes in this space, though a potentially infinite number of list structures when content is taken into account. If the tester desires to test only structures with particular shapes, without regard to contents, the space is quite small. However, the need for contents can blow up the search space in a completely uninteresting direction. With CLP, it is possible to reason about shape and contents independently and to request only a single satisfying solution for a list of any given length. With the prior techniques, it would be necessary to tweak various bounds in an ad-hoc manner just to get a single solution, and this sort of tweaking does not scale to arbitrary data structures.

IV. Disadvantage of CLP: Usability

For simple data structures such as singly-linked lists and binary search trees, CLP can specify the appropriate structure invariants simply and concisely. In fact, the specifications tend to mirror the invariants closely in a very elegant way, as with the sorted list example in Section III. The prior work in data structure generation has only looked at these sorts of simple structures, and thus CLP seems like a clear winner for both performance and usability. However, when we extend our evaluation to more complex structures and properties, we see that while CLP still has much better performance than prior work, its usability suffers in manner that has not been noted before in the literature. We elaborate on these issues below.

Cycles and Node Sharing. Consider the problem of generating a tree data structure with `parent` pointers, imparting a type of cycle in the data structure. Such cyclical structures cannot be directly specified in CLP. The problem is that the fundamental unit of data structure creation in CLP (known simply as a *structure*) does not permit cycles. The solution is to employ indirection, e.g., to label nodes with identifiers and maintain a mapping from identifiers to nodes, and to use those identifiers to describe the data structure's shape rather than the actual nodes themselves. This strategy adds additional complexity to the specification and obscures the connection to the data structure invariants being specified. If any part of the structure may contain a cycle, then indirection must be used for the entire data structure specification.

To illustrate this problem, consider the simple update of a `parent` pointer. Ideally, to update node `n1`'s parent to be node `n2` we need only write `n1.parent := n2`, which corresponds closely to the expected implementation. However, with CLP, we instead need to write something similar to the following (in pseudocode):

```
removeEdgesAnnotatedWith(
    edges, getNode(nodes, "n1"), "parent"),
addEdgeAnnotatedWith(
    edges, getNode(nodes, "n1"),
    getNode(nodes, "n2"), "parent").
```

where `getNode`, `removeEdgeAnnotatedWith`, and `addEdgeAnnotatedWith` perform map lookups and updates to observe and change the underlying representation of the graph. This style is clearly much more verbose than the original, making CLP a poor choice for the representation of cyclical data structures.

A similar problem holds for data structures that are DAG-like, i.e., where a single sub-structure may be referenced via multiple paths in the data structure. CLP naturally tends to divide its generation into independent sub-structures in a bottom-up manner. CLP will recursively build a series of independent sub-structures, then bind them together into the overall structure. This strategy does not work if those sub-structures are not independent, e.g., if they need to all refer to the same elements. In this case, the CLP specification must explicitly build the to-be-shared elements and explicitly pass them down through all of the recursive calls so that each sub-structure will be referencing the same elements. Again, this adds complexity and obscures the connection to the data structure invariants being specified.

Imperative Operations. CLP does not offer imperative-style `for` and `while` loops, which is inconvenient when it comes to representing imperative operations which use these features. More importantly, while most CLP engines do allow for limited forms of imperative reassignment, these operations behave in a manner which is incompatible with data structure generation. That is, while these operations can be easily used to verify if a given data structure satisfies a property, they cannot be used to generate such satisfying data structures. Attempts to do so will silently produce incorrect results or otherwise bizarre behavior, ultimately because these operations break the otherwise logical semantic model of CLP. This issue forces the user to rewrite such operations in a functional manner, often using a *store-passing* style [26] that adds yet another layer of indirection. This extra indirection is fraught with the same sort of issues as described previously with cycles, and is highly undesirable from a usability standpoint.

The need for data structure specifications that model imperative operations is novel to this work, and it is a direct consequence of our modeling of advanced properties on complex data structures.

Metaprogramming. Most CLP engines support metaprogramming through the `call` instruction, which structure values to behave as calls to clauses (i.e., code). In many ways, `call` is comparable to the notorious `eval` construct in languages like JavaScript [27], as it fundamentally allows for dynamically-generated code to be executed. Metaprogramming can be used to parameterize computations and minimize redundant code, which we have found very useful in specifying the data structures and properties described in Section V. However, it is an error-prone technique with substantial performance drawbacks. In fact, because of performance concerns it is common practice in the CLP community to write macros which expand code using metaprogramming into code that does not use metaprogramming, which imparts complexity [28].

In general, it is desirable to avoid metaprogramming both from a correctness and performance perspective, but we have found in our own specifications that it is almost impossible to avoid metaprogramming without significantly bloating the specifications. We have thus reluctantly come to the conclusion that metaprogramming is an unfortunate must-have.

Downstream Testing. With CLP, assuming the downstream code we want to test is not itself in CLP, there is a disconnect between the generated structures and the appropriate datatype in the language of the application being tested. This means that some sort of translation layer between the data structures in CLP and the data structures in the system under test must be in place, which adds complexity and could potentially harm the performance gains afforded by CLP. This issue was one of the motivating factors behind Korat [2], and is of importance for any practical data structure generation technique.

V. Data Structures and Properties

In this section we describe the seven data structures on which we evaluate the competing data structure generation techniques. In addition to the baseline data structure definitions, we also describe for each data structure an additional property that targets an interesting part of the space of such structures; these additional properties stress both the performance and usability of the competing generation techniques. Three of the structures have been evaluated in prior work, and are included here for comparison: sorted linked lists, red-black trees, and heaps. Four of the structures have never been evaluated for generation before this work: image grammars, skip lists, splay trees, and B-trees. The additional properties for all of the structures, including the three structures seen in prior work, are novel to this work.

Here we informally describe the structures and properties. The exact predicates that we use to specify them are available in the supplementary materials[3], including the Korat, UDITA, and CLP specifications.

Sorted Linked Lists. A sorted linked list is a linked list whose nodes are ordered according to their contents, in this case integers. Both UDITA [4] and Senni et al. [13] generated these data structures.

Additional Property: We target lists where each integer element is separated by at most a value of k. For example, a valid list for $k = 3$ would be `[0, 2, 5, 5]`.

Red-Black Trees. A red-black tree is a type of balanced binary search tree that is commonly used as an efficient representation for sets and maps. Korat [2], UDITA [4], and Senni et al. [13] all showed that they could generate red-black trees with varying degrees of success.

Additional Property: We target red-black trees such that inserting a given element is guaranteed to cause rebalancing. Intuitively, this means that given an element value, we generate red-black trees such that if the given element is inserted into the tree it will cause a rebalance to occur. This property requires us to encode the insertion and rebalancing operations in our predicate that describes acceptable red-black trees.

Heaps. A heap is a type of balanced binary tree that is commonly used to represent priority queues efficiently. While heaps are usually described as trees, they are often backed by arrays in imperative settings and thisis the underlying representation that we use as well. As with red-black trees, Korat [2], UDITA [4], and Senni et al. [13] all generated these structures.

Additional Property: We target heaps which require exactly $\log_2 n$ operations on `dequeue`, where n is the number of nodes in the heap. Such data structures show worst-case behavior, and are interesting not only for testing but for benchmarking. This property requires encoding the `dequeue` operation in the predicate describing acceptable heaps.

Image Grammars. Many data representations can be explained in terms of grammars. ASTGen [3] heavily focused on grammars, and claimed that declarative approaches were generally ill-suited to generating structures that obey grammars. While there is prior work on generating programs from context-free grammars [29], we instead focus on the context-sensitive grammar of the ANI image format [30]. Bugs in parsers for this grammar have been historically costly [31]. We observe that the ANI grammar is not well-documented, and that there are several edge cases where it is unclear if a parser should accept or reject a given image. For our standard definition of ANI images we avoid these edge cases.

Additional Property: We target specifically those edge cases that we avoid in the standard definition. In other words, we target ANI images that are guaranteed to contain at least one edge case. These edge cases are:

- A `Rate` subsection is named "`LIST`", which introduces a parsing ambiguity with another image component with the same name.
- An `InfoList` subsection has size 2, which should not be possible with valid data.
- The `title` or `author` field holds a non-printable character.
- The image contains no icons (indicated with an icon length of 0), which are core components of the image.
- The `jifRate` is 0, which corresponds to an animation that would move infinitely fast.

Skip Lists. A skip list [32] is a special DAG-like representation of a linked list that allows for multiple elements in a list to be traversed in a single operation. The consequence of this on peformance is that inserting an element into a sorted linked

[3]http://www.cs.ucsb.edu/~pllab/ under the `Downloads` link.

list can be performed in $\mathcal{O}(\log n)$, unlike the typical $\mathcal{O}(n)$. Of special interest is that these data structures rely on probabilistic features and thus do not have deterministic shapes.

Additional Property: We target skip lists where fewer than $k\%$ of the elements have the maximum height. The observation this property is based on is that the smaller this percentage becomes, the less likely the data structure is in practice (due to the probabilistic features of the skip list algorithm), and thus we are more likely to generate what can be considered an edge case. Ideally we would like a very small percentage, though this percentage also influences the number of elements in the tree. To keep list sizes manageable, we use $k = 25$.

Splay Trees. Splay trees [33] are a type of binary search tree, which are known for their imperative implementation. The value of these trees is in their ability to reconfigure themselves so that, from an intuitive standpoint, elements which are freqently accessed are cheaper to access than others. Central to this reconfiguration is a `splay` operation that moves a given element to the root of the tree via a series of modifications.

Additional Property: Due to the `splay` operation, the shape of a splay tree can vary widely between different operations which call `splay`. For testing, we are interested in particularly dramatic changes to the tree's shape. Specifically, we want to generate trees for which the following two properties hold in conjunction, where N is the total number of nodes in the tree:

- The tree contains at least one node at depth greater than $\lceil 1.5 \times \log_2 (N) \rceil$.
- If a `splay` operation is performed on any single node in the tree, all nodes in the tree would have depth $\leq \lceil 1.5 \times \log_2 (N) \rceil$.

The aforementioned properties define splay trees which can become more balanced via some particular use of `splay`. Generating such splay trees would be useful for testing any optimization scheme based on this observation.

B-Trees. B-trees [34], [35] are a complex tree-based data structure which are used heavily in databases and filesystems. Given the fact that these are so popular at base system levels, it is useful to be able to generate these automatically for testing purposes.

Additional Property: We take a similar approach as with red-black trees, generating trees which would experience node-splitting given some particular value to insert.

VI. Evaluation

We evaluate and compare Korat, UDITA, and CLP for performance, scalability, and usability. The overall goal of our evaluation is to back our claims that CLP is a high-performance technique, but one which can be unwieldy to use, especially with respect to imperative operations.

A. Experimental Methodology

We have specified basic versions of the seven data structures described in Section V and also advanced versions containing the additional properties, in each of Korat, UDITA, and CLP. To be concise, we uniformly refer to these 14 versions as "data structures". The basic data structure is referred to as "basic", and the version of the data structure with the additional property is referred to as "special".

To measure performance we record the time each technique takes to generate all structures within a given set of bounding values. While evaluations in prior work report bounds as a single uniform value n, we observe that this does not reflect reality for even the simplest of data structures. That is, for all the data structures involved, there are multiple distinct bounding values that must be specified. Therefore, we report all of the bounding values used for each data structure. A description of these bounding values is provided in Table I. Henceforth we will refer to these bounding values via comma-separated lists of integers, where the integer's position reflects which bound is being referred to in Table I and the integer value is the actual bound. For example, with basic sorted lists the bounds "2, 3" would mean a maximum of two nodes, and a maximum element value of three. Additionally, we set $k = 3$ for special sorted lists (see Section V) and we ensure that we insert an element distinct from the tree contents for special red-black trees and special B-trees. To measure scalability, we break the performance results into three separate groupings based on small, medium, and large bounding values. For CLP, we chose GNU Prolog [36], [17] as our engine due to its public availability and high performance.

As a proxy for usability, we record the approximate amount of time needed to specify the given data structure for an author already familiar with the particular generator language being used. In cases where code was already provided by the authors of Korat and UDITA, we consider this to have taken 0 minutes. Where possible we used publicly available code for our specifications. In all cases, we specify the additional property for a given data structure after the data structure itself was specified, which often helped to reduce the amount of time necessary to specify the additional property.

Secondary to the amount of time needed to specify the data structures are various measures which act as proxies for code complexity. For each of the techniques we record lines of code (LOC). We record the maximum number of variables ever in scope at once, which gives an idea of the amount of state manipulation and passing that is required to specify a given data structure, and we argue is correlated with the amount of state the programmer must reason about. For Korat and UDITA we record the number of conditionals (treating loops as conditionals), along with the maximum nesting depth for conditionals. The observation here is that conditionals, especially deeply nested conditionals, signify complex control flow. This measure does not have a direct translation in CLP, which is why it is only recorded for the Java-based Korat and UDITA. For CLP, we measure the maximum call-graph strongly-connected component (SCC) size after implications are refactored into clauses, which provides a measure of how many levels deep mutual recursion becomes. For example, a maximum SCC size of four means that there are four clauses that contain mutually recursive references to each other. Mutual recursion indicates complex control flow and a high cognitive burden on the programmer. We also measure the number of metacalls performed in CLP, specifically `call` and `maplist` instructions, so chosen because these tend to be unintuitive and again translate to high cognitive overhead. The last metric we report is the number of static assignments used in the Korat and UDITA versions. While this is not strictly a measure of code complexity, it is a potentially useful predictor of complexity in the CLP version because the assignment operation is fundamentally unavailable in CLP and must be worked around.

B. Performance Results and Discussion

Performance data for Korat, UDITA, and CLP generators for small, medium, and large bounds are shown in Tables II, III, and IV, respectively. There are several key points to make with this data, which are brought out in order of increasing bounds.

Poor UDITA Performance. As shown in Table II, even with small bounds UDITA often takes orders of magnitude more time than Korat. According to the prose found in the literature [3], [4], UDITA is faster than ASTGen which itself is implied to be much faster than Korat. We initially thought that our setup must be in some way malformed, but upon further examination this observation turns out to be consistent with results reported separately for Korat [2] and UDITA [4]. For example, while Korat and UDITA both evaluate on heap arrays of length 8 in the two works cited above, UDITA is a full order of magnitude slower than the same result in Korat, despite the fact that UDITA was introduced nearly 8 years after Korat. UDITA was never directly evaluated against Korat, though it was evaluated against its predecessor ASTGen, which is implied to be faster than Korat (though this is never evaluated empirically). Given that UDITA is generally faster than ASTGen, we conclude that the imperative-style generation used in ASTGen and UDITA has significantly worse performance than declarative-style generation using CLP.

Korat on B-Trees. Given the above results, an unexpected datapoint in Table II is that Korat takes orders of magnitude more time than UDITA with basic B-trees, though its performance improves by orders of magnitude when considering special B-trees. This performance is due to the fact that Korat must generate arrays all at once—Korat cannot piece arrays together incrementally, in contrast to UDITA. Our B-tree specification relies heavily on arrays and constraints on arrays, and so Korat ends up generating many unsatisfiable arrays in relation to UDITA. Korat on special B-trees is much faster because there is only one satisfying structure in the space. In other words, the vast majority of the structures in this space are invalid—since Korat learns from negative information, it very quickly learns the fact that the space is nearly entirely unsatisfiable, and is able to skip most of that space without having to explore it.

Excellent CLP Performance. For all data structures and for all bounds, CLP outperforms Korat and UDITA, usually by orders of magnitude. This is best shown in Table IV, which features large bounds for these data structures. As shown, with large bounds, Korat and UDITA timeout on all data structures, whereas CLP does not timeout on anything.

TABLE II: Performance data for small bounds, in seconds.

Data Structure	Bounds	Korat	UDITA	CLP
Basic Sorted Lists	6, 6	0.266	898	0.001
Special Sorted Lists	6, 6	0.226	898	0.001
Basic Red-Black Trees	10, 10	45.8	322	0.3
Special Red-Black Trees	10, 10	24.7	327	0.168
Basic Heaps	8, 8	5.6	335	0.96
Special Heaps	8, 8	4.8	347	0.036
Basic Images	2, 1, 1, 1, 2	12.9	1027	2.8
Special Images	1, 1, 1, 1, 1	41	1611	7
Basic Skip Lists	4, 3, 3	67.4	1461	0.001
Special Skip Lists	4, 3, 3	58.4	1467	0.001
Basic Splay Trees	5, 5	1.86	66.8	0.001
Special Splay Trees	4, 4	0.386	81.8	0.001
Basic B-Trees	2, 2	140	1.05	0.001
Special B-Trees	2, 2	3.64	1	0.001

There are several reasons why CLP offers such good performance. With CLP we have more direct control over the search strategy. For example, consider B-trees, which have an invariant that all leaves must be on the same level. With Korat, the best we can do is assert that this is true and hope that the Korat engine learns the pattern. With CLP, we directly constrain the generation so that all leaves are on the same level by construction, and so that they are only put at positions where they could legally be with respect to the number of nodes in the tree. This ends up cutting down dramatically on the amount of unsatisfiable search space.

A second major reason for CLP's performance is the presence of an arithmetic constraint solver. As an example, consider the performance results for the additional property on red-black trees in Table III. While the property intuitively requires a generate-and-filter approach, we can actually do better with CLP. CLP allows us to impose symbolic arithmetic constraints on the data structure which force rebalancing to occur with respect to a given key, even without knowing the concrete values in the tree. This means that once the constraints are imposed, it is a simple matter of enumerating all integers which satisfy the symbolic constraints, which is a relatively low-cost operation. In contrast, with both Korat and UDITA, we are forced to take a generate-and-filter approach.

A third reason for the performance gains is that with CLP, we are implicitly utilizing decades of research into making nondeterministic search and arithmetic constraint solvers fast. We get these benefits "for free" just using the existing GNU Prolog engine.

C. Usability Results and Discussion

We compare our proxy usability metrics for Korat and CLP in Tables V and VI, respectively. UDITA results are comparable to Korat and omitted for space. First, we observe that it took less time and code to specify the image grammars for CLP (Table VI) than it did for Korat (Table V). While CLP introduces many more variables in scope, most of these are used in trivial ways. With this in mind, we argue that CLP is well-suited to specifying grammars. This is contrary to statements made by ASTGen regarding the usability of declarative

TABLE I: DESCRIPTION OF BOUNDS FOR ALL DATA STRUCTURES UNDER TEST. IN ALL CASES, THE MINIMUM ELEMENT VALUE IS 0.

Data Structure	Bound 1	Bound 2	Bound 3	Bound 4	Bound 5
Sorted Lists	Max # of Elements	Max Element Value	—	—	—
Red-Black Trees	Max # of Internal Nodes	Max Node Value	—	—	—
Heaps	Max # of Internal Nodes	Max Node Value	—	—	—
Images	Max # of Icons	Max Title Length	Max Author Length	Max `cSteps`	Max `jifRate`
Skip Lists	Max Height	Max # Elements	Max Element Value	—	—
Splay Trees	Max # of Internal Nodes	Max Element Value	—	—	—
B-Trees	Max # of Nodes	Max Element Value	—	—	—

TABLE III: PERFORMANCE DATA FOR MEDIUM BOUNDS. "—" SIGNIFIES TIMEOUT AFTER 1800 SECONDS (30 MINUTES).

Data Structure	Bounds	Korat	UDITA	CLP
Basic Sorted Lists	12, 13	62.5	—	1.61
Special Sorted Lists	12, 13	51.5	—	0.38
Basic Red-Black Trees	12, 12	1218	—	1.26
Special Red-Black Trees	12, 12	709	—	0.804
Basic Heaps	9, 9	55.1	—	6.32
Special Heaps	9, 9	45.6	—	1.18
Basic Images	2, 1, 1, 2, 2	908	—	37
Special Images	2, 1, 1, 2, 2	—	—	279
Basic Skip Lists	4, 4, 4	—	—	0.001
Special Skip Lists	4, 4, 4	—	—	0.001
Basic Splay Trees	6, 6	96.7	—	0.016
Special Splay Trees	6, 6	361	—	0.004
Basic B-Trees	4, 4	—	—	0.001
Special B-Trees	4, 4	—	—	0.001

TABLE IV: PERFORMANCE DATA FOR LARGE BOUNDS. "—" SIGNIFIES TIMEOUT AFTER 1800 SECONDS (30 MINUTES).

Data Structure	Bounds	Korat	UDITA	CLP
Basic Sorted Lists	17, 17	—	—	569
Special Sorted Lists	18, 18	—	—	734
Basic Red-Black Trees	18, 18	—	—	189
Special Red-Black Trees	20, 20	—	—	560
Basic Heaps	11, 11	—	—	873
Special Heaps	12, 12	—	—	937
Basic Images	20, 1, 1, 2, 2	—	—	691
Special Images	10, 1, 1, 2, 2	—	—	1792
Basic Skip Lists	9, 9, 9	—	—	1136
Special Skip Lists	9, 9, 9	—	—	810
Basic Splay Trees	11, 11	—	—	487
Special Splay Trees	12, 12	—	—	380
Basic B-Trees	10, 20	—	—	67.5
Special B-Trees	20, 20	—	—	102

techniques for grammar generators, though it is supported by prior work on automated program generation [29].

Comparing Korat to CLP shows major differences with respect to our data. The specification time for CLP is generally much higher than for Korat, exemplified best by B-trees. One explanatory hypothesis is that the more assignment statements there are, the more cognitive overhead is required to translate to CLP. We argue that this is a poor predictor, as evidenced by the fact that the number of assignments in Table V does not correlate well to the specification times in Table VI. The lack of correlation is explained by the observation that certain assignments are more difficult to translate than others. For example, consider special splay trees, which use the imperative-style `rotateLeft` operation shown below:

```
private Node rotateLeft(Node h) {
  Node x = h.right;
  h.right = x.left;
  x.left = h;
  return x;
}
```

The translation of this code into CLP yields the following:

```
rotateLeft(tree(B, AElem, tree(F, CElem, G)),
           tree(tree(B, AElem, F), CElem, G)).
```

While the CLP version is quite short, it took a full 30 minutes to derive and looks markedly different from the original imperative definition of the `rotateLeft` operation. This leads us to argue that the number of assignments is a poor predictor of specification time.

We argue that ultimately the reason why the CLP specification time is so high is because the data structures under consideration generally have only imperative descriptions. For Korat, third-party code implementing these structures with their various operations is publicly available, making it possible to simply borrow existing implementations. In contrast, with CLP, not only is code unavailable, existing imperative code is so different semantically that it cannot be translated directly. Instead, it is absolutely required to have a deep understanding of the data structure and the operations at hand, so that these can be emulated in a declarative way. In short, with CLP we must have in-depth knowledge of exactly what is being generated, and we do not have the luxury of being able to borrow code or even translate pseudocode.

The proxy metrics for code complexity tell a similar story. In terms of LOC, CLP appears to be the clear winner. However, as shown via the max SCC size in Table VI, not only is recursion required for all data structures, mutual recursion (with all its added complexity and cognitive burden) is often needed. This observation holds true especially for those structures that are novel to this work, e.g., an SCC size of 10 for special splay trees and an SCC size of 8 for special heaps. A similar, though less strong, pattern is seen with the number of variables, especially with the special B-trees. All the data structures involve metacalls, which impede understanding. For these reasons, we argue that CLP easily leads to code which is higher in complexity than corresponding imperative code, contrary to the assertions of Senni et al. [13].

D. Threats to Validity

In cases where existing data structure generation code was not already available from the authors of Korat [2] and UDITA [4], we had to write our own generators. The performance of these tools can be sensitive to the coding style used, so it is conceivable that our generators for these data structures could be further optimized. With respect to our usability and complexity results, a full user study has not been performed and these results are specific to the authors and the coding style used. As such, these values are not necessarily representative

TABLE V: Usability Data for Korat. Times to specify are approximate, in minutes.

Data Structure	Time to Specify	LOC	# Assignments	# Conditionals / Max Depth	Max # Variables In Scope
Basic Sorted Lists	0	68	3	12 / 2	5
Special Sorted Lists	20	92	6	17 / 2	7
Basic Red-Black Trees	0	174	1	33 / 3	10
Special Red-Black Trees	30	296	29	49 / 3	10
Basic Heaps	0	42	2	8 / 3	5
Special Heaps	40	77	11	14 / 3	13
Basic Images	90	185	4	10 / 2	7
Special Images	40	230	10	21 / 2	8
Basic Skip Lists	90	79	4	16 / 3	9
Special Skip Lists	30	86	4	16 / 3	9
Basic Splay Trees	60	93	3	14 / 3	10
Special Splay Trees	40	191	21	34 / 3	11
Basic B-Trees	60	113	5	19 / 4	7
Special B-Trees	40	155	12	24 / 3	7

TABLE VI: Usability Data for CLP. Times to specify are approximate, in minutes.

Data Structure	Time to Specify	LOC	Max SCC Size	# Metacalls	Max # Variables In Scope
Basic Sorted Lists	5	13	1	1	4
Special Sorted Lists	5	26	1	1	5
Basic Red-Black Trees	60	33	1	2	14
Special Red-Black Trees	120	67	1	2	14
Basic Heaps	20	43	2	1	7
Special Heaps	180	103	8	1	11
Basic Images	60	149	1	7	19
Special Images	30	165	1	7	19
Basic Skip Lists	240	54	4	2	7
Special Skip Lists	20	77	4	5	7
Basic Splay Trees	15	31	1	2	10
Special Splay Trees	360	101	10	2	11
Basic B-Trees	600	85	3	4	11
Special B-Trees	480	217	7	5	22

of a typical professional programmer. Additionally, our code complexity metrics serve only as easily-measurable proxies for actual code complexity, which is arguably more subjective.

VII. Discussion

We have shown that CLP is, by far, the best-performing technique for automated data structure generation, with Korat and UDITA trailing behind significantly. However, we argue that CLP is also less usable than Korat and UDITA, and that CLP can lead to more complex code. Ideally, we would like a single solution that gets the best of both worlds. In this section we argue *against* certain solutions that might seem intuitively appealing, and *for* one particular solution that we feel would be a fruitful avenue for future work.

The observation underlying all of the possible solutions is that we need to have imperative elements (i.e., loops and assignment) in the data structure specification language to maximize usability, yet still take advantage of CLP's superior performance. Modifying the CLP language itself to include these features is a non-starter, as these do not compose well with existing CLP optimization work. Similarly, attempting to put CLP features into an imperative language is undesirable, as evidenced by UDITA's poor performance. One seemingly appealing idea is to compile existing Korat specifications written in Java down to CLP, though this is problematic. For one, the mere translation from Java to CLP is a difficult problem (e.g., [37], [38]). Moreover, Korat assumes a completely different execution model than CLP, so direct translations would be inherently inefficient and likely non-terminating. Korat predicates must explicitly check for acyclicity in the data structure being specified, while acyclicity is guaranteed by CLP under certain common constraints, so directly translating Korat predicates to CLP would yield unnecessary code. Conversely, since Korat bounds the state space upfront, Korat predicates need not concern themselves with bounds. CLP predicates, in contrast, need bounds to ensure termination.

Bearing all this in mind, we believe that the most promising strategy is to design a small, custom imperative-style domain-specific language (DSL) for data structure specification that compiles down to CLP. The language's execution model would be similar to that of CLP, with loops and assignment translated to recursion and store-passing style, respectively. The language would be designed from the ground up to avoid the sort of issues seen in translating Korat predicates directly to CLP.

VIII. Conclusions and Future Work

We have shown that for data structure generation the common wisdom regarding the performance benefits of imperative techniques is incorrect, with declarative techniques being orders of magnitude faster than existing imperative techniques. Conversely, we have argued that the commonly-heralded usability benefits of declarative techniques are exaggerated when it comes to more complex data structures unexamined in prior work, and with these data structures imperative techniques seem to be more usable. Fundamentally, the lack of imperative features in CLP makes it unattractive for implementing many data structures. Building from these insights, for future work we plan to implement an external DSL that offers the imperative features desirable for usability, but compiles to CLP for performance.

References

[1] D. Marinov and S. Khurshid, "Testera: A novel framework for automated testing of java programs," in *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ser. ASE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 22–. [Online]. Available: http://dl.acm.org/citation.cfm?id=872023.872551

[2] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on java predicates," in *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '02. New York, NY, USA: ACM, 2002, pp. 123–133. [Online]. Available: http://doi.acm.org/10.1145/566172.566191

[3] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 185–194. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287651

[4] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in udita," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 225–234. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806835

[5] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, Apr. 2002. [Online]. Available: http://doi.acm.org/10.1145/505145.505149

[6] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750

[7] N. Tillmann and J. De Halleux, "Pex: white box test generation for .net," in *Proceedings of the 2nd international conference on Tests and proofs*, ser. TAP'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 134–153. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792786.1792798

[8] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[9] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[10] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008. [Online]. Available: http://books.google.com/books?id=anJsH3Dq5BIC

[11] V. Jagannath, Y. Y. Lee, B. Daniel, and D. Marinov, "Reducing the costs of bounded-exhaustive testing," in *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ser. FASE '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 171–185. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00593-0_12

[12] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engg.*, vol. 10, no. 2, pp. 203–232, Apr. 2003. [Online]. Available: http://dx.doi.org/10.1023/A:1022920129859

[13] V. Senni and F. Fioravanti, "Generation of test data structures using constraint logic programming," in *Proceedings of the 6th international conference on Tests and Proofs*, ser. TAP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 115–131. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30473-6_10

[14] D. H. D. Warren, "An abstract prolog instruction set," AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Tech. Rep. 309, Oct 1983.

[15] H. Nässén, M. Carlsson, and K. Sagonas, "Instruction merging and specialization in the sicstus prolog virtual machine," in *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ser. PPDP '01. New York, NY, USA: ACM, 2001, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/773184.773191

[16] P. Van Roy and A. M. Despain, "High-performance logic programming with the aquarius prolog compiler," *Computer*, vol. 25, no. 1, pp. 54–68, Jan. 1992. [Online]. Available: http://dx.doi.org/10.1109/2.108055

[17] D. Diaz and P. Codognet, "The gnu prolog system and its implementation," in *Proceedings of the 2000 ACM Symposium on Applied Computing - Volume 2*, ser. SAC '00. New York, NY, USA: ACM, 2000, pp. 728–732. [Online]. Available: http://doi.acm.org/10.1145/338407.338553

[18] J. Hooker, "Solving the incremental satisfiability problem," *The Journal of Logic Programming*, vol. 15, no. 12, pp. 177 – 186, 1993. [Online]. Available: http://www.sciencedirect.com/science/article/pii/074310669390018C

[19] J. Whittemore, J. Kim, and K. Sakallah, "Satire: A new incremental satisfiability engine," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. New York, NY, USA: ACM, 2001, pp. 542–545. [Online]. Available: http://doi.acm.org/10.1145/378239.379019

[20] D. H. D. Warren, L. M. Pereira, and F. Pereira, "Prolog - the language and its implementation compared with lisp," *SIGART Bull.*, no. 64, pp. 109–115, Aug. 1977. [Online]. Available: http://doi.acm.org/10.1145/872736.806939

[21] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson, "Software assurance by bounded exhaustive testing," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 133–142. [Online]. Available: http://doi.acm.org/10.1145/1007512.1007531

[22] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard, "An evaluation of exhaustive testing for data structures," MIT Computer Science and Artificial Intelligence Laboratory Report MIT -LCS-TR-921, Tech. Rep., 2003.

[23] J. Ruderman, "Introducing jsfunfuzz," 2007. [Online]. Available: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/

[24] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[25] E. J. G. Arias, J. M. Carballo, and J. M. R. Poza, "A proposal for disequality constraints in curry," *Electronic Notes in Theoretical Computer Science*, vol. 177, no. 0, pp. 269 – 285, 2007, proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming (WFLP 2006). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066107002265

[26] S. Krishnamurthi, "Store-passing style," 2002. [Online]. Available: http://cs.brown.edu/courses/cs173/2003/Textbook/2003-10-10.pdf

[27] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do: A large-scale study of the use of eval in javascript applications," in *Proceedings of the 25th European Conference on Object-oriented Programming*, ser. ECOOP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 52–78. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032497.2032503

[28] R. A. O'Keefe, *The Craft of Prolog*. Cambridge, MA, USA: MIT Press, 1990.

[29] K. Dewey, J. Roesch, and B. Hardekopf, "Language fuzzing using constraint logic programming," in *Proceedings of the 29th IEEE International Conference on Automated Software Engineering*, ser. ASE '14, 2014.

[30] J. Houghtaling, "Windows 95 animated cursor format." [Online]. Available: http://www.wotsit.org/download.asp?f=ani&sc=463191359

[31] M. Christakis and P. Godefroid, "Proving memory safety of the ani windows image parser using compositional exhaustive testing," Tech. Rep. MSR-TR-2013-120, November 2013. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=204915

[32] W. Pugh, "Concurrent maintenance of skip lists," College Park, MD, USA, Tech. Rep., 1990.

[33] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, no. 3, pp. 652–686, Jul. 1985. [Online]. Available: http://doi.acm.org/10.1145/3828.3835

[34] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIGFIDET '70. New York, NY, USA: ACM, 1970, pp. 107–141. [Online]. Available: http://doi.acm.org/10.1145/1734663.1734671

[35] R. Bayer, “Binary b-trees for virtual memory,” in *Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIGFIDET ’71. New York, NY, USA: ACM, 1971, pp. 219–235. [Online]. Available: http://doi.acm.org/10.1145/1734714.1734731

[36] “The gnu prolog website.” [Online]. Available: http://www.gprolog.org/

[37] C. Flanagan, “Automatic software model checking via constraint logic,” *Science of Computer Programming*, vol. 50, no. 13, pp. 253 – 270, 2004, 12th European Symposium on Programming (ESOP 2003). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642304000073

[38] M. Gómez-zamalloa, E. Albert, and G. Puebla, “Test case generation for object-oriented imperative languages in clp*,” *Theory Pract. Log. Program.*, vol. 10, no. 4-6, pp. 659–674, Jul. 2010. [Online]. Available: http://dx.doi.org/10.1017/S1471068410000347