Correcting a Naming Architecture using Lightweight Constraint Analysis

Sarfraz Khurshid and Daniel Jackson
MIT Laboratory for Computer Science
200 Technology Square
Cambridge, MA
khurshid@lcs.mit.edu

Abstract

We used lightweight formal modeling and automatic analysis to explore and correct the design and implementation of the Intentional Naming System (INS). INS is a new scheme for resource discovery and service location in dynamic networks. We constructed a model of INS in Alloy, a lightweight relational notation, and analyzed it with the Alloy Analyzer (AA), a fully automatic simulation and checking tool. In doing so, we exposed several serious flaws in both the algorithms of INS and its underlying naming semantics.

To correct the flaws in the INS algorithms, we developed a formal specification for naming in Alloy and refined it to specify the correctness criteria for INS algorithms. We accordingly modified the INS algorithms and verified their correctness using AA. Finally, we mapped these changes onto the original INS implementation, thereby correcting both the original design and implementation of the naming architecture of INS.

1 Introduction

In this article, we present how we used Alloy [13] and the Alloy Analyzer (AA) [11, 12] to explore and correct the design and implementation of the Intentional Naming System (INS) [1, 22]. We built an Alloy model of the published design of INS [1] and used AA to expose several flaws in both the algorithms of INS and its underlying naming semantics.

AA's analysis helped us identify the cause of failure of correctness properties in the INS design: the core naming architecture of INS had a broken semantics. To correct the INS algorithms, we developed a formal specification for naming in Alloy. We refined our generic specification to specify the correctness criteria for INS algorithms.

Accordingly, we modified the INS algorithms and verified their correctness by changing our original Alloy model of INS and re-analyzing it with AA. Finally, we mapped these changes onto the original INS implementation, thereby correcting both the design and implementation of INS. Our corrections now form a part of the INS code base.

Intentional Naming System

INS is a new scheme for resource discovery and service location in dynamic networked environments. INS allows client applications to refer to *what* service they want without having to specify *where* in the network topology it resides. Services in INS are described and referred to using *intentional* names. These names describe a set of properties that the services (should) provide. Using intentional names, applications are able to communicate seamlessly with services, despite changes in the mapping from service names to service locations during a session.

The mapping between service descriptions and their locations is maintained by *name resolvers*. Like IP routers or conventional name servers, name resolvers route client requests to services. Name resolvers provide operations to register services against their descriptions, resolve client queries describing services into appropriate service locations, and disseminate information about services among resolvers.

Alloy

Alloy is a first-order, declarative language based on relations. Alloy is suitable for specifying structural properties of software. Alloy *specifications* can be analyzed automatically using the Alloy Analyzer (AA). Given a finite *scope*, i.e., a bound on the number of atoms in the universe of a specification, AA translates the specification into a propositional formula and uses SAT solving technology to generate *instances* that satisfy the properties expressed in the specification. AA can be used to check the correctness of theorems; an instance of a specification that expresses the negation of an assumed theorem provides a counterexample to that theorem.

To explore the design of INS using AA, we modeled in Alloy concrete representations of the complex data structures and recursive algorithms of INS from their published imperative descriptions [1]. Our model of computation in INS focused on the most important procedures and abstracted away others by using appropriate constraints on data structures.

We exploited AA's ability to analyze partial declarative specifications, using it to analyze our model of INS and check various partial correctness properties of INS algorithms. AA's analysis refuted several theorems expressing these properties.

Using declarative modeling, we were able to identify a wide range of flaws *without* first having to fix the ones already discovered by AA. By expressing simple constraints, we ruled out instances that were no longer of interest, thereby *masking* bugs already revealed by AA.

AA's analysis required only small scopes to refute theorems expressing correctness properties of INS. Counterexamples to these theorems indicate subtle design flaws that went undetected for over a year of use.

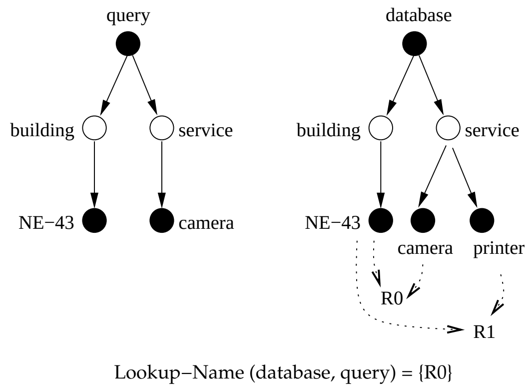

Figure 1: Example of a query, database and *Lookup-Name* operation

```
Lookup-Name(T,n)
   S <- the set of all possible records
   for each av-pair p := (na, nv) in n
      Ta <- the child of T such that
            Ta’s attribute = na’s attribute
      if Ta = null
         continue

      if nv = *         // wild card matching
          S’ <- empty-set
          for each Tv which is a child of Ta
             S’ <- S’ union (all of the records in
                             subtree rooted at Tv)
          S <- S intersection S’
      else              // normal matching
          Tv <- the child of Ta such that
                 Tv’s value = nv’s value
          if Tv is a leaf node or p is a leaf node
             S <- S intersection (the records of Tv)
          else
             S <- S intersection Lookup-Name(Tv, p)
   return S union (the records of T)
```

Figure 2: *Lookup-Name* algorithm

The rest of this article is organized as follows. Section 2 presents an overview of INS focusing on its naming architecture. Section 3 describes the key notions in Alloy and AA. Section 4 explains our Alloy model of INS. Section 5 shows the analysis of this model using the Alloy Analyzer. Semantics of naming is formalized in Section 6, and in Section 7 the existing design of INS and its algorithms are corrected. Related work is discussed in Section 8, and conclusions are presented in Section 9.

2 Intentional Naming System

One particular service discovery solution in dynamic networked environments is the *Intentional Naming System* (INS) [1], which allows services to describe and refer to each other using *intentional* names. These names describe a set of properties that the services should provide rather than specify a low-level network location, e.g. “the nearest, least-loaded color printer that handles transparencies” rather than “printer643.lcs.mit.edu” or “18.31.0.99”. This allows applications to refer to *what* service they want without having to specify *where* in the network topology it resides. It also allows applications to communicate seamlessly with end-nodes, despite changes in the mapping from name to end-node addresses during the session.

INS comprises applications and name resolvers. Applications may be clients or services. Services provide the functionality or data required by clients. Name resolvers maintain a periodically updated mapping between service descriptions and their network locations.

Like IP routers or conventional name servers, name resolvers in INS route requests from clients seeking services to appropriate locations, using a database that maps service descriptions to their physical network locations.

Both queries and advertisements are represented in INS using intentional names. An intentional name (see query in Figure 1) is conceptually an arrangement of alternating levels of *attributes* and *values* in a tree structure. In Figure 1, hollow circles identify attributes and filled circles identify values. Attributes represent categories in which an object can be classified. Each attribute has a corresponding value that is the objects classification within that category. A wildcard may be used in place of a value to show that any value is acceptable. For uniformity, the root node is treated as a value.

An attribute together with its value form an *av-pair*; each av-pair has a set of child av-pairs that further describe the object. In the query in Figure 1, $(building, NE\text{-}43)$, and $(service, camera)$ are examples of av-pairs. An av-pair that specializes another is a descendant of it, and av-pairs that are orthogonal to each other but specialize the same av-pair are siblings in the tree. The query in Figure 1 describes an object in building NE-43 that provides a camera service. Its root is an av-pair with attribute `null`, value `null`, and two children $(building, NE\text{-}43)$ and $(service, camera)$.

A name resolver stores its information in a database that maps names to *records*, which include the IP addresses of services advertising the name. In a database, unlike an intentional name, there can be multiple values per attribute, and it can be viewed as a superpositioning of all the service descriptions the name resolver knows about. The database in Figure 1 stores information about two services, one advertised using the name $((building, NE\text{-}43), (service, camera))$, and the other one using the name $((building, NE\text{-}43), (service, printer))$.

A value in a database that corresponds to a leaf av-pair of an advertisement also contains the relevant record. In Figure 1 this is represented by dotted arrows, and the database shown stores two objects, one, R0, that provides a camera service in building NE-43 and the other one, R1, that provides a printer service in the same building.

The database in a name resolver provides three key operations: resolving queries to records, adding new services, and finding the name used by a given service in its advertisement.

The records for a query are retrieved using the *Lookup-Name* operation. An algorithm for this operation is given in pseudo-code in the published description of INS [1], and is replicated in Figure 2.

The algorithm makes a series of recursive calls, but does not backtrack. Each call reduces the set of possible records by intersecting it with those contained in matching leaf nodes. When it is invoked on the query and database shown in Figure 1, R0 is returned since the value of attribute 'service' sought by the client (i.e. camera) does not match that provided by R1 (i.e. printer).

When a service advertises its availability to a name resolver, it is included in the database stored by that name resolver using the *Add-Name* algorithm. To periodically update adjacent name resolvers about new or expired services, a name resolver extracts advertisements from its database using the *Get-Name* algorithm, and conveys its neighbors.

Here we focus on the most interesting operation *Lookup-Name*, while abstracting away *Add-Name* by including constraints enforced on the structure of a database created by valid additions.

An implementation of the naming architecture of INS appears in [22]. About 1400 lines of Java code implement *Lookup-Name* and *Add-Name* and the relevant data structures and methods. INS's developers had also written 900 lines of Java code for testing purposes.

3 Alloy

Our formalization of the core model of the naming scheme of INS is written in Alloy [13], a first-order relational notation with transitive closure.

Alloy is a modeling language for describing structural properties of software. It is strongly influenced by Z: in its relational basis, its type system, and in the uniform expression of invariants and constraints as formulas. A detailed rationale for Alloy's design is given elsewhere [13].

3.1 Language

An Alloy model is a sequence of paragraphs that can be of two kinds: signatures (`sig`), used for construction of new types, and a variety of formula paragraphs (`fact`, `fun`, and `assert`), used to record constraints. Before we describe these paragraphs we give a brief description of the syntax and semantics of formulas.

3.1.1 Expressions

Alloy is a strongly typed language that assumes a universe of atoms partitioned into subsets, each of which is associated with a basic type. The standard mathematical notion of a relation gives the only datatype. The value of any expression is always a relation – that is a collection of tuples of atoms. Each element of such a tuple is atomic and belongs to some basic type (and may not itself be a relation). A relation may have any arity greater than one. Relations are typed.

Sets are expressed as unary relations (i.e., relations whose tuples have only one element). Scalars and tuples are expressed as singleton relations (i.e., relations containing only one tuple).

Relations are combined with a variety of operators to form expressions. The standard set operators – `+` for union, `&` for intersection, `-` for difference – combine two relations of the same type, viewed as sets of tuples.

There are two product operators, dot and arrow, and three unary operators, transpose, transitive closure, and reflexive transitive closure. The dot operator is relational composition: p`.`q is the relation containing the tuple $\langle p_0, \ldots, p_{m-1}, q_1, \ldots, q_n\rangle$ where p contains $\langle p_0, \ldots, p_m\rangle$, q contains $\langle q_o, \ldots, q_n\rangle$, and $p_m = q_0$. One of p and q must have an arity of two or greater, so that the result is not degenerate. When p is a unary relation (i.e., a set) and q is a binary relation, the composition p`.`q is standard relational image. For convenience, Alloy allows the relational image p`.`q to be written equivalently as q`[`p`]`,where '`[]`' binds more loosely than '`.`'. Double colon '`::`' can also be used instead of '.', but binds stronger than '.'.

The arrow operator is cross product: p`->`q is the relation containing the tuple $\langle p_0, \ldots, p_m, q_0, \ldots, q_n\rangle$ when `p` contains $\langle p_0, \ldots, p_m\rangle$, `q` contains $\langle q_0, \ldots, q_n\rangle$. When p and q are unary relations, p`->`q is just the cross product of the two sets.

The unary operators `~` (transpose), `^` (transitive closure), and `*` (reflexive transitive closure), can only be applied to binary relations, and they have their standard interpretation.

3.1.2 Formulas

Elementary formulas are formed from from the subset operator, `in`. Thus p *in* q is true when every tuple in p is in q. The formula $p : q$ has the same meaning, but when q is a set, adds an implicit constraint that p be scalar (i.e., a singleton). This constraint is overridden by writing p : `option` q (which lets p to be empty or scalar) or p : `set` q (which eliminated the constraint entirely). Equality (`=`) is just standard set equality, and is short for a subset constraint in each direction.

An arrow that appears as the outermost expression operator on the right-hand side of a subset formula can be annotated with *multiplicity markings*: `+` (one or more), `?` (zero or one) and `!` (exactly one). The formula r`:` S m`->`n T where m and n are multiplicity markings constrains the relation r to map each atom of S to n atoms of T, and to map m atoms S to each atom of T. S and T may themselves be product expressions, but are usually variables denoting sets. For example, r : S `->!` T and r`:` S `?->!` T make r respectively a total function on S and an injection.

Larger formulas can be obtained using the standard logical connectives: `&&` (and), `||` (or), `!` (not), `=>` (implies), `iff` (bi-implication). The formula a `=>` b`,` c is short for `if` b `then` f `else` g (i.e., b `=>` f `&&` `!`b `=>` g). Within curly braces, consecutive formulas are conjoined.

Quantifications take their usual form: `all` x`:`e `|` F is true when the formula F holds under every binding of the variable x to a member of the set e. In addition to the standard quantifiers, `all` (universal) and `some` (existential), Alloy has `no`, `sole`, and `one` meaning respectively that there are no values, at most one value, and exactly one value satisfying the formula. For a quantifier Q and expression e, the formula Q e is short for Q x`:` T `|` x `in` e (where T is the type of e), so `no` e, for example, says that e is empty.

The declaration of a quantified formula is itself a formula—an elementary formula in which the left-hand side is a variable.

Declarations `disj` $v_1, v_2, \ldots$ `:` e and `part` $v_1, v_2, \ldots$ `:` e are equivalent to a declaration for each of the variables $v_1, v_2, \ldots$, with an additional constraint that the relations denoted by the variables are respectively disjoint (i.e., share no tuple) and form a partition (i.e., share no tuple and their union is `e`).

3.1.3 Signatures

The signature declaration introduces a basic type and a collection of fields in it along with the types of the fields and constraints on their values. For example,

```
sig Value{}
```

introduces `Value` as a *basic signature* (or an uninterpreted type) with no fields. A signature may inherit fields and constraints from another signature. The signature declaration

```
static disj sig Root, Wildcard extends Value {}
```

declares `Root` and `Wildcard` as two disjoint subsets of `Value`. The keyword `static` constrains `Root` and `Wildcard` to contain exactly one element each. Sub-signatures do not introduce new Alloy types. The Alloy type of atoms in `Root` and `Wildcard` is therefore `Value`. New Alloy types are only introduced by basic signature declarations.

The signature declaration

```
sig AVTree {
  vnodes: set Value,
  anodes: set Attribute,
  av: anodes !->+ (vnodes - Root),
  va: (vnodes - Wildcard) !-> anodes
}
{ // all value nodes are reachable from root
  vnodes = Root.*(va.av)
  // non-empty tree
  some Root.va
}
```

declares `AVTree` as a basic signature with fields `vnodes`, `anodes`, `av`, and `va`. The expression on right side of the colon after each field denotes that field's type. For example, for each `AVTree` t, t`.vnodes` is a subset of `Value`, t`.anodes` is a subset of `Attribute`, t`.av` is a relation mapping t`.anodes` to elements in t`.vnodes` other than `Root`, and t`.va` is a relation mapping t`.vnodes` other than `Wildcard` to t`.anodes`. The multiplicity markings on t`.av` require that for each element v in t`.vnodes - Root`, there is exactly one element in t`.anodes` that is mapped to v by the relation t`.av`, and each element in t`.anodes` is mapped to at least one element in t`.vnodes - Root`. Similarly, the `!` marking on t`.va` enforces that for each a in t`.anodes`, there is exactly one v in t`.vnodes - Wildcard` that maps to a.

The two constraints attached at the end express more structural properties of `AVTree`. The first constraint states that all values (and hence also attributes) in an `AVTree` are reachable from *Root*. The second constraint states that each `AVTree` is non-empty.

A formula F attached to a signature S is implicitly enclosed by `all` *this*: S | `with` *this* `|` F. The shorthand `with` e `|` F is like F, but e prefixes field names wherever appropriate. Appropriateness is determined by type: e is matched to any field name with which it can be composed using the dot operator. So, the formula attached to `AVTree` declaration above expands to:

```
all this: AVTree |
  this.vnodes = Root.*((this.va).(this.av))
```

3.1.4 Functions, Facts and Assertions

A *function* (`fun`) is a formula that can be 'invoked' (that is, imported) elsewhere. It's called a function because given values of some arguments, it returns true/false or a relational value. The function

```
fun Lookup(db:DB, q: Query, r: set Record) {
  r = State.lookup[db][q][Root]
}
```

returns true if the arguments `db` of type `DB`, `q` of type `Query`, and subset `r` of type `Record` satisfy the constraint in curly braces, and false otherwise. Alternatively `Lookup` could be written as

```
fun DB::Lookup(q: Query): set Record {
  result = State.lookup[this][q][Root]
}
```

which means exactly the same as before but illustrates the special treatment of first and last arguments of a function and the use of keywords `this` and `result`. `Lookup` can be invoked as db`..Lookup(`q`)` on a database db and parameter query q.

A *fact* (`fact`) is a formula that takes no arguments and need not be invoked explicitly; it is always true. An *assertion* (`assert`) is a formula that is intended to be true. An analyzer tool should try to determine whether assertions follow from facts.

Because functions are defined to be used elsewhere, they must be named. Facts are often named too, but they need not be.

3.2 Analyzer

The Alloy Analyzer (AA) [11, 12] is an automatic tool for analyzing models created in Alloy. Given a formula and a *scope* – a bound on the number of atoms in the universe – AA determines whether there exists a model of the formula (that is, an assignment of values to the sets and relations that makes the formula true) that uses no more atoms than the scope permits, and if so, returns it.

First-order logic is undecidable, so AA's analysis cannot be a decision procedure. AA therefore limits its analysis to a finite scope that bounds the size of the carrier sets of the basic types. We say that a model is *within a scope of* k if it assigns to each type no more than k elements. Clearly, if AA succeeds in finding a model to a formula, it has demonstrated the formula's consistency. Failure to find a model within a given scope, however, does not prove that the formula is inconsistent. In practice, however, many errors can be found by considering only a small scope (e.g., at most 3 elements of each type). This *small scope hypothesis* is purely empirical of course, since the language is undecidable, so for any scope there is a formula whose smallest model occurs in a larger scope.

AA provides two kinds of analysis: *simulation* in which the consistency of a fact or function is demonstrated by generating a snapshot showing its invocation, and *checking*, in which a consequence of the specification is tested by attempting to generate a counterexample for an assertion. In both cases the tool produces graphical (and also textual) output.

To perform AA's analysis, one needs to fix the number of atoms in each basic type. Typically, if the tool quickly reports back its completion of analysis without generating an instance or counterexample, the scope is increased to raise the user's level of confidence in the model. One needs to check an instance and counterexample to make sure it is not an artifact of a modeling error.

The power of declarative specification, in which negation and conjunction are used freely, comes with two risks: over- and under-constraint. Simulation helps catch errors of over-constraint, by reporting, contrary to the user's intent, that no instance exists within the finite bounds of a given scope, or by showing instances sharing an unintended structural property. Checking helps catch errors of under-constraint, by showing instances that are acceptable to the specification but which violate an intended property.

```
sig Value {}
sig Attribute {}
sig Record {}

static disj sig Root, Wildcard extends Value {}

sig AVTree {
  vnodes: set Value,
  anodes: set Attribute,
  av: anodes !->+ (vnodes - Root),
  va: (vnodes - Wildcard) !-> anodes
}
{ // all value nodes are reachable from root
  vnodes = Root.*(va.av)
  // non-empty tree
  some Root.va
}

disj sig Query extends AVTree {}
{ // every attribute node has exactly one child
  all a:anodes | one a.av
}

disj sig DB extends AVTree {
  records : set Record,
  recs: (vnodes - Root) +-> records
}
{ // wildcard does not appear in the database
  Wildcard !in vnodes
  // leaf values nodes contain a record
  all v: vnodes | no v.va => some v.recs
  // records appear at lowest possible level
  all v: vnodes | no v.recs & v.^(~av.~va).recs
  // in advertisements each att node has one child
  all a: anodes | all disj v1, v2: a.av |
    no v1.*(va.av).recs & v2.*(va.av).recs
}

static sig State {
  lookup : DB -> Query -> Value -> Record
}
```

Figure 3: Alloy model of intentional names

AA's analysis is based on a translation to a boolean satisfaction problem, and gains its power by exploiting state-of-the-art SAT solvers. The analysis algorithm is described in [11].

One can use AA in the style of traditional model checking, in an attempt to find subtle errors in existing designs. We use this strategy to expose flaws in INS in the following sections.

4 INS model

We first build an Alloy model of the data structures that represent a query and a database in INS and then we build a model of the computation of *Lookup-Name*.

4.1 Data structures

Our model of the data structures representing queries and databases (Figure 3) is based on the INS description in [1].

A query is essentially modeled as two relations `av` and `va`, while a database is modeled as three relations `av`, `va`, and `recs`. Well-formedness constraints are expressed as Alloy facts. Since Alloy functions may not be recursive, *Lookup-Name* is modeled by introducing a new Alloy relation in the specification; appropriate constraints on the relation `lookup` in signature `State` model *Lookup-Name*.

In detail, the components of the model of intentional names are as follows:

- the signatures `Value`, `Attribute`, `Record` introduce these basic types.
- `Root` is a special `Value` that (without loss of generality) acts as the root value of all queries and databases in our model.[1] `Wildcard` is also a special `Value` (different from `Root`) and models the wildcard value used by queries.
- `AVTree` defines the basic type representing trees with alternating levels of attributes and values like intentional names in INS. An `AVTree` has fields `vnodes`, `anodes`, `av` and `va` representing respectively values, attributes, edges from attributes to values and edges from values to attributes that form the tree. There is no need to have a distinct root value for each tree since a tree is uniquely identified by these four fields.
- `Query` just inherits all the fields and facts of `AVTree`, with the additional restriction that each `a` in `anodes` has exactly one child from `vnodes`.
- `DB` also inherits the fields and constraints of `AVTree`, and its elements model databases. For a database, *db*, *db*`.records` is the set of all records contained in it, *db*`.recs` is a relation that maps a non-root value in *db*`.values` to *db*`.records` stored at that value. The structure of a database is further constrained by the facts that all leaf values must contain some record, records occur in the lowest possible nodes, and each advertisement in a database has exactly one value corresponding to an attribute in the advertisement. These constraints are modeling the construction of a valid database.
- Singleton signature `State` is used solely to model the computation of *Lookup-Name*. A tuple (db, q, v, r) in the relation *State.lookup* indicates that the record r is an element in the result of (recursive) invocation of *Lookup-Name* on sub-trees rooted at value v, when initially invoked on database db, with parameter query q. Equivalently, the result of this invocation is modeled by `State.lookup[`*db*`][`*q*`][`*v*`]`.

4.2 Computation

Modeling the recursive computation of *Lookup-Name* is perhaps the most subtle aspect of our model (Figure 4). Since AA does not yet handle recursive Alloy functions, we model recursion using a novel technique that we have developed in the course of this work. We model a recursive procedure by introducing a new relation in the state and expressing constraints on that relation that compute a fixed point of the procedure.

For a database db, query q, the invocation *db*`..Lookup(`*q*`)` models the method invocation *db.Lookup-Name*(q) in INS. The relation `lookup` in `State` models recursion. The function `Lookup` simply invokes its helper function `LookupAux` on

[1] Not using a different root node for each query or database expedites the automatic analysis.

```
fact LookupFixPoint {
  all db: DB, q: Query, v: Value, r: Record |
    r in db..LookupAux(q,v) <=>                          // record r is in the result if and only if
    {
     all na: q.va[v] | all nv: q.av[na] |                // for all child av-pairs (na,nv) of av-pair with value v in q
      no db.va[v] & na ||                                // either: no child of the db node with value v has attribute na
      some Ta: db.va[v] {
         Ta = na                                         // or: Ta is a child node with attribute na
         nv = Wildcard =>                                //     wildcard matching
           r in db.recs[db.av[Ta]] + db.recs[v],         //      r is a record of any child of Ta or of v
                                                         //     normal matching
           no db.av[Ta] & nv ||                          //      either: no child of Ta has value nv
           some Tv: db.av[Ta] {
             Tv = nv                                     //      or: Tv is a child of Ta with value nv
             (no q.va[nv] || no db.va[Tv]) =>            //          if Tv is a leaf-node or nv is a leaf-node
               r in db.recs[Tv] + db.recs[v],            //               r is a record of Tv or of v
               r in db..LookupAux(q,Tv) + db.recs[v] } } //          else r is a record of the recursive call at Tv or of v
    }
}

fun DB::LookupAux(q: Query, v: Value): set Record {      // helper function for Lookup
  result = State.lookup[this][q][v]
}

fun DB::Lookup(q: Query): set Record {                   // models Lookup-Name algorithm invocation
  result = this..LookupAux(q,Root)
}
```

Figure 4: Modeling *Lookup-Name*

the input database and parameters q and `Root`. The `result` of invocation db`..LookupAux(`q`,`v`)` is the set of records that `State.lookup` relation maps db, q and v to.

The fact `LookupFixPoint` expresses constraints that compute a fixed point[2] of `Lookup`. These constraints closely follow the pseudocode in Figure 2. A difference, however, is that in our model a value (attribute) may not appear more than once in an `AVTree`. Note also, the initial invocation of *Lookup-Name* is on the root nodes of database and query and each subsequent recursive invocation is (also) on nodes that have identical values. Therefore, we declare

```
lookup: DB -> Query -> Value -> Record
```

and not

```
lookup: DB -> Query -> Value -> Value -> Record
```

This (simplifying) optimization improves the time to analyze properties of our model, without sacrificing the soundness of counterexamples.

In our model, we also make use of the following two mathematical equivalences for finite sets $S, T, S_i, i \in I = \{1, \ldots, n\} \subset N$:

$$S = \cap_{i \in I} S_i \ \equiv \quad \forall x \cdot x \in S \iff \forall i \in I \cdot x \in S_i \qquad (1)$$

$$(\cap_{i \in I} S_i) \cup T \ \equiv \ \cap_{i \in I} (S_i \cup T) \qquad (2)$$

Equivalence (1) is simply the definition of intersection over a finite collection of sets. Equivalence (2) expresses distributivity of union over intersection.

We use equivalence (1) to formulate our constraints using expressions involving membership operator, e.g., **`r in db.recs[Tv]...`**.

[2]Since *Lookup-Name* is deterministic, these constraints compute the least fixed point.

We use equivalence (2) to distribute the union operation from the last line of pseudocode in Figure 2, over each of the intersections in the algorithm, e.g., **`r in db.recs[Tv] + db.recs[v]`**.

5 Analysis of INS

We use AA to inspect correctness of *Lookup-Name* by analyzing the model that we constructed in Section 4. In particular, we evaluate both partial correctness properties and the claims and proposed fixes of INS inventors regarding the *Lookup-Name* algorithm. The theorems we check can broadly be divided into the following categories:

- obvious (partial correctness) properties, e.g., the services returned by *Lookup-Name* do not have functionality conflicting a client's request, so that the search of a camera does not result in a printer;
- essential (partial correctness) properties, e.g., addition to database does not reduce results of *Lookup-Name* so that the availability of new services in the system does not reduce the availability of services to a client;
- published claims about the behavior of INS algorithms e.g., correspondence of wildcards to missing attributes in the behavior of *Lookup-Name*.
- desirable properties about the behavior of INS algorithms, e.g., clients may use partial descriptions so that if a client only knows of a service's partial description, it can still access the service by posing a query that partially specifies the service's properties;
- INS implementation fixes, e.g., implementation of modified algorithms works correctly;

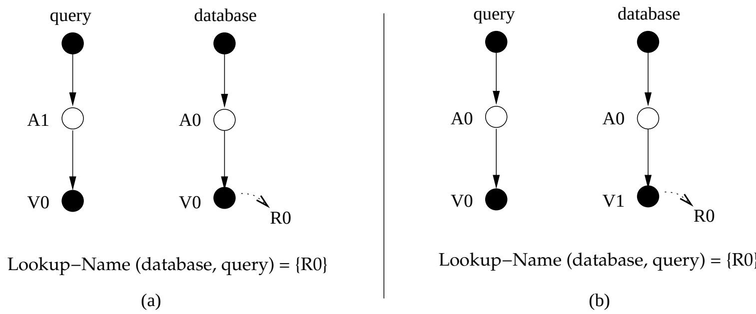

Figure 6: AA counterexamples to `AttMismatch` (a), `ValMismatch`(b)

```
assert AttMismatch {
  //if no common top level attribute,
  //lookup should return empty-set
  all db:DB | all q:Query |
    no Root.(q.va) & Root.(db.va) => no db..Lookup(q)
}

fun MaskAttMismatch (db: DB, q: Query) {
  all v: q.vnodes & db.vnodes |
    some q.va[v] && some db.va[v]
    => some q.va[v] & db.va[v]
}

assert ValMismatch {
  //if some top level att has value not in db
  //lookup should return empty-set
  all db:DB | all q:Query |
    {
     MaskAttMismatch(db,q)
     some a: Root.(q.va) & Root.(db.va) |
       not a.(q.av) in a.(db.av) + Wildcard
    } =>
    no db..Lookup(q)
}

fun MaskValMismatch (db: DB, q: Query) {
  all a: q.anodes & db.anodes |
    q.av[a] in db.av[a] + Wildcard
}
```

Figure 5: Checking obvious partial correctness properties

5.1 Theorems

We formulate the theorems we need to check as Alloy assertions and use AA to check each theorem. We start AA's analysis in each case using the default scope which bounds each signature to 3 atoms and vary it as needed.

5.1.1 Obvious properties

Our first theorem (`AttMismatch`, Figure 5) states a rather obvious partial correctness property that if no attribute in the query tree (at the top level) exists in the database, no service should be returned by *Lookup-Name*.

AA refutes this theorem by generating a counterexample (Figure 6 (a)) showing a query that has no attributes in common with the database, and the result of invoking *Lookup-Name* on these parameters that would actually return an (invalid) service, R0. This erratic behavior of the *Lookup-Name* algorithm is a result of a missing case to deal with the situation when none of the attributes sought is present in the database (lines 5–6, Figure 2).

Our next theorem states a similar property that if the services in the database provide values for an attribute in the query different from the value sought by the query, no services should be returned. Notice that AA's analysis of `AttMismatch` has indicated that *Lookup-Name* behaves erratically when it fails to find in the database any child attribute of a value in the query. Before we check our next theorem, we rule out instances in which such an attribute mismatch occurs. Ruling out such instances guarantees us that if any other flaws are identified, their cause is different from simply an attribute mismatch.

To *mask* the bug due to an attribute mismatch we express the Alloy function `MaskAttMismatch` that requires some child attribute of every (non-leaf) value in the given query to have a corresponding attribute in the given database. It is worth noting here that `MaskAttMismatch => AttMismatch`, i.e., invoking `MaskAttMismatch` rules out counterexample(s) that refute `AttMismatch`.

The theorem `ValMismatch` (Figure 5) assumes `MaskAttMismatch` as a pre-condition and asserts that if all services in database *db* provide a value for an attribute other than a value sought in query *q* then no service should be returned on *q*'s resolution in *db*. Even this claim is invalidated by AA as shown by the counterexample in Figure 6(b). The *Lookup-Name* algorithm fails to handle the case when none of the values sought for an attribute in the query actually exist in the database (between lines 16 and 17, Figure 2).

Notice that we have been able to identify another bug by just masking the previous one, without actually fixing it first. Continuing this strategy, for future analyses we include `MaskValMismatch` as well as `MaskAttMismatch` as preconditions. `MaskValMismatch` simply states that the values corresponding to all non-wildcarded attributes in the given query have matching values in the given database.

5.1.2 Essential properties

An essential partial correctness property for a naming scheme like INS is *monotonicity of addition*: simply adding a new service to a database should not reduce results of *Lookup-Name*. We next formulate a theorem that states this property.

We define an Alloy function (Figure 7) `Add` that for given databases *db* and *dbr* determines whether *dbr* is *db* with (possibly) some more advertisements (non-deterministically) added to it i.e., *dbr* contains (at least) all advertisements

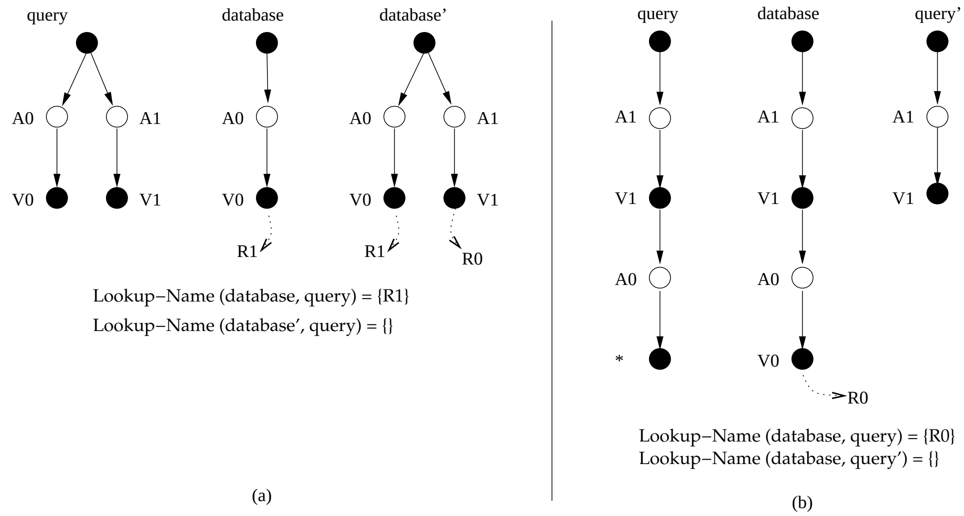

Figure 8: AA counterexamples to `AddMonotonic` (a), `WildcardClaim`(b)

```
fun Add(db:DB, dbr:DB) {
  //dbr is db with some non-deterministic additions
  db.anodes in dbr.anodes
  db.vnodes in dbr.vnodes
  db.records in dbr.records
  db.av in dbr.av
  db.va in dbr.va
  db.recs in dbr.recs
  all v:dbr.vnodes |
    dbr.recs[v] - db.recs[v] in dbr.records - db.records
}

assert AddMonotonic {
  all db,dbr: DB | all q: Query |
    {
     MaskAttMismatch(db,q)
     MaskValMismatch(db,q)
     Add(db,dbr)
     MaskAttMismatch(dbr,q)
     MaskValMismatch(dbr,q)
    } =>
    db..Lookup(q) in dbr..Lookup(q)
}
```

Figure 7: Checking monotonicity of addition

that exist in *db*. Assertion `AddMonotonic` checks monotonicity of addition. `AddMonotonic` assumes `MaskAttMismatch` and `MaskValMismatch` as pre-conditions with respect to the query q and both the original database *db* and the database resulting after addition(s) *dbr*.

AA produces a counterexample (Figure 8(a)) that displays a simple situation in which this key property is violated. On inputs *database*, *query*, *Lookup-Name* returns $R1$, but when invoked on inputs $database'$, *query*, it returns an empty set. Notice that $database'$ contains the advertisement of service $R1$ in *database* and also contains an additional service. This defect in INS is a result of a flawed design.

5.1.3 Claims

The published description of *Lookup-Name* claims [1]:

```
fun RemoveWildcard(q: Query, qr: Query) {
  //qr is q with wildcarded attribute removed
  qr.vnodes = q.vnodes - Wildcard
  qr.anodes = q.anodes - Wildcard.~(q.av)
  qr.av = q.av - Attribute -> Wildcard
  qr.va = q.va - Value -> Wildcard.~(q.av)
}

assert WildcardClaim {
  all db : DB | all q,qr : Query |
    {
     MaskAttMismatch(db,q)
     MaskValMismatch(db,q)
     RemoveWildcard(q,qr)
     MaskAttMismatch(db,qr)
     MaskValMismatch(db,qr)
    } =>
    db..Lookup(q) = db..Lookup(qr)
}
```

Figure 9: Checking INS inventors claim about wildcards

```
assert MissingAttRestrict {
  all db : DB | all q,qr : Query |
    {
     MaskAttMismatch(db,q)
     MaskValMismatch(db,q)
     RemoveWildcard(q,qr)
     MaskAttMismatch(db,qr)
     MaskValMismatch(db,qr)
    } =>
    db..Lookup(qr) in db..Lookup(q)
}
```

Figure 10: Checking whether removing wildcarded attribute always reduces results

> This algorithm uses the assumption that omitted attributes correspond to wildcards; this is true for both the queries and advertisements.

We next formulate a theorem (`WildcardClaim`, Figure 9) to check the validity of this claim using AA. The Alloy func-

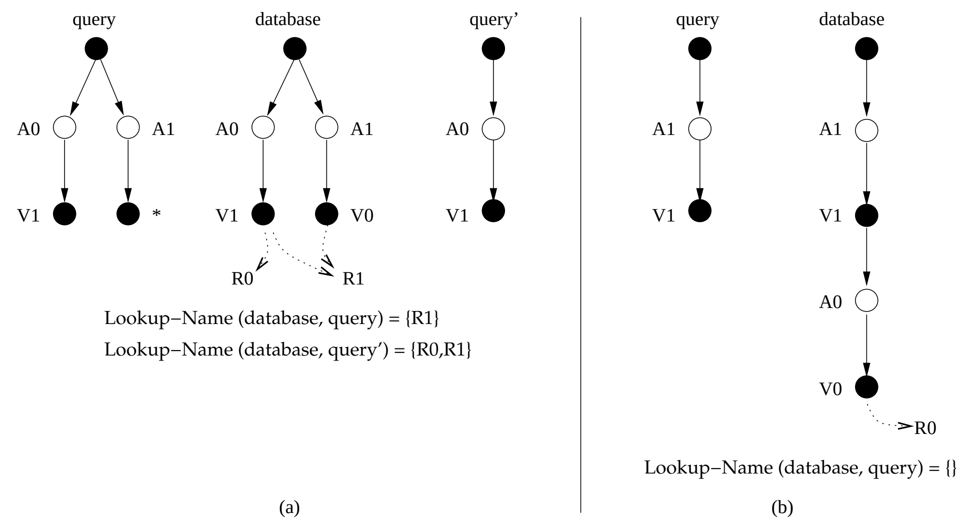

Figure 11: AA counterexamples to `MissingAttRestrict` (a), `PartialMatch` (b)

tion `RemoveWildcard` determines if query qr is the same as q but has omitted attributes in q that had wildcard values. `WildcardClaim` asserts that for a database db and such queries q, qr, the result of *Lookup-Name* on db and q should be the same as that on db and qr.

The counterexample shown in Figure 8(b) produced by AA refutes this claim about the behavior of the name resolution algorithm. Resolving *query* in *database* results in record $R0$, but when the wildcarded attribute $A0$ is removed from *query* to form *query*$'$, *Lookup-Name* resolves it in *database* to an empty set.

Observing this behavior, we conjectured that perhaps removal of an attribute with wildcard value always has this restrictive effect. Assertion `MissingAttRestrict` (Figure 10) checks this conjecture.

AA produces a counterexample to this assertion (Figure 11(a)). Notice that the use of wildcards instead of omitting wildcarded attributes could both increase or decrease the services that result from *Lookup-Name* algorithm.

5.1.4 Desirable properties

We next formulate a theorem to check whether INS permits partial matches in the case when an application looking for a service only knows its partial description, e.g., in the case when a client requires a picture of the Whitehouse but does not know its organization into wings and rooms, and a service provides one for the Oval office in West Wing and mentions that in its advertisement.

Function `QuerySubTreeAdv` (Figure 12) states that the given query is a sub-tree of the database and there is exactly one advertisement in the database. Assertion `PartialMatch` (Figure 12) therefore checks that if the query is a sub-tree of the advertisement in the database then that service should be returned. Here we do not include `MaskAttMismatch` and `MaskValMismatch` since `QuerySubTreeAdv` is already imposing a stronger requirement.

AA generates a counterexample (Figure 11(b)) to show that such desirable partial matches are not necessarily permitted by INS.

```
fun QuerySubTreeAdv(db: DB, q: Query) {
  q.av in db.av
  q.va in db.va
  one db.records
}

assert PartialMatch {
  all db: DB | all q: Query |
    QuerySubTreeAdv(db,q) => some db..Lookup(q)
}
```

Figure 12: Checking a desirable property

5.1.5 Fixes

The counterexamples produced by AA in checking the theorems `AttMismatch` and `ValMismatch` hold for the published description of *Lookup-Name*; apparently these flaws were fixed in the Java implementation of INS [22].

We incorporated the fixes from the Java implementation by modifying our Alloy model to check their correctness. AA showed that these fixes were actually incorrect.

The fix for bug revealed by analysis of `AttMismatch` involved representing the set of all services by setting a boolean flag without actually adding any service to the set. In this case when *Lookup-Name* is invoked on query and database that give a counterexample to `AttMismatch`, *Lookup-Name* implementation (correctly) returns an emptyset. However, AA helped us identify a flaw in this fix in the implementation: a union of a set with its flag set with any other set (incorrectly) results in a set with its flag set—this results in valid services not to be returned when the union on last line of pseudo-code in Figure 2 is performed.

Details of analyzing the proposed fixes are given in [16].

5.2 AA's counterexample generation

The counterexamples presented in this section were generated by AA in a few seconds, and required at most 4 elements in each signature. Mostly 2 or 3 elements in each signature were sufficient to generate a counterexample. Ta-

	Theorem refuted by AA
Partial correctness	No record if no attributes match
	No record if no values match
	Addition does not invalidate existing records
Published claim	Missing attribute ∼ wild-card
Desirable (not claimed)	Clients may specify partial names
Fixes in implementation	Attribute mismatch bug fixed
	Value mismatch bug fixed

Table 2: Summary of theorems refuted

	Scope				*Time*
Assertion	*Val*	*Att*	*Rec*	*AVT*	*(sec)*
`AttMismatch`	3	2	1	2	3.3
`ValMismatch`	4	1	1	2	3.8
`AddMonotonic`	4	2	2	3	11.3
`WildcardClaim`	4	2	1	3	5.5
`MissingAttRestrict`	4	2	2	3	7.3
`PartialMatch`	4	2	1	2	4.1

Table 1: AA's counterexample generation

ble 1 tabulates the time and scope needed for AA's analysis. Table 2 summarizes the theorems refuted by AA.

All experiments were performed on a 700 MHz Pentium III processor with 256 MB of RAM.

6 Naming

Naming is a fundamental issue of growing importance in distributed systems. Intentional Naming System (INS) [1] presents an intriguing approach to solving the problem of mobile resource discovery and service location. However, as our analysis in Section 5 showed, *Lookup-Name*, the algorithm in INS that resolves client requests by providing access to services needs to be corrected.

In this section, we develop a specification of intentional names as they are used in INS and develop a formal definition of conformance between an advertisement and a query. Notice that our analysis in Section 5 focused on checking theorems concerning partial correctness properties of *Lookup-Name*. There we focused on these properties since INS's inventors do not state the correctness properties of their algorithms. This lack of formal treatment of correctness in INS contributed to the flaws in *Lookup-Name*.

6.1 Generic specification

Figure 13 gives a generic specification for a naming architecture that uses a tree structure to name objects and provides procedures for adding new objects and resolving queries. A name is simply a rooted labeled tree, where labels are drawn from some atomic domain. Service objects are also drawn from an atomic domain. A database must support operations to add new services, and resolve a client's request based on a description of the desired service or provide the description a given service used in its advertisement.

In detail, `Record`, `DB`, and `Label` are uninterpreted types representing sets of services, databases, and labels to form names. A label can simply be thought of as being a string.

```
module NamingSpec

sig Record {}

sig DB {}

sig Label {}

sig Node {
  label: Label
}

sig LabelTree {
  root: Node,
  nodes: set Node,
  children: nodes !-> (nodes - root)
}
{ // connected
  nodes = root.*children
}

sig Query extends LabelTree {}

sig Advertisement extends LabelTree {}

fun DB::Add(a: Advertisement, r: Record):DB {}

fun DB::Get(r: Record):Advertisement {}

assert AddGet {
  all a:Advertisement | all r: Record | all db: DB |
    db..Add(a,r)..Get(r) = a
}

fun DB::Lookup(q:Query):set Record {}

fun Conforms(q:Query, a:Advertisement) {}

assert LookupConforms { //soundness and completeness
  all q: Query | all db: DB |
    all r: Record | all a: Advertisement |
      Get(db,r,a) =>
      {
       Conforms(q,a) <=> r in db..Lookup(q)
      }
}
```

Figure 13: Generic specification

`Node` specifies the nodes that form trees. For each atom a in signature `Node`, a`.label` is the label assigned to a.

`LabelTree` specifies a rooted labeled tree that denotes a name describing a service. For each atom t in `LabelTree`, t`.root` is the root node of t, t`.nodes` is the set of all nodes in

t, and t.children is the edge relation of t. The fact attached to this sig declaration constrains t to be connected. This fact together with the multiplicity markings on children constrain t to be (structurally) a tree.

Query and Advertisement denote respectively the (labeled) trees that form queries and advertisements.

Function Add gives the signature of addition operation on a database *db*, that returns a database resulting after the addition of service r advertised with name a in *db*. Get gives the signature of an operation on database *db* that for service r returns the name used to advertise r.

The assertion AddGet algebraically specifies the correctness condition for Add and Get. It simply states that adding a service r with name a into database *db* and then extracting the name of r in the resulting database is a. Notice that this fact requires that Add refreshes *db*'s contents, and the new advertisement overwrites any previous information about that service—this property holds for common naming systems like the domain name system and the Unix file system, and also for INS.

Lookup provides the signature of the function for resolving a query q in the database *db*, resulting in a set of services.

Function Conforms for input query q and an advertised name a determines whether a service advertising a is valid for serving the application requesting q.

The assertion LookupConforms algebraically specifies the soundness and completeness conditions for the query resolution operation. It requires Lookup to return exactly all the conforming services in the database.

We could have specified DB as an extension of LabelTree with appropriate relations modeling mapping of advertisements to services, but our primary concern is to develop a notion of conformance between a query and an advertised name, so we leave the structure of a database unspecified. The functions Conforms, Add, Get, and Lookup are also left unspecified. Next, we develop a concrete definition of Conforms.

6.2 Refinement

Various naming systems have characteristics similar to INS. One such example is the domain name system (DNS). Most hierarchical file systems also share similar features. The specification given in Figure 13 can be refined to model such systems in a straightforward fashion.

6.2.1 DNS

As an illustration, in Figure 14, we give a specification for DNS where an operation like nslookup provides resolving an IP address for a given DNS entry. DNS names can be modeled as labeled trees where nodes have at most one child. Conforms simply calls the helper function ConformsAux, which is specified in a recursive fashion for simplicity. ConformsAux tests whether the labels in the query and advertisement match (exactly) at the given level, and then traverses down the trees by making a recursive call, accepting when it reaches leaf nodes in both the trees.

The definition of conformance developed above considers an advertised DNS name to conform to the query if all the nodes in the query have corresponding nodes with matching labels in the advertisement and moreover the trees have the same depth. We use a similar idea for specifying conformance in INS.

```
sig DNSTree extends LabelTree {}
{
  // at most one child
  all n: nodes | sole n.children
}

disj sig Query extends DNSTree {}

disj sig Advertisement extends DNSTree {}

fun ConformsAux (q: Query, a: Advertisement,
               qn: Node, an: Node) {
  qn.label = an.label
  (no q.children[qn] && no a.children[an]) ||
    ConformsAux(q, a, q.children[qn], a.children[an])
}

fun Conforms(q:Query, a:Advertisement) {
  ConformsAux(q, a, q.root, a.root)
}
```

Figure 14: DNS to IP resolution (nslookup)

6.2.2 INS

Names in INS can also be modeled as rooted labeled trees with alternating levels of attributes and values. Figure 15 specifies this notion by giving an alternative refinement of Fig 13.

Attribute and Value are subsets of Label denoting the sets of attributes and values used to construct intentional names. Wildcard and Null are special values; these labels are not in Attribute.

AVTree is a subset of LabelTree and introduces two new fields partitioning nodes, namely anodes and vnodes, representing the set of nodes that contain attribute labels and those that contain value labels. Only the root node contains the value label Null; anodes contain attributes while vnodes contain values. All nodes have children with distinct labels. A node containing an attribute has children nodes containing values, and the children of a node with a value can only contain attributes. Wildcard cannot be specialized.

Query is a subset of AVTree, and each attribute node in query has exactly one child. A database (DB) is also a subset of AVTree. Moreover, for each database *db*, *db*.records is a relation that denotes the set of all records stored in it, and *db*.recs is a relation that maps each value node in it to the set of records stored there; the root node has no records. A database does not have wildcarded values, all leaf nodes in a database must contain some record, records appear at the lowest possible level in the database tree, and addition of services should reflect that in an advertisement, an attribute has exactly one value.

Once again, we model recursion by introducing the singleton signature State to specify the functions Conforms and Lookup. Constraints on relations State.conforms and State.lookup respectively constrain Conforms and Lookup.

Note that this model allows for repeated occurrences of attributes (values) in AVTrees. Since our goal in building a specification for INS is to correct INS, we model the intentional names and *Lookup-Name* in their full generality as allowed by INS.

Our notion of conformance (Figure 16) for correcting INS is a slight generalization of the sub-tree relation for rooted labeled trees. We require all attributes and values appearing

```
sig Attribute extends Label {}

sig Value extends Label {}

static disj sig Wildcard,Null extends Value {}

fact AttributesNotWildcardedOrNull {
  Wildcard !in Attribute
  Null !in Attribute
}

sig AVTree extends LabelTree {
  part vnodes, anodes: set nodes
}
{
  root in vnodes
  root.label = Null
  Null !in (vnodes - root).label - anodes.label
  anodes.label in Attribute
  vnodes.label in Value
  all n: nodes | all disj c,c': n.children |
    c.label != c'.label
  all a: anodes | some a.children
  children in (anodes -> vnodes) + (vnodes -> anodes)
  no Wildcard.~label.children
}

disj sig Query extends AVTree {}
{
  all a: anodes | one a.children
}

disj sig Advertisement extends AVTree {}
{
  Wildcard !in vnodes.label
}

disj sig DB extends AVTree {
  records: set Record,
  recs: (vnodes - root) +-> records
}
{
  Wildcard !in vnodes.label
  all v: vnodes {
    no v.children => some v.recs
    no v.recs & v.^(~children).recs }
  all a: anodes | all disj v,v': a.children |
    no v.*children.recs & v'.*children.recs
}

static sig State {
  conforms: Query -> Advertisement -> Node -> Node,
  lookup: DB -> Query -> Node -> Node -> Record
}
```

Figure 15: A specification for INS data structures based on the (generic) initial specification

in the query to be present in the advertisement in the correct order. A non-wildcard label in a query *matches* an identical label in an advertisement; the wildcard matches any label.

The function `Conforms` simply invokes its helper function `ConformsAux`. The fact `ConformsFixPoint` expresses the constraints that compute the (least) fixed point of `Conforms` by appropriately constraining the relation `State.conforms`.

A service under this definition is, therefore, considered valid to serve a client if it provides all the properties requested by the client but may also provide additional func-

```
fact ConformsFixPoint {
  all q: Query | all a: Advertisement |
    all nq: Node | all na: Node |
      q..ConformsAux(a,nq,na) <=>
      {
       nq.label in Wildcard + na.label
       all nq': q.children[nq] | some na': a.children[na] |
         q..ConformsAux(a,nq',na')
      }
}

fun Query::ConformsAux(a: Advertisement, nq: Node, na: Node) {
  na in State.conforms[this][a][nq]
}

fun Query::Conforms(a:Advertisement) {
  this..ConformsAux(a, this.root, a.root)
}
```

Figure 16: Basic notion of conformance in INS

tionality. Using this definition also has the nice consequence that it allowing clients to partially specify their requests, if they don't have complete information of the name used by some service.

7 Correcting INS

We now focus on using the specification for naming in INS created above to correct the *Lookup-Name* algorithm and to check its correctness using AA.

7.1 Modifying *Lookup-Name*

Our initial task is to modify *Lookup-Name* to fix the flaws identified by AA's analysis. The counterexamples that AA has already produced in Section 5 are used in conjunction with the formalization of semantics of conformance to correct this algorithm. Figure 17 gives the corrected algorithm. (The original *Lookup-Name* algorithm is given in Figure 2.)

AA counterexamples to Alloy assertions `AttMismatch` and `ValMismatch` (attribute and value mismatch theorems, Figure 6(a and b)) suggest inclusion of a statement of the form `return empty-set` (lines 6 and 17–18 in Figure 17) to respectively handle these boundary cases.

Figure 11(b) suggests that the existing *Lookup-Name* algorithm terminates before going to sufficient depth in the database tree. We, therefore, add an auxiliary function (lines 11–12 and 20–21 in Figure 17) to traverse the database tree and collect the service objects that correctly advertised all attributes and values in the query and possibly more.

Another change we make is to remove the union in line 21 in Figure 2 and replace it by line 24 in Figure 17, since the previous algorithm would consider services valid even if they did *not* provide all characteristics required by a client. For the same reason, `Tv is a leaf node` is removed from the `if` test (line 15, Figure 2), as shown in line 19 of Figure 17.

7.2 Checking correctness

Recall the correctness condition for *Lookup-Name* (from Section 6):

```
assert LookupConforms { //soundness and completeness
  all q: Query | all db: DB |
    all r: Record | all a: Advertisement |
```

```
fact LookupFixPoint {
  all db: DB, q: Query, T: Node, n: Node, r: Record | with db |
    r in db..LookupAux(q,T,n) <=>                               // record r is in the result if and only if
    {
     all na: n.q::children | all nv: na.q::children |           // for all child av-pairs (na,nv) of av-pair n in q
      some Ta: T.children {
         Ta.label = na.label                                    //  Ta is a child node with attribute na
         nv.label = Wildcard =>                                 //  wildcard matching
           r in Ta.^children.recs,                              //   r is a record of any child of Ta
           some Tv: Ta.children {                               //  normal matching
             Tv.label = nv.label                                //   Tv is a child of Ta with value nv
             no nv.q::children =>                               //   if Tv is a leaf-node
               r in Tv.*children.recs,                          //        r is a record of Tv or of v
               r in db..LookupAux(q,Tv,nv) } }                  //   else r is a record of the recursive call at Tv
    }
}

fun DB::LookupAux(q: Query, vd: Node, vq: Node): set Record {   // helper function for Lookup
  result = State.lookup[this][q][vd][vq]
}

fun DB::Lookup(q: Query): set Record {                          // models Lookup-Name algorithm invocation
  result = this..LookupAux(q,this.root,q.root)
}
```

Figure 19: Model of corrected *Lookup-Name*

```
Lookup-Name(T,n)
   S <- the set of all possible records
   for each av-pair p := (na, nv) in n
      Ta <- the child of T such that
            Ta's attribute = na's attribute
      if Ta = null
         return empty-set

      if nv = *        // wild card matching
          S' <- empty-set
          for each Tv which is a child of Ta
             S' <- S' union (all the records in
                subtree rooted at Tv and its subtrees)
          S <- S intersection S'
      else             // normal matching
          Tv <- the child of Ta such that
                  Tv's value = nv's value
          if Tv = null
             return empty-set
          if p is a leaf node
             S <- S intersection (all the records in
               subtree rooted at Tv and its subtrees)
          else
             S <- S intersection Lookup-Name(Tv, p)
   return S
```

Figure 17: Corrected *Lookup-Name* algorithm

```
      Get(db,r,a) =>
      {
       Conforms(q,a) <=> r in db..Lookup(q)
      }
}
```

To check correctness of the modifications we have made to the *Lookup-Name* algorithm, we first specify the function *Get-Name* and then model the changes to *Lookup-Name*.

Function `Get` in Figure 18 specifies *Get-Name*. `Get` extracts the advertisement corresponding to record r from the database it is invoked on. Intuitively, it does so by climbing up the database tree from each node corresponding to a leaf in the advertisement, and collecting on its way the nodes in the tree until it gets to the root.

```
fun DB::Get(r: Record):Advertisement {
  root[result] = root[this]
  nodes[result] =
    nodes[this] & r.~this::recs.*~this::children
  anodes[result] =
    anodes[this] & r.~this::recs.*~this::children
  vnodes[result] =
    vnodes[this] & r.~this::recs.*~this::children
  all n: result.nodes |
      n.~result::children = n.~this::children
}
```

Figure 18: Specification for *Get-Name*

Figure 19 gives the Alloy constraints that model the modified *Lookup-Name*. This model closely follows the technique we used to build our initial model of *Lookup-Name* given in Figure 4.

We use AA to check the correctness of the assertion `LookupConforms` in a scope of 6 nodes, 6 labels and 2 records to build each query, advertisement, and database. Notice that this scope is sufficient to generate all counterexamples that appear in Section 5. AA completes its analysis of `LookupConforms` without producing any counterexamples, thereby giving us confidence that our modifications have corrected the *Lookup-Name* algorithm.

8 Related work

8.1 Alloy applications

Alloy has previously been used to analyze Java programs. The approach for analyzing bounded initial segments of computation sequences manipulating linked lists in [15] presents a way to automatically build an Alloy model of computation. This technique, however does not appear to scale. This is because Alloy does not yet support recursion, control flow, or sequencing of operations, which makes it hard to model imperative code.

Our previous work [16, 17] on analyzing an Alloy model of computation in INS focused on the published description of *Lookup-Name*, and discussed the flaws in the design and implementation of INS presented in Section 5. In this work, we formalize the semantics of naming in INS, modify the original design and implementation and check the correctness of the modified algorithms.

We have recently developed TestEra [18, 19], a novel framework for testing Java programs. TestEra uses Alloy and AA as an enabling technology. TestEra does not require modeling computation in Alloy, instead the key idea is to execute the actual program. Using a model of input data structures to a program, TestEra generates all non-isomorphic test cases. The program is executed on each test case and a model of the program's correctness criteria is used to check the functional correctness of the program. TestEra provides a promising approach to scaling AA's analysis capabilities in testing real world applications where computation may be too complex to be modeled effectively in a declarative notation.

Alloy and Alloy Analyzer were used to refine a model of Microsoft COM's query interface and aggregation mechanism [14]. This study produced a much simpler model and a better understanding of its key properties.

8.2 Analysis tools

Many analysis tools are available, with varying degrees of automation and coverage. They can be broadly divided into the following categories:

- Model checkers such as SPIN [10] provide similar exhaustive search to AA. They generally require the system to be described as an abstract program and do not support partial, declarative specification. In this study, for example, it would not have been possible to analyze *Lookup-Name* in isolation. The input languages of model checkers generally provide only low-level datatypes, and are not designed for the kind of structural complexity of this problem.

- Theorem provers such as PVS [21] can, unlike AA, prove a theorem for all possible cases, thus offering greater confidence, but at greater expense. Theorem provers tend not to fail gracefully, and do not generally provide counterexamples. They tend to require considerable expertise on the part of the user, in the development of lemmas and proof strategies.

- Specification animation tools, such as IFAD's VDM tool [2], allow an abstract specification to be executed from given states. Executability is obtained by limiting the language, ruling out the kind of declarative specification that we used here. Also, like conventional testing tools, they generally do not perform an exhaustive search, but rather check cases specified explicitly by the user.

There has been a lot of recent interest in applying model checking to software. The Bandera project [5] provides automated support for reducing a program's state space through program slicing and data abstraction. It allows users to analyze properties of Java programs by modeling them into the input language of one of several existing model checkers like SPIN and SMV [20].

VeriSoft [7] is a tool for systematically exploring state spaces of systems composed of several concurrent processes executing arbitrary code written in C or C++. VeriSoft can automatically detect coordination problems between the processes of a concurrent system.

Most of the work on applying model checking to software has focused on analyzing event sequences in asynchronous concurrent software whose complexity primarily lies in the large number of possible execution orderings, and not on analyzing data structures whose complexity lies in the data structures themselves and algorithms that manipulate them.

8.3 Case studies

State-space exploration techniques have been used successfully to detect subtle yet important errors in the design and implementation of several complex hardware and software systems.

Clarke et al [4] used SMV to formalize and verify the cache coherence protocol in a draft IEEE Futurebus+ standard. They used temporal logic model checking techniques to uncover several errors in the standard. This appears to be the first time that formal methods have been used to find nontrivial errors in a proposed IEEE standard.

Godefroid et al [8] used VeriSoft to analyze the "Heart-Beat Monitor" (HBM), a telephone switching application developed at Lucent Technologies. They discovered flaws in the original documentation on this application and unexpected behavior in the software itself. The results of their analysis led to a redesign of the HBM software for its next release.

Boigelot et al [3] used SPIN to analyze the ACCESS.bus protocol—a serial communication protocol aimed at providing a simple way to connect peripheral devices to a host computer. They developed a model of the protocol in Promela and checked linear-time temporal logic properties. Their analysis uncovered subtle flaws in the protocol design.

Dwyer et al [6] used SPIN to analyze Ada implementations of simple data structures like queues and stack. They checked some basic partial correctness properties of these data structures by limiting the size of each data structure to be at most 3. Their approach inlines procedure calls and does not handle recursion. The properties they check do not specify behavior in terms of data element values, rather they refer only to the identity of the data elements.

9 Conclusions

In this article, we showed how we used a lightweight formal method to explore and correct the design of a novel naming architecture for dynamic networked environments.

We built an Alloy model of concrete data structures and computation in INS and analyzed it with the Alloy Analyzer (AA). The model of computation in INS was constructed in a week by the first author, who at the time had no experience with the modeling notation or prior knowledge of the system to model. The key leverage was provided by AA, which allowed us to gain confidence in our model, root out modeling errors quickly, and verify theorems without the need to construct proofs.

AA's analysis identified several subtle flaws in the original design and implementation of the name resolution algorithm in INS. It also showed that an intuition of INS's inventors which motivated the design of INS was in fact false. The Alloy model of INS was an order of magnitude smaller that the Java implementation of the naming architecture of INS and the testing code used by its developers.

AA's analysis helped us identify the cause of failure of correctness properties in the original INS design: the core naming architecture of INS had a broken semantics. INS originally lacked a consistent notion of conformance between an advertised name and a query.

We gave a formal definition of conformance and constructed a specification for naming in Alloy. We specified the correctness conditions for the core naming architecture of INS using a refinement of our specification. We used this specification and the counterexamples produced by AA to modify the design of the name resolution algorithm. AA verified the modified algorithm's correctness. We then mapped these changes onto the Java implementation to correct it. This corrected implementation now forms the basis for further development of INS.

Our work illustrates how useful exhaustive analysis is in locating hard to detect bugs, even within a small scope. It also illustrates how intuition can fail even for experienced designers, and emphasizes the need for an automatically analyzable lightweight specification in building systems.

Our study is significant for three reasons. First, it realizes the vision of Guttag and Horning [9], in which a formal model is used interactively to explore the design of a system. Second, it lays a formal foundation for analysis of a class of systems that is likely to become increasingly important. Third, it is not just a routine application of model checking. Although there are many case studies (e.g., [3, 4, 6, 8]) for model checking, very few have tried to automatically analyze complex data structures [6].

We believe that the use of this kind of lightweight approach has great benefits and could result in considerable savings by detecting errors prior to implementation, especially structural flaws that are particularly hard to correct later.

Acknowledgments

We wish to thank William Adjie-Winoto for explaining INS to us. Hari Balakrishnan and Magdalena Balazinska provided us valuable feedback in formalizing the notion of conformance. We are also thankful to Darko Marinov for comments on an earlier draft. This work was funded in part by ITR grant #0086154 from the National Science Foundation.

References

[1] 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 (SOSP)*, Kiawah Island, Dec. 1999.

[2] S. Agerhold and P. Larsen. The IFAD VDM tools: Lightweight formal methods. *FM-Trends*, 1998.

[3] B. Boigelot and P. Godefroid. Model checking in practice: An analysis of the ACCESS.bus protocol using SPIN. In *Proceedings of Formal Methods Europe'96*, volume 1051 of *Lecture Notes in Computer Science*, pages 465–478, Oxford, Mar. 1996. Springer-Verlag.

[4] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and Their Apllications*. North-Holland, 1993.

[5] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, June 2000.

[6] M. B. Dwyer and C. S. Pasareanu. Model checking generic container implementations. In *International Seminar on Generic Programming*, pages 162–177, Apr. 1998.

[7] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, pages 174–186, Paris, France, Jan. 1997.

[8] P. Godefroid, R. Hanmer, and L. Jagadeesan. Model Checking Without a Model: An Analysis of the Heart-Beat Monitor of a Telephone Switch using VeriSoft. In *Proceedings of ACM SIGSOFT ISSTA'98 (International Symposium on Software Testing and Analysis)*, pages 124–133, Clearwater Beach, Mar. 1998.

[9] J. Guttag and J. Horning. Formal specification as a design tool. In *Proc. 7th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, Las Vegas, Nevada, 1980.

[10] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[11] D. Jackson. Automating first-order relational logic. In *Proc. 8th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, San Diego, CA, November 2000.

[12] 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.

[13] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proc. 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Vienna, Austria, Sept. 2001.

[14] D. Jackson and K. Sullivan. COM revisited: Tool-assisted modeling of an architectural framework. In *Proc. 8th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, San Diego, CA, 2000.

[15] 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.

[16] S. Khurshid. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. Master's thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 2000.

[17] S. Khurshid and D. Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, Sep 2000.

[18] S. Khurshid and D. Marinov. Checking Java implementation of a naming architecture using TestEra. In S. D. Stoller and W. Visser, editors, *Electronic Notes in Theoretical Computer Science (ENTCS)*, volume 55. Elsevier Science Publishers, 2001.

[19] 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.

[20] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[21] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, Feb. 1995.

[22] E. Schwartz. Design and implementation of intentional names. Master's thesis, MIT Laboratory for Computer Science, Cambridge, MA, June 1999.