An Automated Approach for Writing Alloy Specifications Using Instances

Sarfraz Khurshid Muhammad Zubair Malik Engin Uzuncaova
Software Verification, Validation and Testing Group
The University of Texas at Austin
Austin, TX 78712
Email: {khurshid,mzmalik,uzuncaov}@ece.utexas.edu

***Abstract*— We present aDeryaft, a novel technique for automating the writing of specifications in Alloy—a first-order relational logic with transitive closure. Alloy is particularly suitable for specifying structural properties of software, and has steadily been gaining popularity due to the rapid feedback that its SAT-based analyzer provides fully automatically. Alloy users however, still have to manually write specifications in a *declarative* language and use a paradigm that is different from the commonly used imperative programming paradigm.**

aDeryaft assists Alloy users in writing their specifications by providing a novel specification-writing approach, which is particularly tailored to users, such as engineers or practitioners in industry, who may not have much prior experience or proficiency in Alloy or similar logics. The user constructs by hand a few small concrete instances that represent the constraints of the software structure they want to specify. aDeryaft then fully automatically generates an executable Alloy specification, which represents the constraints that summarize the given structures. The generated specification is fully executable. For example, the Alloy Analyzer can use it to systematically enumerate a large number of concrete instances that satisfy the same constraints as the given instances. Indeed, a user may use aDeryaft to generate partial specifications while the user writes the rest of the specification by hand.

To efficiently generate Alloy specifications, aDeryaft exploits the relational basis of Alloy and formulates graph properties that are likely to hold for the given instances. It then checks the properties for these instances and translates the valid properties into Alloy constraints, which it outputs as an Alloy specification. We illustrate aDeryaft's constraint generation on a variety of commonly used data structures.

I. Introduction

As software systems steadily grow in complexity and size, designing such systems manually becomes more and more error-prone. The last few years have seen a new generation of lightweight design tools that allow formulating designs formally, as well as checking their correctness to detect crucial flaws that, if not corrected, could lead to massive failures. The Alloy tool-set is one such design tool that is rapidly gaining prominence [1], [2]. The user formulates their design in the Alloy language, which is a first-order logic (with transitive closure) based on relations, and checks the correctness properties using the Alloy Analyzer.

The Alloy tool-set has been used successfully to check designs of various applications, such as Microsoft's Common Object Modeling interface for interprocess communication [3], the Intentional Naming System for resource discovery in mobile networks [4], and avionics systems [5], as well as designs of cancer therapy machines [6].

The Alloy Analyzer provides bounded exhaustive analysis and translates Alloy formulas using a given *scope*, i.e., a bound on the universe of discourse, to propositional formulas, which are solved using state-of-the-art satisfiability (SAT) solvers. Given an Alloy specification, which consists of declarations of sets and relations and constraints on their values, the analyzer computes *instances*, i.e., assignments to the sets and relations, which satisfy the given constraints. The analyzer's ability to rapidly solve constraints makes the Alloy tool-set particularly attractive.

Practitioners new to Alloy, however, face a very basic challenge: the need to learn a new *declarative* language—a language that requires using a programming paradigm different from the imperative programming paradigm that is predominantly used in industry. Indeed, one of the motivating factors behind the design of Alloy is to keep the language simple and small. In particular, Alloy's support of path expressions using transitive closure enables intuitive and succinct formulation of specifications, and minimizes the learning burden on new users. Nonetheless, automated tool support in assisting with writing Alloy specifications is valuable in several aspects. To illustrate, for beginner users, it can help with learning the basics of the language; for advanced users, it can help with checking whether they formulated their constraints correctly or whether they need additional constraints.

We present aDeryaft, a novel technique for automating the writing of Alloy specifications. Instead of writing a complete specification by hand, the user constructs a few small concrete instances that represent the constraints of the software structure they want to specify. aDeryaft then fully automatically generates an executable Alloy specification, which summarizes the given structures as Alloy constraints.

To efficiently and effectively generate Alloy specifications, aDeryaft exploits the relational basis of Alloy and formulates properties of graphs that are likely to hold for the given instances. Our current prototype focuses on generating constraints that define heap-allocated data structures, such as lists and trees, which pervade modern software and can be viewed as rooted edge-labeled graphs.

aDeryaft incorporates a number of heuristics, which allow it to focus on relevant structural properties. For example, it

identifies a *core* set of fields such that for any given structure, all the objects that are reachable from the root along any field are also reachable from the root along the core fields only. aDeryaft conjectures reachability properties, such as acyclicity, only for the fields in the core set. aDeryaft treats the fields not in the core set as *derived* and conjectures relations between derived and core fields, e.g., whether a derived field is the transpose of a core field. For non-linear structures, aDeryaft also conjectures properties that relate path lengths from the root. Thus, the conjectured properties include both local and global properties. aDeryaft checks the conjectured properties for the given instances and translates the valid properties into Alloy.

The undecidability of the problem that aDeryaft addresses necessitates that its constraint generation, in general, cannot be sound and complete [7]. The generated constraints are sound with respect to the set of given instances. Of course, unseen instances may or may not satisfy them. aDeryaft's generation is not complete: it may not generate all possible constraints that hold for the given set of instances.

Nevertheless, the constraints that aDeryaft generates serve several useful purposes. The constraints can be viewed as drafts that include data declarations and constraints that are of interest to the user, who may refine them. The user could indeed choose to put together a specification by using different sets of inputs to generate different constraints, which represent different parts of a desired specification. The user can then combine the parts using standard logical operators to form a complete specification. It is worth pointing out that the Alloy Analyzer can directly use the specifications generated by aDeryaft to systematically enumerate a large number of instances that satisfy the same constraints as the given instances. Indeed the generated specifications can be used to check the soundness and completeness of hand-written specifications.

aDeryaft's approach has the potential to change how practitioners develop software designs and specifications. It not only makes writing them easier and enables various design-level and specification-level analyses, it also enables automated test input generation: the constraints generated by aDeryaft can be used directly by a test generation tool, such as TestEra [8], to enumerate a high quality test suite.

We make the following contributions:

- **Specification generation.** We introduce the idea of using a given set of Alloy instances to generate Alloy formulas that are fully executable and can also be used to augment or correct existing Alloy specifications;
- **Graph view.** We introduce a rooted edge-labeled graph view of the program heap where the set of edges is partitioned into core and derived sets to enable efficient and effective constraint generation;
- **Algorithm.** We present an algorithm for generating Alloy specifications using a small number of instances.
- **Experiments.** We present experiments that show the feasibility of generating Alloy constraints for a variety of commonly used data structures.

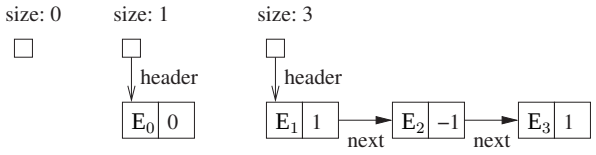

Fig. 1. Three acyclic singly-linked lists, one each containing zero, one, and three entries, as indicated by the value of the `size` field. Small hollow squares represent the list atoms. The labeled arrows represent the fields `header` and `next`. E_0, E_1, E_2, and E_3 represent the identities of entry atoms. The entries also contain the integer elements.

The rest of the paper is structured as follows. Section II describes an example to illustrate aDeryaft's specification generation and to introduce basics of Alloy. Section III defines core and derived fields and describes the aDeryaft algorithm. Section IV illustrates the algorithm using a case study with heaps implemented using binary trees. Section V discusses limitations of our current prototype and presents some threads of future work. Section VI compares our approach to previous approaches and Section VII concludes our paper.

II. Example

This section presents an example of using aDeryaft to generate an Alloy specification that represents the structural invariants of a singly-linked acyclic list. We also use the example to introduce the basics of Alloy; details can be found elsewhere [1].

For a given set of instances, aDeryaft generates an executable Alloy specification, which includes both the data declarations and the constraints in Alloy. It generates the data declarations as Alloy *signatures*, and generates the constraints as Alloy *facts* and *predicates*. It also generates appropriate commands to execute the analyzer.

Consider providing aDeryaft as input the three lists illustrated in Figure 1. Our current prototype takes the input instances as a hash-set of Java objects, and based on the classes of the objects, it generates appropriate Alloy specifications using a simple traversal of the given structures and reflection to determine the types of objects and their fields. We describe in Section III how an instance can be described in Alloy as well as how it can be described in Java. Here, we assume a set of instances is given.

Figure 2 shows the specification aDeryaft generates for the three list instances shown in Figure 1.

The keyword `module` names a specification. A `sig` declaration introduces a set of atoms; the signatures `List` and `Entry` declare sets of list and entry atoms. The *fields* of a signature declare relations. The field `header` introduces a binary relation of type `List x Entry`, while `size` has the type `Entry x Int`, where `Int` is a built-in Alloy type that represents the domain of integers; `elem` and `next` introduce relations of type `Entry x Int` and `Entry x Entry`. The keyword `set` declares a field to be an arbitrary relation; `lone` and `one` declare partial functions and total functions.

A fact states constraints that any instance of a specification must satisfy. The operator '`.`' represents relational composi-

```
module list

sig List {
  header: set Entry,
  size: set Int
}

sig Entry {
  elem: set Int,
  next: set Entry
}

fact ListFieldConstraints {
  all l: List {
    // at most one header
    lone l.header

    // exactly one size
    one l.size
  }
}

fact EntryFieldConstraints {
  all n: Entry {
    // exactly one elem
    one n.elem

    // at most one next
    lone n.next
  }
}

pred RepOk(l: List) {
  /* acyclicCore */
  all n: l.header.*next | n !in n.^next

  /* sizeOk */
  int l.size = #l.header.*next
}

run RepOk
```

Fig. 2. Alloy specification generated by aDeryaft for singly-linked acyclic lists with integer elements.

tion. If `o` is a scalar, and `r` a binary relation, `o.r` denotes the image of `o` under `r`. `ListFieldConstraints` uses universal quantification over the domain of lists and implicit conjunction to constrain `header` to be a partial function and `size` to be total. Similarly, `EntryFieldConstraints` constrains `elem` to be total and `next` to be partial.

A predicate is a parameterized formula that can be invoked elsewhere or executed directly using the `run` command, as we show here. The body of `repOk` uses universal quantification to constrain the lists to be acyclic, and also constrains the value of size to represent the number of nodes in the list. The operator '`*`' denotes reflexive transitive closure. The expression `l.header.*next` denotes the set of all entries reachable from `l`'s `header` following zero or more traversals along `next`. The operator '`^`' denotes transitive closure, '`!`' denotes negation, and `in` denotes set membership. Thus, the acyclicity constraint states that for all reachable entries, it is not possible to start traversal at any entry and get back to that entry. The operator '`#`' denotes set cardinality; the keyword `int` downcasts an `Int` atom to its integer literal value. The command `run` instructs the analyzer to generate instances that satisfy `RepOk` as well as all the facts.

The Alloy Analyzer performs bounded exhaustive analysis using a given scope to bound the universe of discourse. If no scope is given, the analyzer uses the default scope of 3. The specification generated by aDeryaft can thus be directly analyzed by the analyzer to systematically enumerate instances using one of its enumerating SAT solvers, such as mChaff [9].

III. Algorithm

This section presents the details of the aDeryaft algorithm for generating Alloy constraints. We focus on structural constraints of instances that represent heap-allocated data structures, which can be viewed as rooted edge-labeled graphs. Section V discusses generation of constraints representing more general instances. Our current prototype takes as input a `java.util.Set` object that contains Java structures that represent Alloy instances. aDeryaft traverses the structures to build the set and relation declarations for the resulting Alloy specification, and to conjecture constraints that are likely to hold for the given instances. It then checks the constraints for these instances, and translates the valid constraints into Alloy.

We first describe our abstract view of heap-allocated Java structures. Next, we define core and derived fields. Then, we characterize the invariants that aDeryaft can generate. Finally, we describe how the aDeryaft algorithm works.

A. Heap-allocated data as an edge-labeled graph

We take a *relational view* [8], [10], [11]: we view heap-allocated data as rooted edge-labeled directed graphs, where nodes represent objects and edges represent fields. An edge labeled f from node o to v says that the f field of the object o points to the object v (or is `null`) or has the primitive value v. Mathematically, we treat this graph as a set of nodes and a collection of relations, one for each field. We partition the set of nodes according to the types, and partition the set of edges according to the fields; we represent `null` as an empty set, i.e., if a field `f` of object `o` is `null`, there is no tuple in relation `f` that maps `o` to some atom. A particular heap is represented by an assignment of values to the sets and relations. Our model of heap at the concrete level induces an isomorphism between heaps and assignments of values to the sets and relations.

Recall the `List` example (Section II). Our view of the heap consists of three sets:

```
List
Entry
Int
```

and four relations:

```
header: List x Entry
size: List x Int
elem: Entry x Int
next: Entry x Entry
```

The "`size: 3`" list from Figure 1 represents the following Alloy instance:

```
List = { L2 }
Entry = { E1, E2, E3 }
Int = { -1, 0, 1, 3 }

header = { <L2, E1> }
size = { <L2, 3> }
elem = { <E1, 1>, <E2, -1>, <E3, 0> }
next = { <E1, E2>, <E2, E3> }
```

aDeryaft assumes that each structure has a unique root. Thus, our view of a structure is a *rooted* edge-labeled directed graph, and aDeryaft focuses on properties of such graphs, including properties that involve reachability, e.g., acyclicity.

B. Representing an Alloy instance using Java

To illustrate how our current aDeryaft prototype can be used to construct an Alloy instance using Java, we next present an example Java code snippet that builds a set containing the three lists shown in Figure 1.

```
java.util.Set instances =
                  new java.util.HashSet();

List l0 = new List();
l0.header = null;
l0.size = 0;
instances.add(l0);

List l1 = new List();
Entry e0 = new Entry();
l1.header = e0;
l1.size = 1;
e0.elem = 0;
e0.next = null;
instances.add(l1);

List l2 = new List();
Entry e1 = new Entry();
Entry e2 = new Entry();
Entry e3 = new Entry();
l2.header = e1;
l2.size = 3;
e1.elem = 1;
e2.elem = -1;
e3.elem = 0;
e1.next = e2;
e2.next = e3;
instances.add(l2);
```

where the classes `List` and `Entry` are defined as expected:

```
class List {
    Entry header;
    int size;
}

class Entry {
    int elem;
    Entry next;
}
```

Thus, using our current prototype, the user is not required to write any Alloy code. However, this limits applicability to constructing Alloy instances that can be viewed as Java object graphs. We plan to provide direct API support for building arbitrary Alloy instances.

```
Set coreFields(Set ss) {
    // post: result is a set of core fields
    // with respect to the structures in ss

    Set cs = allClasses(ss);
    Set fs = allReferenceFields(cs);
    foreach (Field f in fs) {
        Set fs' = fs - f;
        boolean isCore = false;
        foreach (Structure s in ss) {
            if (reachable(s, fs') !=
                  reachable(s, fs)) {
                isCore = true;
                break;
            }
        }
        if (!isCore) fs = fs';
    }
    return fs;
}
```

Fig. 3. Algorithm to compute core fields. The method `allClasses` returns the set of all classes of objects in structures in `ss`. The method `allReferenceFields` returns the set of all reference fields declared in classes in `cs`. The method reachable returns a set of objects reachable from the root of `s` via traversals only along the fields in the given set.

C. Core and derived fields

aDeryaft partitions the set of reference fields declared in the classes of objects in the given structures into two sets: *core* and *derived*, which are defined as follows.

Definition 1. A reference field f that represents a homogeneous relation is core with respect to a set of structures S if and only if there is some structure $s \in S$ such that there exists an object o reachable from the root object r of s such that all paths that connect o to r include a traversal along f.

In other words, if we remove all the edges labeled f from the graph that represents s, there is no directed path from r to o in the resulting graph.

To illustrate, both `header` and `next` are core in the example from Section II.

Definition 2. A reference field g is derived with respect to a set of structures S if and only if g is not core with respect to S.

Since `elem` in Section II is a primitive field, the `List` example has no derived fields.

Our field partitioning is inspired by the notion of a structure's *back-bone* [12], [13].

1) Algorithm: The set of core fields can be computed by taking each reference field in turn and checking whether removing all the corresponding edges from the graph changes the set of nodes reachable from root. Figure 3 gives an algorithm to compute core fields.

```
String deryaft(Set structs) {
    // post: result is a string representation of a Alloy constraints
    //       that represent structural invariants of given structures

    Set classes = allClasses(structs);
    Set fields = allFields(structs);
    Set core = coreFields(fields);
    Set derived = derivedFields(fields, core);
    Set relevantGlobal =
            globalProperties(structs, core, classes);
    Set relevantLocal =
            localProperties(structs, derived, classes);
    Set propertiesThatHold =
            checkProperties(relevantGlobal, structs);
    propertiesThatHold.addAll(
            checkProperties(relevantLocal, structs));
    simplify(propertiesThatHold);
    return generateInvariants(propertiesThatHold);
}
```

Fig. 4. The aDeryaft algorithm. The methods `allClasses` and `allFields` respectively return a set of all classes and a set of all fields from the given set of structures. The method `coreFields` (`derivedFields`) returns the set of core (derived) fields. The methods `globalProperties` (`localProperties`) compute sets of global (local) properties relevant to the given structures. The method `checkProperties` returns a subset of given properties, which hold for all given structures. The method `simplify` removes redundant constraints. The method `generateInvariants` translates given properties to Alloy constraints.

D. Graph properties of interest

We consider *global* as well as *local* properties of rooted edge-labeled directed graphs. The properties are divided into various categories as follows.

1) Global: reachability: Reachability properties include the *shape* of the structure reachable from root along some set of reference fields. The shapes can be *acyclic* (i.e., there is a unique path from the root to every node), *directed-acyclic* (i.e., there are no directed cycles in the graph), *circular* (i.e., all the graph nodes of a certain type are linked in a cycle), or *arbitrary*; note that any acyclic graph is also directed-acyclic. To illustrate, the property $acyclic($`header`, $\{$`next`$\})$, i.e, the structure reachable from `header` along the field `next` is acyclic, holds for all the given list structures in Figure 1.

2) Global: primitive fields: In reasoning about graphs, the notion of cardinality of a set of nodes occurs naturally. We consider properties relating values of integer fields and cardinalities of sets of reachable objects. For example, the property $equals($`size`, $reachable($`header`, `next`$).cardinality())$ checks whether the `size` field denotes the number of objects reachable from `header` following zero or more traversals of `next`.

3) Global: path lengths: For non-linear structures, such as trees, we consider properties that relate lengths of different paths from root. For example, the property *balanced* represents that no simple path from the root differs in length from another simple path by more than one. For binary trees, this property represents a *height-balanced* tree.

4) Local: reference fields: In edge-labeled graphs that are not acyclic (along the set of all fields), local properties that relate different types of edges are likely. To illustrate, consider a graph where if an edge connects a node n of type N to a node m of type M, there is a corresponding edge that connects m to n. We term such properties *two-cycles*. For a doubly-linked list, such as those implemented in the Java Collection Framework [14], `next` and `previous` form a two-cycle.

Another local property is whether some node never has a particulr edge emanating from it, i.e., the corresponding object always has a certain field set to `null`.

5) Local: primitive fields: Another category of local properties pertains to primitive values. For example, in a binary tree, the value in a node might be greater than the values in the node's children. We consider local properties that relate a node's value to it's successors along reference fields.

E. Algorithm

Given a set of structures, aDeryaft traverses the structures to formulate a set of hypotheses. Next, it checks which of the hypotheses actually hold for the given structures. Finally, it translates the valid hypotheses into Alloy constraints that represent the structural invariants of the given structures, i.e., it generates a predicate that takes an input structure and evaluates to true if and only if the input satisfies the invariants.

To make invariant generation feasible, a key heuristic that aDeryaft incorporates to focus on relevant properties is: hypothesize properties about reachability, such as acyclicity or circularity, only for the fields in the core set; and hypothesize local properties that relate derived fields and core fields, e.g., whether a derived field forms two-cycles with some core fields.

Figure 4 presents the aDeryaft algorithm using Java-like pseudo-code.

To minimize the number of properties that are checked on the given structures, the `checkProperties` does not check a property p if a property q that contradicts p is already known to be true, e.g., if acyclic holds then circular (for the same set of fields) is not checked.

To minimize the number of constraints in the generated `repOk`, the `simplify` method removes redundant properties from set of properties that actually hold, e.g., if a graph is acyclic, there is no need to generate a check for directed-acyclic. We are considering the use of a decision procedure for simplification. Indeed we may use the Alloy Analyzer to simplify.

In summary, the algorithm performs the following six key steps:

- Identification of Alloy signatures and fields;
- Identification of core and derived fields;
- Formulation of relevant global and local properties;
- Computation of properties that actually hold;
- Minimization of properties; and
- Generation of Alloy code that (1) declares signatures and fields and (2) defines constraints on them based on the discovered properties.

IV. Case-study: binary tree representation of heaps

To illustrate the variety of invariants that aDeryaft can generate, we next present a case study on generating invariants of the heap data structure, which is also called a priority queue [15]. Consider a binary tree representation of heaps, where nodes have `left` and `right` children and `parent` pointers, as well as integer `key`s, with the constraints:

- acyclicity along `left` and `right`
- `parent` is transpose of `left + right`
- `size` is number of nodes in the tree
- heap property: for any node, its key is greater than any key in a left or right child

To illustrate execution of the `deryaft` algorithm (Figure 4), consider computing the constraints for the given structures. The formulation of relevant global properties gives[1]:

- $acyclic$(`root`, {`left`, `right`})
- $directed\text{-}acyclic$(`root`, {`left`, `right`})
- $circular$(`root`, {`left`, `right`})
- $equals$(`size`, $reachable$(`root`, {`left`, `right`}).$cardinality$())
- $equals$(`size` + 1, $reachable$(`root`, {`left`, `right`}).$cardinality$())

The formulation of relevant local properties gives:

- $two\text{-}cycle$(`root`, `parent`, `left`)
- $two\text{-}cycle$(`root`, `parent`, `right`)
- $is\text{-}null$(`root`, `parent`)
- $less\text{-}than$(`root`, {`left`})
- $less\text{-}than$(`root`, {`right`})
- $less\text{-}equal$(`root`, {`left`})
- $less\text{-}equal$(`root`, {`right`})
- $greater\text{-}than$(`root`, {`left`})
- $greater\text{-}than$(`root`, {`right`})
- $greater\text{-}equal$(`root`, {`left`})
- $greater\text{-}equal$(`root`, {`right`})

[1]We have not listed the height-balanced property since our aDeryaft prototype does not yet handle it. Section V discusses the prototype's limitations.

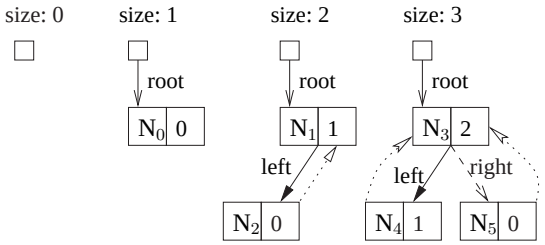

Fig. 5. Four heaps represented using binary trees, containing zero, one, two and three nodes, as indicated by the `size` field. Small hollow squares represent the `BinaryTree` objects. The labeled arrows represent the fields `root`, `left`, `right`. The dotted arrows with hollow heads represent `parent` fields, which have not been labeled for clarity. $N_0, \ldots, N_5$ represent the identities of node objects. The nodes also contain the integer keys.

Consider the heaps represented in Figure 5. As an example execution of the algorithm for computing the core fields (Figure 3), consider computing the set with respect to these structures. The algorithm initially sets `fs` to {`left`, `right`, `parent`}, i.e., the set that contains all the fields that represent homogeneous relations. Removing `left` from the set changes reachability, e.g., in the case of the structure with three nodes and therefore `left` is core; similarly `right` is core; however, removing `parent` does not effect the reachability in any of the given structures and therefore `parent` is not core.

The computation of properties that actually hold gives:

- $acyclic$(`root`, {`left`, `right`})
- $directed\text{-}acyclic$(`root`, {`left`, `right`})
- $equals$(`size`, $reachable$(`root`, {`left`, `right`}).$cardinality$())
- $two\text{-}cycle$(`root`, `parent`, `left`)
- $two\text{-}cycle$(`root`, `parent`, `right`)
- $is\text{-}null$(`root`, `parent`)
- $greater\text{-}than$(`root`, {`left`})
- $greater\text{-}than$(`root`, {`right`})
- $greater\text{-}equal$(`root`, {`left`})
- $greater\text{-}equal$(`root`, {`right`})

Removal of redundant properties gives:

- $acyclic$(`root`, {`left`, `right`})
- $equals$(`size`, $reachable$(`root`, {`left`, `right`}).$cardinality$())
- $two\text{-}cycle$(`root`, `parent`, `left`)
- $two\text{-}cycle$(`root`, `parent`, `right`)
- $is\text{-}null$(`root`, `parent`)
- $greater\text{-}than$(`root`, {`left`})
- $greater\text{-}than$(`root`, {`right`})

In addition to these properties, aDeryaft computes the following constraints on the relations that model the object fields:

- $partial\text{-}function$(`root`)
- $total\text{-}function$(`size`)
- $total\text{-}function$(`key`)
- $partial\text{-}function$(`left`)
- $partial\text{-}function$(`right`)
- $partial\text{-}function$(`parent`)

Given the four heaps shown in Figure 5 as input, aDeryaft outputs the Alloy specification that represents the structural constraints of the inputs. Figure 6 shows this specification.

V. Discussion and future work

This section discusses current limitations of aDeryaft and future work.

Constraint generation using a given set of structures has two limitations. One, the set may not be representative of the class of desired structures. Two, not all relevant properties can feasibly be identified, e.g., conjecturing all possible relations among integer fields is infeasible even using simple arithmetic operators. aDeryaft's current generation algorithm therefore, focuses on structural properties which involve reference fields, which can naturally be viewed as edges in a graph. In future, we plan to further explore likely relations among primitive fields.

Our aDeryaft prototype at this stage can handle structures similar to the ones illustrated in this paper, but does not yet support properties of path lengths in a graph.

1) User-controlled specification generation: It is useful to have a specification generation system that can focus on constraints that are deemed interesting by the user. This enables not only focused generation for a particular domain, but also generation of a wider class of constraints, since a generic constraint generation system may hypothesize too many properties and as a result may not feasibly generate any that hold. We plan to extend aDeryaft to support user-controlled generation using the Alloy grammar to define the class of constraints to consider. We believe aDeryaft can greatly assist users in developing their Alloy specifications correctly.

2) Connectivity and symmetry breaking constraints: When the specifications generated by aDeryaft are used for enumerating instances using the Alloy Analyzer, connectivity and symmetry breaking constraints play an important role [16]. Consider enumerating acyclic lists as specified in Figure 2. Even though all the input structures represent connected components (by construction), no constraints in the specification require that all nodes in the generated instance belong to exactly one list. Indeed, the specification does not even restrict instances to consist of exactly one list. To restrict generation to interesting instances, it is useful to add such constraints.

Another issue with enumerating instances arises due to structural symmetries, as induced by graph isomorphism [16], [17]. Even though Alloy provides default support for symmetry breaking [18], the support is not sufficient to allow efficient enumeration and explicit symmetry breaking constraints need to be written in Alloy [16].

We plan to explore generation of connectivity and symmetry breaking constraints.

3) Instance construction API: Our current aDeryaft prototype takes its input structures as Java objects. Thus, the constraint generation does not (directly) consider properties of arbitrary relations, e.g., a relation that may map a node to more than one nodes. We plan to develop a simple instance

```
module binarytree

sig BinaryTree {
  root: set Node,
  size: set Int
}

sig Node {
  left: set Node,
  right: set Node,
  parent: set Node,
  key: set Int
}

fact BinaryTreeFieldConstraints {
  all b: BinaryTree {
    lone b.root
    one b.size }
}

fact NodeFieldConstraints {
  all n: Node {
    lone n.left
    lone n.right
    lone n.parent
    one n.key }
}

pred RepOk(t: BinaryTree) {
  /* acyclicCore */
  all n: t.root.*(left + right) {
    n !in n.^(left + right)
    no n.left & n.right
    lone n.~(left + right) // '~' is transpose
  }

  /* sizeOk */
  int t.size = #t.root.*(left + right)

  /* parentNull */
  no t.root.parent

  /* parentTwoCycleLeft */
  all n: t.root.*(left + right) {
    some n.left => n.left.parent = n
  }

  /* parentTwoCycleRight */
  all n: t.root.*(left + right) {
    some n.right => n.right.parent = n
  }

  /* greaterThanLeft */
  all n: t.root.*(left + right) {
    some n.left => int n.key > int n.left.key
  }

  /* greaterThanRight */
  all n: t.root.*(left + right) {
    some n.right => int n.key > int n.right.key
  }
}

run RepOk
```

Fig. 6. Alloy specification generated by aDeryaft for heaps.

construction API for Alloy to allow representation of general Alloy instances.

4) General Alloy specifications: Our presentation has so far focused on generating Alloy specifications that represent instances that can be viewed as heap-allocated structures, which we view as rooted edge-labeled graphs. Alloy specifications, however, can in general specify various other structures and properties. As a simple example, consider constraining an arbitrary relation to be an equivalence relation. We do not currently handle constraints of relations that do not represent total or partial functions.

It is also worth noting that Alloy specifications do not always have instances. For example, consider using the Alloy analyzer to check the equivalence of two formulas, where the analyzer tries to generate a counterexample within the given scope. If the formulas are equivalent there is no such counterexample. Even though aDeryaft can assist with developing such specifications, say by generating a part of the specification using instances that satisfy one of the formulas in the equivalence, aDeryaft may not be able to characterize exactly what the user wants. For example, the user may want to check the equivalence of the following two definitions of acyclicity:

```
pred AcyclicUniversal(l: List) {
  all e: l.header.*next | e !in e.^next
}

pred AcyclicExistential(l: List) {
  no l.header ||
    some e: l.header.*next | no e.next
}
```

Since the definitions are equivalent, any set of valid list instances satisfies both of them, and aDeryaft given any input set might just generate one of the two definitions.

There are various open research issues regarding generation of general Alloy specifications, and we plan to systematically explore them.

VI. Related work

A. Dynamic analyses

Our work is inspired by the Daikon invariant detection engine [7], which pioneered the notion of dynamically detecting likely program invariants in the late 90s and has since been adapted by various other frameworks [19]–[25]. aDeryaft differs from Diakon in three key aspects. One, the model of data structures in Daikon uses arrays to represent object fields. While this representation allows detecting invariants of some data structures, it makes it awkward as to how to detect invariants that involve intricate global properties, such as relating lengths of paths. aDeryaft's view of the heap as an edge-labeled graph and focus on generic graph properties enables it to directly capture a whole range of structurally complex data. Two, aDeryaft employs specific heuristics that optimize generation of invariants for data structures, e.g., the distinction between core and derived fields allows it to preemptively disallow hypothesizing relations among certain fields. We believe this distinction, if adopted, can optimize Daikon's analysis too. Third, aDeryaft generates invariants in Alloy, which can directly be plugged into a variety of tools, such as the TestEra testing framework [8].

For generating class invariants as imperative predicates for Java data structures, we have developed the jDeryaft framework [26]. jDeryaft generates the invariants as Java predicates, i.e., Java methods that traverse a given structure and return true or false based on whether the structure satisfies the invariants.

The Perracotta project presents novel ways of scaling dynamic techniques to large programs, but focuses on mining temporal API usage rules [27].

Software agitation is a recently proposed unit testing technique that builds on Daikon and combines test generation and invariant detection; the Agitator embodies this technique in a commercial tool [25].

B. Static analyses

Researchers have explored invariant generation for over three decades, but primarily in the context of generating loop invariants using static analyses, including recurrence equations, abstract interpretation with widening, matrix theory for Petri nets, and constraint-based techniques [28]–[31]. While static analyses guarantee that generated invariants hold for all executions, they are often restricted to finding relations between primitive variables, and they have so far not been shown to feasibly generate representation invariants for structurally complex data.

C. Combined dynamic/static analyses

Some recent approaches combine static and dynamic analyses for inferring API level specifications [32], [33].

Invariant generation has also been used in the context of model checkers to explain the absence of counterexamples, while focusing on integer variables [34].

VII. Conclusions

We presented aDeryaft, a novel technique for automating the writing of Alloy specifications using a small set of given instances. Alloy has steadily been gaining popularity due to its automatic analyzer's ability to provide rapid feedback. Alloy users however, still have to write their specifications by hand in a declarative language—a language very different from the commonly used (imperative) programming languages.

aDeryaft provides a novel approach to writing Alloy specifications. The user constructs by hand a few small concrete instances that represent the constraints of the software structure they want to model. aDeryaft then fully automatically generates an executable Alloy specification, which represents the constraints that summarize the given structures. The generated specification can, in fact, be used by the Alloy Analyzer to systematically enumerate a large number of concrete instances that satisfy the same constraints as the given instances. We illustrated aDeryaft's constraint generation on a variety of commonly used data structures.

We believe the approach presented by aDeryaft holds promise in making construction of designs and specifications

in Alloy amenable to practitioners, thereby holding potential to make a profound impact on improving the reliability of software.

ACKNOWLEDGMENT

We thank Aman Pervaiz for helping us with the design and implementation of jDeryaft, which forms the foundation of aDeryaft. We thank the anonymous reviewers and Darko Marinov for useful comments. This work was funded in part by the NSF Science of Design award #0438967.

REFERENCES

[1] D. Jackson, *Software Abstractions: Logic, Language and Analysis.* The MIT Press, Cambridge, MA, 2006.

[2] D. Jackson, I. Schechter, and I. Shlyakhter, "ALCOA: The Alloy constraint analyzer," in *Proc. 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.

[3] D. Box, *Essential COM*. Addison Wesley, 1998.

[4] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The design and implementation of an intentional naming system," in *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, Dec. 1999.

[5] G. Dennis, "TSAFE: Building a trusted computing base for air traffic control software," Master's thesis, Massachusetts Institute of Technology, 2003.

[6] D. Jackson and M. Jackson, *Separating Concerns in Requirements Analysis: An Example*. Springer-Verlag, (To appear).

[7] M. D. Ernst, "Dynamically discovering likely program invariants," Ph.D. dissertation, University of Washington Department of Computer Science and Engineering, Seattle, Washington, Aug. 2000.

[8] D. Marinov and S. Khurshid, "TestEra: A novel framework for automated testing of Java programs," in *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, Nov. 2001.

[9] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 39th Design Automation Conference (DAC)*, June 2001.

[10] D. Jackson and A. Fekete, "Lightweight analysis of object interactions," in *Proc. Fourth International Symposium on Theoretical Aspects of Computer Software*, Sendai, Japan, Oct. 2001.

[11] D. Jackson and M. Vaziri, "Finding bugs with a constraint solver," in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Portland, OR, Aug. 2000.

[12] A. Moeller and M. I. Schwartzbach, "The pointer assertion logic engine," in *Proc. SIGPLAN Conference on Programming Languages Design and Implementation*, Snowbird, UT, June 2001.

[13] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard, "Field constraint analysis," in *Proc. 7th International Conference on Verification, Model Checking and Abstract Interpretation*, 2006.

[14] *Java 2 Platform, Standard Edition, v1.3.1 API Specification*, Sun Microsystems, http://java.sun.com/j2se/1.3/docs/api/.

[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[16] S. Khurshid, D. Marinov, I. Shlyakhter, and D. Jackson, "A case for efficient solution enumeration," in *Proc. Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, Santa Margherita Ligure, Italy, May 2003.

[17] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, July 2002, pp. 123–133.

[18] I. Shlyakhter, "Generating effective symmetry-breaking predicates for search problems," in *Proc. Workshop on Theory and Applications of Satisfiability Testing*, June 2001.

[19] T. Xie and D. Notkin, "Tool-assisted unit test generation and selection based on operational abstractions," *Automated Software Engineering Journal*, 2006.

[20] C. Csallner and Y. Smaragdakis, "Dynamically discovering likely interface specifications," in *ICSE'06, Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, May 24–26, 2006, emerging results track.

[21] D. Lorenzoli, L. Mariani, and M. Pezzè, "Inferencing state-based behavior models," in *Workshop on Dynamic Analysis (WODA)*, Shanghai, China, May 23, 2006.

[22] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss, "Automated fault localization using potential invariants," in *AADEBUG'2003, Fifth International Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, September 8–10, 2003, pp. 273–276.

[23] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, 2002.

[24] N. Gupta and Z. V. Heidepriem, "A new structural coverage criterion for dynamic detection of program invariants," in *Proc. 18th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, October 2003.

[25] M. Boshernitsan, R. Doong, and A. Savoia, "From daikon to agitator: lessons and challenges in building a commercial tool for developer testing," in *ISSTA'06: Proceedings of the 2006 international symposium on Software testing and analysis*, 2006.

[26] S. Khurshid and M. Z. Malik, "Generating representation invariants of structurally complex data," (Under submission).

[27] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal api rules from imperfect traces," in *Proc. 28th International Conference on Software Engineering (ICSE)*, May 2006.

[28] S. M. German and B. Wegbreit, "A synthesizer of inductive assertions." *IEEE Trans. Software Eng.*, vol. 1, no. 1, 1975.

[29] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proc. 5th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, Tucson, Arizona, 1978.

[30] A. Tiwari, H. Rue, H. Saidi, and N. Shankar, "A technique for invariant generation," in *Proc. 7th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, London, UK, 2001.

[31] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Non-linear loop invariant generation using gröbner bases," in *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2004.

[32] M. Taghdiri, "Inferring specifications to detect errors in code," in *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, Washington, DC, 2004.

[33] J. Whaley, M. C. Martin, and M. S. Lam, "Automatic extraction of object-oriented component interfaces," in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.

[34] M. Vaziri and G. Holzmann, "Automatic detection of invariants in spin," in *Proc. SPIN Workshop on Software Model Checking*, Nov. 1998.