Exploring the Design of an Intentional Naming Scheme with an Automatic Constraint Analyzer

Sarfraz Khurshid and Daniel Jackson
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
khurshid@lcs.mit.edu

Abstract

Lightweight formal modeling and automatic analysis were used to explore the design of the Intentional Naming System (INS), a new scheme for resource discovery in a dynamic networked environment. We constructed a model of INS in Alloy, a lightweight relational notation, and analyzed it with the Alloy Constraint Analyzer, a fully automatic simulation and checking tool. In doing so, we exposed several serious flaws in both the algorithm of INS and the underlying naming semantics. We were able to characterize the conditions under which the existing INS scheme works correctly, and evaluate proposed fixes.

1. Introduction

Naming is a fundamental issue of growing importance in distributed systems. As the number of directly accessible systems and resources grows, it becomes increasingly difficult to discover the (names of) objects of interest. Moreover, in many distributed environments – especially those involving mobile devices – applications do not know the optimal network location providing the information or functionality they require.

1.1. Intentional Naming

In an *intentional naming* and *resolution architecture*, applications describe their intent and specify *what* they are looking for but not *where* it is situated. This shifts the burden of resolving 'what is desired' to 'where it is' from the user to the network infrastructure. It also allows applications to communicate seamlessly with end-nodes, despite changes in the mapping from name to end-node addresses during the session.

The *Intentional Naming System* (INS) [1] is a recently developed framework that provides this functionality. It comprises applications (clients and services) and *intentional name resolvers* (INRs), which respond to queries from clients.

Like IP routers or conventional name servers, name resolvers route requests from clients seeking services to appropriate locations, using a database that maps service descriptions to their physical network locations. But in a name resolver, a service is described using a tree-like structure of alternating levels of attributes and values where an element at a certain level specializes the ones above it.

A name resolver provides a few fundamental operations. When a service wants to advertise itself – because, for example, it has come online after being down, or because its functionality has been extended – it calls the *Add-Name* operation to register the service against an *advertisement* describing it. Applications make queries by calling the resolver's *Lookup-Name* operation. There are other operations used to disseminate information amongst resolvers, for example. Here we focus on the most interesting operation, *Lookup-Name*, accounting for calls to *Add-Name* by characterizing legal configurations of the resolver with an invariant.

1.2. Alloy Analysis

In this paper, we explain how we used Alloy [6], a lightweight formal modelling notation, and the Alloy Constraint Analyzer [7,8], its automatic analyzer, to expose flaws in INS and explore variants of its design. We used the Alloy Constraint Analyzer interactively to refine our model to only 50 lines of Alloy. In contrast, the code of the operation is about 1400 lines of Java, does not express the key properties directly, and is not amenable to exhaustive analysis.

Our main contributions are as follows:

0-7695-0710-7/00 $10.00 © 2000 IEEE

- We show how, by construction and analysis of a succinct model, we were able to expose a variety of flaws in INS, some of which were not known to its designers. We also evaluated claims made about the properties of wildcards, and showed these to be false. In all these cases, our tool generated counterexamples showing a query and a database state that violate the expected property.
- Also using the tool, we were able to establish conditions under which the current INS algorithm for name resolution returns correct results.
- We raise issues of naming semantics that arose from our analysis and would be relevant to any intentional naming scheme. We make an attempt to deduce the essential properties of a general intentional naming scheme from our simplified model.

Our study is significant for three reasons. First, it realizes the vision of Guttag and Horning [4], 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, this is not just a routine application of model checking. Although there are many case studies for model checking, very few have tried to automatically analyze complex data structures.

We believe that this kind of lightweight approach to formal methods [15] has a promising future. The model was completed in a week by a researcher (the first author) who had no experience with Alloy or prior knowledge of INS. The key leverage was provided by our tool, which allowed us to gain confidence in our model, root out modeling errors quickly, and check theorems, without the need to construct proofs.

Our paper is organized as follows. Section 2 presents an overview of INS as described in [1]. Section 3 presents a formal Alloy model of INS, which we derived by simplifying our earlier model. In section 4, we explain the analysis we performed using the Alloy Constraint Analyzer, and discuss the soundness conditions we established, and some general issues of naming semantics. Section 5 evaluates the cost of the case study, in the relative sizes of model to code, and the performance of the tool. Section 6 discusses related work, and section 7 summarizes and concludes.

2. Overview of INS

In INS, services are described using intentional names. Clients and services use intentional names to form their *queries* and *advertisements*, respectively. Intentional names are implemented using 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 object's classification within that category. A wildcard may be used in place of a value to show that any value is acceptable.

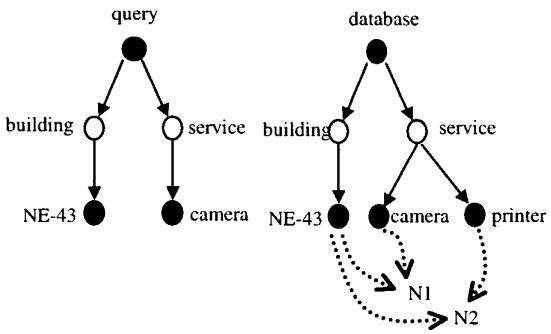


Figure 1. An illustration of Lookup-Name

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. 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, for example, consists of av-pairs (building, NE-43) and (service, camera) and describes an object in building NE-43 that provides a camera service.

An INR stores its information in a database that maps names to *records*, which include the IP addresses of services advertising the name. In a database, however, there can be multiple values per attribute, and it can be viewed as a superposition of all the service descriptions the INR knows about.

A value in a database that corresponds to a leaf av-pair of an advertisement also contains a pointer to the relevant record. In figure 1 this is represented by broken arrows, and the database shown stores two objects, one (i.e. N1) that provides a camera service in NE-43 and the other one (i.e. N2) that provides a printer service in the same building.

INRs interact with databases in two key ways: resolving queries to records and disseminating information about advertisements amongst themselves. The records for an advertisement are retrieved using the *Lookup-Name* operation. An algorithm for this operation is given in pseudocode in the published description of INS [1], and is replicated in figure 2.

The *Lookup-Name* 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

```
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 = wildcard          // wild card matching
      S' ← ϕ
      for each Tv which is a child of Ta
        S' ← S' ∪ all of the records in the
                  subtree rooted at Tv
      S ← S ∩ 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 ∩ the records of Tv
      else
        S ← S ∩ Lookup-Name(Tv, p)
  return S ∪ the records of T
```

Figure 2. Lookup-Name pseudocode from [1]

and database shown in figure 1, the first execution of the *for* loop sets S to be {N1,N2}, while the intersection in the second execution of the loop sets it equal to {N1}, which is returned as the result.

The inventors of INS claim [1] that in the execution of the algorithm 'omitted attributes correspond to wildcards'. Our analysis establishes this to be false (Section 4.2).

3. Modeling INS

Our formalization of the core model of the naming scheme of INS is written in Alloy, a first-order notation with transitive closure, that attempts to combine the best of features of Z [13] and UML [11]. From UML and its predecessors, it takes various declaration shorthands, navigations, and a focus on set-valued rather than relation-valued expressions; from Z, it takes schema structuring and a simple set-theoretic semantics.

An Alloy model is built by layering properties using conjunction, in contrast to operational languages in which the model is given by an abstract program. This allows partial models to be built, in which constraints describe how state components are related to one another, without explicit rules for how each component is updated.

A detailed rationale for Alloy's design is given elsewhere [6].

3.1. Basic components of Alloy

Domains. The *domain* paragraph introduces basic sets that partition the universe of atoms. Alloy is strongly but implicitly typed; there is a basic type associated with each domain (which in Z would be declared explicitly as a 'given type'). *Attribute*, and *Value* model respectively the attributes and values that may appear in a query and a database. *Record* models the set of records that exist in a database. Unlike a given type, a domain is a set of atoms that exist in a particular state and not a platonic set of possible atoms. So *Record* represents a set of records in a particular configuration, not the set of all imaginable records.

Multiplicities and *Mutabilities.* The symbols + (one or more), ! (exactly one) and ? (zero or one) are used in declarations to constrain sets and relations. The declaration

r : S m -> T n

where m and n are multiplicity symbols, makes r a relation from S to T that maps each S to n atoms of T, and maps m atoms of S to each T. So *recDB,* for example, maps at least one *Value* to each *Record*, which informally means that all records appear in some value. Similarly, the declaration

S : T m

makes S a set of m atoms drawn from the set T. So WildCard, for example, is a set of values with one element – ie, a scalar. Omission of a multiplicity symbol implies no constraint.

The keyword *fixed* introduces a mutability constraint. A set S declared to be fixed is unchanging: an object cannot be a member of S at one time and a non-member at another. So the declaration of *WildCard* as fixed simply means that the same value must be used consistently to represent wildcards.

Expressions. All expressions denote sets of atoms. The conventional set operators are written in ASCII form: + (union), & (intersection), - (difference). The *navigation* expression *e.r* denotes the image of the set e under the relation r: that is, the set of atoms obtained by 'navigating' along r from atoms in e. In *e.+r*, the image under the transitive closure of r is taken instead, while * and ~ denote reflexive transitive closure and transpose. Scalars are treated as singleton sets. This allows us to write navigations more uniformly, without converting between sets and scalars or worrying about the difference between functions and more general relations. So the expression

Root.attQ & Root.attDB

for example, denotes the set of attributes common to both the query and the database at the top level.

Formulas. Alloy uses the standard logical operators, written in programming-language form: && (and), || (or) and *not*. There are two elementary formulas: *s in t*, which

says that the expression s denotes a subset of the expression t (or membership when s is a scalar), and $s = t$, which says that the expressions denote the same set. Logical operator -> denotes implication.

Quantifiers. The existential and universal quantifiers are written *some* and *all*. Less conventionally, *no* $x \mid F$ and *sole* $x \mid F$ mean that there is no x and at most one x that satisfies F. Quantifiers are used in place of set constants, so

no Root.attQ & Root.attDB

for example, says that there is no attribute in the intersection of *Root.attQ* and *Root.attDB*. Bounds of quantified variables may optionally be omitted; in

all $v \mid v.immFolQ = v.attQ.valQ$

the variable v is inferred to belong to domain *Value*, and could have been written equivalently as

all v : *Value* $\mid v.immFolQ = v.attQ.valQ$

Here, we have omitted most bounds for brevity's sake, but used variable names consistently to avoid confusion: variables beginning with a, v, and r are used for attributes, values, and records, respectively.

Paragraphs. An Alloy model is divided into paragraphs much like Z schemas, but Alloy distinguishes different kinds of constraints. An invariant (introduced by the keyword *inv*) models a constraint in the world being modeled; a definition (*def*) defines one variable in terms of others, and can in principle always be eliminated along with the variable being defined. An assertion (*assert*) is a putative theorem to be checked. A condition (*cond*) is a constraint whose consistency is to be checked, but unlike an invariant is not required always to hold.

An operation (*op*) specifies transitions of the model with constraints that relate pre-states and post-states, the latter being referred to by priming the names of state components.

3.2. Alloy model of INS

Our model of INS (Figure 3), based on the INS description in [1], focuses on the abstract data structures representing a query and a database, and the behavior of *Lookup-Name*. We also capture the constraints imposed by valid additions to the database.

The query is modeled as two relations, *attQ* and *valQ*, and the database as three relations, *attDB*, *valDB*, and *recDB*. Well-formedness constraints are expressed as Alloy invariants. *Lookup-Name* is modeled by a relation *lookup*. (Alloy actually has operations, which we use later, but they may not be recursive. Since the definition of this operation is recursive, it is more convenient to model it as a relation.)

In more detail, the components are:

- the domains *Attribute, Value, Record,* that represent the sets of attributes, values, and records;
- *WildCard,* which is designated to be a special *Value*;
- *Root*, a unique *Value* that acts as both the root of the query to resolve, and the database to search;
- *valQ* and *attQ*, relations that map an attribute to its child value, and a value to its children attributes respectively, in the query. The query of figure 1 is represented by

  attQ = {Root → {building, service}}

  valQ = {building → NE-43, service → camera};
- *valDB* and *attDB*, relations that similarly map an attribute to the possible values it can take and a value to its descendant attributes respectively, but in the database. The database of figure 1 would have

  attDB = {Root → {building, service}}

  valDB = {building → {NE-43}, service → {camera, printer}};
- *recDB*, a relation that maps a value to records. The database of figure 1 would have

  recDB = {NE-43 → {N1, N2}, camera → {N1}, printer → {N2}};
- *immFolQ* and *immFolDB*, defined relations that map a value to possible values its children attributes can take in a query and a database respectively;
- *immPreDB*, a defined relation that is the transpose of *immFolDB*;
- *lookup*, the relation that models the *Lookup-Name* method, and maps each value v to a set of records; these records model the return value of *Lookup-Name* when invoked on the av-pair containing the value v and sub-database rooted at value v; thus *Root.lookup* is the result of resolving the query in the database.

This completes the definition of the domain and state and we now describe the constraints in our model.

Input database and query are non-null (*NonEmpty*).

WildCard does not appear in the database (*WC1*), and no attribute specializes it in the query (*WC2*). Also, it does not contain any records (*WC3*).

The structure of query is constrained by the following invariants:

- no attribute maps to root under the *valQ* relation, i.e. root has no ascendants (*Q1*);
- if a non-root value exists in the query, it has exactly one ascendant (*Q2*);
- if an attribute exists in the query, it has exactly one descendant value (*Q3*) and one ascendant value (*Q4*);
- the query data structure is acyclic (*Q5*); this is expressed using the transitive closure operator;

```
model INS {
 domain {Attribute, Value, Record}
 state{     disjoint Root, WildCard : fixed Value!
            valQ : Attribute? -> Value?
            attQ : Value? -> Attribute
            valDB : Attribute? -> Value
            attDB : Value? -> Attribute
            recDB : Value+ -> Record
            immFolQ : Value -> Value
            immFolDB (~immPreDB): Value -> Value
            lookup : Value -> Record}
 inv NonEmpty {some Root.attQ && some Root.attDB}

 inv WC1 {no a | WildCard in a.valDB}
 inv WC2 {no WildCard.attQ}
 inv WC3 {no WildCard.recDB}

 def immFolQ {all v | v.immFolQ = v.attQ.valQ}
 inv Q1 {no Root.~valQ}
 inv Q2 {all v : Value - Root | some v.attQ -> some v.~valQ}
 inv Q3 {all a | some a.~attQ -> some a.valQ}
 inv Q4 {all a | some a.valQ
                -> (one a.valQ && one v | a  in v.attQ)}
 inv Q5 {no v | v  in v.+immFolQ}

 def immFolDB {all v | v.immFolDB = v.attDB.valDB}
 inv DB1 {no Root.~valDB}
 inv DB2 {all v : Value - Root | some v.attDB -> some v.~valDB}
 inv DB3 {all a | some a.~attDB -> some a.valDB}
 inv DB4 {all a | some a.valDB -> one v | a  in v.attDB}
 inv DB5 {no v | v  in v.+immFolDB}

 inv Add1 {no Root.recDB}
 inv Add2 {all v | no v.~valDB -> no v.recDB}
 inv Add3 {all v | some v.~valDB && no v.attDB -> some v.recDB}
 inv Add4 {all v | all r: v.recDB | no v1: v.+immPreDB | r  in v1.recDB}
 inv Add5 {no v1,v2 | v1 != v2 && (some v1.recDB & v2.recDB) &&
                      some v1p:v1.*immPreDB, v2p:v2.*immPreDB |
                          v1p != v2p && v1p.~valDB = v2p.~valDB}

 cond indexedSubset(r:Record,v:Value)
    {all a : v.attQ & v.attDB, v1 : a.valQ & a.valDB |
                                        r in v1.lookup + v.recDB}
 cond indexedSuperset (r : Record, v:Value)
    {all v1 : v.immFolDB | v1.recDB + v.recDB  in r}

 inv Lookup1 {all v:Value - WildCard |
                  (no v.attQ || no v.attDB) -> v.lookup = v.recDB}
 inv Lookup2 {all v:Value - WildCard |
         (some v.attQ && some v.attDB)
         -> (indexedSubset(v.lookup,v) &&
             no r : Record - v.lookup | indexedSubset(v.lookup+r,v))}
 inv Lookup3 {all v:WildCard | some v.~valQ.valDB
       -> (indexedSuperset(v.~immFolQ.lookup, v.~immFolQ) &&
           no r : v.~immFolQ.lookup |
            indexedSuperset(v.~immFolQ.lookup - r, v.~immFolQ))}}
```

Figure 3. Alloy model of INS name resolution

Similar invariants (*DB1-5*) also hold for validating the database.

Add-Name is safely abstracted by modeling the constraints it imposes on the database. *Add1* just requires that no service satisfies all demands. *Add2* says that a value not appearing in the database does not contain any records. When an advertisement is added to a database, its leaf nodes contain the corresponding record (*Add3*). Moreover, since only leaf nodes contain that record any ascendant nodes do not contain it (*Add4*). Finally, since each attribute has exactly one corresponding value in an advertisement, sibling values do not share a record (*Add5*).

Lookup-Name is modeled by three mutually exclusive invariants. *Lookup1-2* handle the case without wild cards, while *Lookup3* adds the functionality for handling them.

Lookup1 says that if v corresponds to a leaf value in the database or to a leaf av-pair in the query, then *v.lookup* is just the records contained in that value. This does imply that *Lookup-Name* may return records even if the database is less specific than the query. This is so because missing attributes are treated as wildcards by INS inventors.

Lookup2 is more subtle and uses the auxiliary condition *indexedSubset*, which provides the functionality of taking the intersection over a collection of sets. It uses the simple mathematical equivalence that a set, S, equals the intersection of an indexed collection of sets S_i if and only if S is a subset of each S_i and addition of any new element to S violates some subset constraint. *Lookup3* is similar, and makes use of *indexedSuperset*, which behaves as a dual to *indexedSubset*.

4. Analyzing the model

We analyzed our model using the Alloy Constraint Analyzer. The Alloy Constraint Analyzer [7,8] is a tool for analyzing object models with a variety of uses. At one end, it acts as a support tool for object model diagrams, checking for consistencies of multiplicities and generating sample snapshots. At the other end, it embodies a lightweight formal method in which subtle properties of behavior can be investigated. Its input language Alloy supports a declarative description of state and behavioral properties, by conjoining constraints. An Alloy model can, therefore,

```
cond QExistsDB {all a,v | a.valQ= v
                        -> (a.valDB = v && a.~attQ = a.~attDB)}
cond AlreadyAdded
   {QExistsDB &&
    some r | all v | some v.~valQ && no v.attQ -> r in v.recDB}
cond IsLeafAVPair(a:Attribute, v:Value){a.valQ=v && no v.attQ}
cond IsValidRecord(r:Record)
   {all a,v | IsLeafAVPair(a,v) ->
           r in v.recDB + v.+immPreDB.recDB}
cond SomeRecordReturned {some Root.lookup}
cond AllRecordsReturnedAreValid
   {all r | r in Root.lookup -> IsValidRecord(r)}
cond AllValidRecordsAreReturned
   {all r | IsValidRecord(r) -> r in Root.lookup}

assert LookupOK1 {AlreadyAdded -> SomeRecordReturned}
assert LookupOK2 {AlreadyAdded -> AllRecordsReturnedAreValid}
assert LookupOK3 {AlreadyAdded -> AllValidRecordsAreReturned}
```

Figure 4. Three basic theorems

be developed incrementally, with the Alloy Constraint Analyzer investigating whatever has been developed so far.

Alloy is not a decidable language, so its constraint analyzer cannot provide a sound and complete analysis. Instead, it conducts a search within a finite *scope* chosen by the user that bounds the number of elements in each primitive type. Here, for example, an analysis of a theorem about *Lookup-Name* for a scope of 4 would account for every possible lookup in which the query and database are constructed from at most 4 attributes, 4 values and 4 records. Needless to say, this is a huge space that could not be covered by traditional simulation methods.

The Alloy Constraint Analyzer's output is either an *instance* – a particular state or transition – or a message that no instance was found in the given scope. When checking an assertion, an instance is a counterexample to the theorem. When exercising an invariant or operation, an instance is a demonstration of consistency.

Theoretically, when no instance is found, the user is not entitled to infer anything. However, in practice, if an instance exists, there is one usually in small scope. So when none is found, it is quite likely that an assertion holds, or that an invariant is inconsistent.

The Alloy Constraint Analyzer works by translating the problem to be analyzed into a (usually huge) Boolean formula. This formula is handed to a SAT solver, and the solution is translated back by the Alloy Constraint Analyzer into the language of the model. The algorithm is described in [7].

The Alloy Constraint Analyzer comes with a suite of public domain SAT solvers including SATO [14] and

```
assert LookupOK4 {no Root.attDB & Root.attQ -> no Root.lookup}

cond NoRecordOnAVMismatch
   {all a,r | (a in Root.attQ & Root.attDB && no a.valQ & a.valDB
           && r in a.valDB.recDB && not a.valQ=WildCard)
           -> not r in Root.lookup}
assert LookupOK5 {NoRecordOnAVMismatch}

cond SomeCommonAV
  {some a | a in Root.attQ & Root.attDB && some a.valQ & a.valDB}
assert LookupOK6{SomeCommonAV -> NoRecordOnAVMismatch}

cond QMatchesDB {QExistsDB && all a | some a.~attQ & a.~attDB}
assert LookupOK7 {QMatchesDB -> some Root.lookup}

cond RConformsQ (r:Record)
      {all a,v | IsLeafAVPair(a,v)
              -> some v1 | r in v1.recDB && v in v1 + v1.immPreDB}
assert LookupOK8
       {all r | QExistsDB && RConformsQ(r) -> r in Root.lookup}

op RemoveWildCard {
  all a | a.valQ != WildCard -> a.valQ' = a.valQ
  all a | a.valQ = WildCard -> no a.valQ'
  all v | all a:v.attQ | a.valQ = WildCard  -> v.attQ' = v.attQ - a
  all v | all a:v.attQ | a.valQ != WildCard -> v.attQ' = v.attQ
  all v | no v.attQ -> no v.attQ'
  all v | v.attDB' = v.attDB && v.recDB' = v.recDB &&
       v.immFolDB' = v.immFolDB && v.immPreDB' = v.immPreDB
  all a | a.valDB' = a.valDB
}
assert LookupOK9 {RemoveWildCard  -> Root.lookup= Root.lookup'}
```

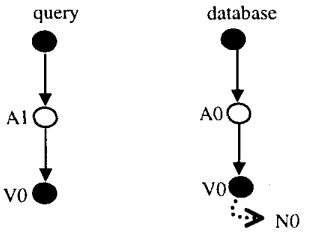
Figure 5. More theorems

RelSAT [3], whose parameters can be adjusted within the Alloy Constraint Analyzer itself.

4.1. When INS works

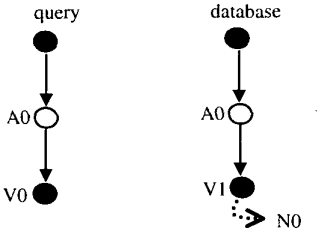
We check the validity of 3 theorems (Figure 4) concerning the soundness of INS using the Alloy Constraint Analyzer. First of all we assume that the intentional name being resolved is added to the database using the *Add-Name* operation, and it exists in the database at the time of resolution. Then we check:

- that *Lookup-Name* returns at least some record (*LookupOK1*);
- that all records returned by *Lookup-Name* are valid (*LookupOK2*);
- that all valid records are returned by *Lookup-Name* (*LookupOK3*).



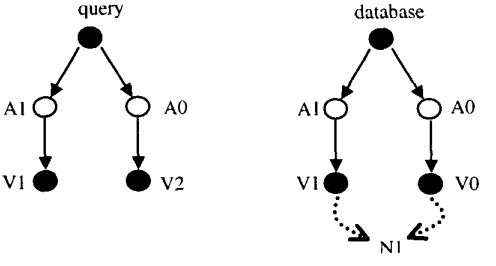
Lookup-Name(database, query) = {N0}

Figure 6. Counterexample to LookupOK4



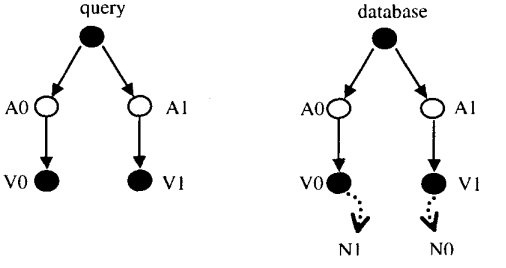
Lookup-Name(database, query) = {N0}

Figure 7. Counterexample to LookupOK5



Lookup-Name(database, query) = {N1}

Figure 8. Counterexample to LookupOK6



Lookup-Name(database, query) = { }

Figure 9. Counterexample to LookupOK7

For now, we consider a record to be valid if and only if it is included in the set of records of all leaf values, that match those of the query, or their parents. A more general treatment is presented in Section 4.3, where we argue that the validity semantics in INS need a reexamination to make the naming more versatile.

In all three cases, the Alloy Constraint Analyzer completes its search without finding a counterexample. This gives us confidence that INS's resolution mechanism is sound when the intentional name being resolved appears exactly in the database due to an advertisement.

4.2. Problems with INS

We uncovered several flaws in the *Lookup-Name* algorithm once we relaxed the condition that the intentional name appear in its entirety (Figure 5). This came to us as a surprise, since generality and expressiveness were two of the primary concerns in the design of INS.

Our first test (*LookupOK4*) checks the claim that if the database has no attributes in common with the query at the top level, then *Lookup-Name* should return the empty set. The Alloy Constraint Analyzer quickly generates a counterexample. Figure 6 shows a graphical illustration of this counterexample. As it happens, the INS algorithm returns all records in the database if there are no matching attributes at the top level! This problem arises since the algorithm tries to model missing attributes as being equivalent to wild cards. As we see below, this putative correspondence gives rise to several other flaws.

Our next assertion (*LookupOK5*) tests the case in which there is some common top level attribute. Naturally, we believed that if a value in the database had no matching av-pair in the query, while its parent attribute had one, then the records of this value would not appear in the result of *Lookup-Name*. The Alloy Constraint Analyzer produces a counterexample to this that appears in figure 7.

This flaw has serious implications, since a client asking for a printer service could get back a camera! It arises because *Lookup-Name* does not handle a mismatch when comparing values (see figure 2). For example, if a query seeking a scanner service in building NE-43 is resolved in the database shown in figure 1, both N1 and N2 would be returned.

We then pose the same question under the assumption that the database and the query have a common attribute at top level and moreover one of the corresponding values also match (*LookupOK6*).

This time, not surprisingly though, the Alloy Constraint Analyzer disproves the assertion with the counterexample illustrated in figure 8. The root of this bug is the same as that illustrated by *LookupOK5*.

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

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

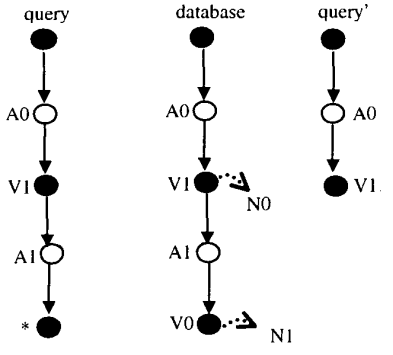


Figure 10. Counterexample to LookupOK9

We put this claim to the test in the case of queries as follows. We defined an Alloy operation *RemoveWildCard* that removes wildcards from a query. The operation mutates the query by removing the av-pair(s) containing wildcard(s), while maintaining the state of other av-pairs and the original database. Our assertion, *LookupOK9*, says that the effect of a lookup should be the same before and after this mutation.

This assertion is not valid; a counterexample is shown in Figure 10. Before mutation (ie, with wildcards) the name records *N0* and *N1* are returned; after mutation (ie with omission in place of wildcards), only *N0* is returned.

For the case of advertisements, one of our analyses (*LookupOK7*) already disproves the claim (Figure 9). It also points out the difference between the intentional name simply being 'present' in the database by virtue of a correspondence in the data structures and it having been 'added' to the database by an advertisement. If the contested equivalence were to hold, *Root.lookup* should be {N0,N1}, but it is empty.

The problems exposed by *LookupOK4*, *LookupOK5* were already known to the developers of INS. The other problems were apparently not known, and represent bugs not only in the description of INS but also in its implementation.

4.3. Naming Issues

It seems reasonable to treat a service that has no conflicting functionality to what a client seeks, but specialises some of the av-pairs in the query, as a valid result in resolving that query. For instance, if an application requires a picture of the Whitehouse and does not care about any particular area (or does not have sufficient information to express that), a service that advertised as providing one in

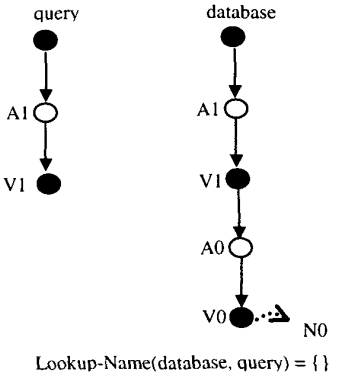


Figure 11. Counterexample to LookupOK8

the West Wing of the Whitehouse should certainly be treated as valid.

A strong reason to allow such conformance is that it is the service providers who have the exact details of the services that they provide, whereas clients who are only seeking services need some additional flexibility in forming their queries.

We test the behavior of INS in such a situation by formulating the assertion *LookupOK8*. *RConformsQ* defines a record to conform to a query if it appears in the values in the database corresponding to each leaf av-pair in the query or one of their descendant values. *LookupOK8* only tests if all the records that conform to the query according to this definition appear in the result of *Lookup-Name*. The Alloy Constraint Analyzer generates a counterexample (Figure 11) showing that such a record is not necessarily returned.

Treating missing attributes as wildcards is certainly one way to add this flexibility but incorporating it correctly in INS requires a significant alteration to the implementation of *Lookup-Name*.

Table 1 summarizes the analyses we performed on INS.

5. Performance

An implementation of INS appears in [12]. The Java code is about 2300 lines. About 900 of these lines constitute the testing code. Our model of the core functionality of the naming scheme of INS consists of 50 lines of Alloy. The theorems that we tested consist of another 44 lines of Alloy code.

The counterexamples appearing in section 4.2 were generated by the Alloy Constraint Analyzer in no more than 6 sec and required at most 4 elements in any domain with the exception of *LookupOK6* that needed 5 elements in the *Value* domain.

Moreover, the Alloy Constraint Analyzer analysis of the three theorems in section 4.1 took between 14 sec and

Table 1. Summary of analyses

Assertion checked	*Result*
Lookup-Name returns something	Yes
Every record returned is valid	Yes
Every valid record is returned	Yes
No record if no attributes match	No
No record if no values match	No
Record returned has matching value	No
Missing attribute ~ wildcard (queries)	No
Missing attribute ~ wildcard (advertisements)	No

30 sec using a scope of 5 in each domain and generated no counterexamples. It comes as no surprise that it is more time consuming to check valid assertions since the entire space within the specified scope must be exhausted.

These results are tabulated in Tables 2 and 3. A 300 MHz Celeron processor with 128 MB of memory was used to perform all analyses.

We were able to generate counterexamples without incorporating the wildcards, which were added when *RemoveWildCard* was introduced. This only emphasizes one of the various uses of incremental modeling.

6. Related Work

The Alloy Constraint Analyzer has not been used previously for the analysis of a recursive algorithm of this sort. We have recently recast a model of COM originally written in Z into Alloy, and shown that its analysis can be automated [9].

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 [5] provide similar exhaustive search to the Alloy Constraint Analyzer. 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 rudimentary data structures, and are not designed for the kind of structural complexity of this problem. Wing and Vaziri-Farahani [16] use SMV [17] to verify cache coherence protocols by abstracting away data structures.
- Theorem provers such as PVS [10] can, unlike the Alloy Constraint Analyzer, 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.

We view the Alloy Constraint Analyzer as complementary to these other tools. A theorem prover, for example, might be used to prove a theorem after the Alloy Constraint Analyzer analysis has failed to find counterexamples in a reasonable scope.

Table 2. Exhaustive search performance

Scope / *Invariant*	3	4	5
LookupOK1	1 s	3 s	17 s
LookupOK2	1 s	2 s	14 s
LookupOK3	0 s	4 s	30 s

Table 3. Counterexample detection performance

Invariant	*Scope* *Val*	*Att*	*Rec*	*Time*
LookupOK4	3	2	1	0 sec
LookupOK5	4	1	1	0 sec
LookupOK6	5	2	1	6 sec
LookupOK7	4	2	2	1 sec
LookupOK8	4	2	1	1 sec
LookupOK9	4	2	2	4 sec

7. Conclusions and Future Work

Constructing and analyzing a model of an intentional naming scheme exposed a number of subtle problems in its design, and showed that one of the basic intuitions held by its designers that motivated aspects of the design was in fact false.

Our original model consisted of six domains and was about 120 lines long. Its structure corresponded closely to the Java implementation, which naturally leads us to inquire whether such a model might be extracted automati-

cally. Using our tool we succeeded in trimming it down to three domains and less than half its original length. The final model was about one twentieth of the size of the implementation of the *Lookup-Name* operation and its test drivers.

Moreover, we were able to formulate the operation using just one parameter. This was so because the first call to *Lookup-Name* only involves roots, and subsequent recursive calls are always made on matching values. This simplification could be carried over into the implementation.

One of the limitations of our new model is that it lacks the capability of representing repeated values and attributes in the database or the query. So if the behavior of some name resolution function is erratic only when multiple nodes have the same value, it would go undetected in the new model. Nonetheless, with this limitation we were able to greatly expedite the Alloy Constraint Analyzer analysis.

Formulation of Alloy invariants that capture the behavior of *Lookup-Name* required some subtle analysis of the algorithm since Alloy does not currently support sequential operations.

Another feature that we would like to add to Alloy is to re-use constraints for similar data structures. For example, in figure 3, despite the very similar representation of query and database we required both invariants Q1–5 and DB1–5.

This work was carried out when the counterexamples produced by the Alloy Constraint Analyzer were textual. That required tedious conversions for graphical illustrations. However, the current version of our tool automatically generates graphical counterexamples isomorphic to those shown in this paper.

We believe that the use of this kind of lightweight modeling 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.

The semantics of a naming scheme such as this is a subtle issue. We believe we can extract necessary properties for the soundness of a general purpose intentional naming scheme from our model, and we plan to pursue this further.

Acknowledgements

We would like to thank William Adjie-Winoto for explaining INS to us and giving us useful feedback.

References

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The Design and Implementation of an Intentional Naming System. *Proc. 17th ACM SOSP*, Kiawah Island, SC. Dec. 1999.

[2] Sten Agerhold, and Peter Gorm Larsen. *The IFAD VDM Tools: Lightweight Formal Methods*. FM-Trends 1998: 326-329.

[3] R.J. Bayardo Jr., and R. C. Schrag. Using CSP look-back techniques to solve real world SAT instances. *Proc. 14th National Conf. on Artificial Intelligence*, 203–208, 1997.

[4] John V. Guttag and James J. Horning. Formal Specification as a Design Tool. *Proc. Conf. on Principles of Programming Languages* (POPL 80), Las Vegas, Nevada, 1980, pp. 251–261.

[5] Gerard J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering, Special issue on Formal Methods in Software Practice*, Volume 23, Number 5, May 1997, 279-295.

[6] Daniel Jackson. *Alloy: A Lightweight Object Modeling Notation.* Technical Report 797, MIT Laboratory for Computer Science, Cambridge, MA, February 2000.

[7] Daniel Jackson. Automating First-Order Relational Logic. To appear, *Proc. Foundations of Software Engineering,* San Diego, California, November 2000.

[8] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: The Alloy Constraint Analyzer. *Proc. International Conference on Software Engineering,* Limerick, Ireland, June 2000.

[9] Daniel Jackson, and Kevin Sullivan. COM Revisited: Tool-Assisted Modeling and Analysis of Software Structures. To appear, *Proc. Foundations of Software Engineering,* San Diego, California, November 2000.

[10] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering,* 21(2):107-125, February 1995.

[11] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison Wesley Object Technology Series, 1998.

[12] Elliot Schwartz. *Design and Implementation of Intentional Names.* S.M. Thesis, MIT Laboratory of Computer Science, Cambridge, MA, June 1999.

[13] J. Michael Spivey. *The Z Notation: A Reference Manual.* Second ed, Prentice Hall, 1992.

[14] Hantao Zhang. SATO: An Efficient Propositional Prover. *Proc. International Conference on Automated Deduction* (CADE-97).

[15] Daniel Jackson and Jeannette Wing. Lightweight Formal Methods, *IEEE Computer*, April 1996.

[16] Jeanette Wing and Mandana Vaziri-Farahani. *A Case Study in Model Checking Software Systems.* Technical Report CMU-CS-96-124, Carnegie Mellon University, Pittsburgh, PA.

[17] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD Thesis, Carnegie Mellon University, 1992. CMU-CS-92-131.