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

by

Sarfraz Khurshid

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000

Author ....
Department of Electrical Engineering and Computer Science
May 5, 2000

Certified by ....
Daniel Jackson
Associate Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

To Nasim, Nighat, Ahmad, Mehreen, Maimoon
and my (late) uncle Khurshid

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

by

Sarfraz Khurshid

Submitted to the Department of Electrical Engineering and Computer Science
on May 5, 2000, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

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 Alcoa, a fully automatic simulation and checking tool. In doing so, we exposed several serious flaws in both the algorithms of INS and the underlying naming semantics. We were also able to characterize conditions under which the existing INS framework works correctly, and evaluate proposed fixes.

We also discuss semantics issues that arose through our analysis with Alcoa and need to be resolved in order to develop a scheme in which applications simply describe their needs in an intentional name that is later resolved by the network infrastructure. This analysis also enabled us to characterize essential properties of such a scheme.

Thesis Supervisor: Daniel Jackson
Title: Associate Professor

Acknowledgments

I am extremely grateful to my advisor Daniel Jackson. He has greatly helped me in both the technical and nontechnical matters. Without his insight, suggestions, and excitement, this work would never have taken place. Daniel also helped me with the writeup of the paper on which this thesis is based. It is remarkable how patiently Daniel can rewrite a section until it has the quality he expects.

I am privileged to be part of the stimulating and friendly environment of the Software Design research group of the MIT Laboratory for Computer Science. I would like to thank all the members of the group, both past and present, for making it a great place to work. Many thanks to Ilya Shlyahkter, and my officemates, Ian Schechter, Mandana Vaziri, and Allison Waingold.

I would also like to thank William Adjie-Winoto for the discussions we had on INS.

I must also thank my flatmates Darko Marinov and Victor Viteri for making our apartment such a pleasant place to live.

Finally, I want to thank my parents, Nasim and Nighat, my brothers, Ahmad and Maimoon, and my dearest sister Mehreen, for their love, encouragement, and help, which kept me going during the more difficult times.

Contents

List of Figures

List of Tables

Chapter 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 and 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) [2] is a recently developed framework that provides this functionality. It comprises applications (clients and services) and name resolvers, which respond to queries from clients.

Like IP routers [21] or conventional name servers [26], 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 just 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, while *Get-Name* is used to disseminate information amongst resolvers.

1.2 Formal modeling

The design of a software structure like INS needs careful evaluation. In the past, numerous other structures of similar complexity have been found with significant design flaws. For example, Garlan et al. found flaws in the High Level Architecture (HLA) for distributed, component-based simulation [4], while Sullivan et al. discovered problems in Microsoft's Component Object Model (COM) [23].

Among the basic properties that INS (or any naming scheme for that matter) must satisfy, foremost is that the name resolution mechanism should not return services that have conflicting functionality to what is requested. Another essential feature is that if there are advertised services with the required elements then clients asking for them should obtain them. It is also crucial that, for a given query, an existing advertisement should either be valid or invalid. More specifically, it should not be the case that a new advertisement invalidates an existing one from a different service. Notice that this could happen due to a flaw in the addition algorithm or the semantics of the naming scheme.

With regards to INS in specific, there are various other properties that need to be tested. These include published claims made by the inventors of INS, both about the behavior of its algorithms and semantics. One of these claims concerns the notion of wild-cards, and it is asserted that a missing attribute corresponds to a wild-card, both for queries and advertisements. This correspondence adds a greater degree of flexibility and expressibility to the naming scheme. However, it is a subtle claim and its consistency with the semantics and algorithms of INS needs to be evaluated carefully.

In order to evaluate these properties, formal modeling and analysis of the structures involved are critical. Unfortunately, the scarceness of tools for rapid cycle-time, interactive, formal modeling and analysis of such structures makes modeling itself difficult and error-prone.

Alcoa [14] is a tool for fast, interactive analysis of models written in the Alloy [11] language - a lightweight notation for expressing relational models of software structures. This thesis explains how we used Alloy and Alcoa to expose flaws in INS and explore variants of its design.

The focus of our model is the most interesting operation of INS, *Lookup-Name*, while accounting for calls to *Add-Name* by characterizing legal configurations of the resolver with suitable invariants. Alcoa was used interactively to refine our model to only 55 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:

- 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 evaluate published claims about the properties of wild-cards, and show these to be false. In all cases Alcoa generates counterexamples showing a query and a database state that violate the expected property.

- Use of this tool also enabled us to establish conditions under which the current INS algorithm for name resolution returns correct results.

- We evaluate the proposed fixes of INS inventors to two of the defects, and use Alcoa to prove that these fixes do not work.

- From our analysis, we are able to raise naming semantics issues that are relevant to any intentional naming scheme. Finally, we outline essential properties of a general intentional naming scheme.

We believe this work is significant for two reasons. First, it realizes the vision of Guttag and Horning [8], 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.

We further believe that this kind of lightweight approach to formal methods as advocated in [16] has a promising future, since it can detect errors prior to implementation, including structural flaws that are particularly hard to correct later. The key leverage in this analysis was provided by the Alcoa tool, which made it possible to gain confidence in the model, root out modeling errors quickly, and check theorems without the need to construct proofs.

1.3 Organization of the thesis

This thesis is organised as follows. Chapter 2 presents an overview of INS as described in [2]. Chapter 3 introduces some fundamental components of the Alloy language and the Alcoa tool. Chapter 4 presents a formal Alloy model of INS and outlines how to extract such models from Java code with similar features. This model is then substantially simplified. In Chapter 5, Alcoa is used to analyze the resulting model, and reveal several flaws in INS. Chapter 6 evaluates the cost of this model, in the relative sizes of model to code, and the time performance of the tool. Chapter 7 discusses some subtle semantics issues that are relevant to any intentional naming scheme, and presents an attempt to extract essential properties of a general intentional naming scheme. Chapter 8 discusses related and future work, and we conclude there.

Chapter 2

Intentional Naming System

What's in a name? that which we call a rose
By any other name would smell as sweet
– Romeo and Juliet (2:2)

INS consists of two distinct components, namely the applications and the *Intentional Name Resolvers* (INRs). Applications may be clients or services with services providing the functionality or data required by clients. The clients use INRs to route their requests to appropriate locations.

When an INR receives a lookup request it decides whether to resolve or forward it, based on the specific service requested by the client application. If the application chooses *early binding*, a list of IP-addresses conforming to the name is returned. Alternatively, a client may opt for *late binding* in which case the INR instead forwards the name with its application payload directly to the services (end-nodes). This integration of name resolution and message routing allows INS to adapt quickly to changes in the 'best' network location of a service. However, this feature is not relevant to our model of the core intentional naming architecture in INS.

INRs form a decentralized network and configure themselves into a spanning tree overlaying the topology of the network. They maintain a periodically updated mapping between service descriptions and their network locations, thus eliminating the need for explicit de-registration of services. Self-configuration of INRs and periodic advertisements of services make INS an easily configurable architecture with a high degree of robustness. However, they too do not form an integral part of its naming mechanism.

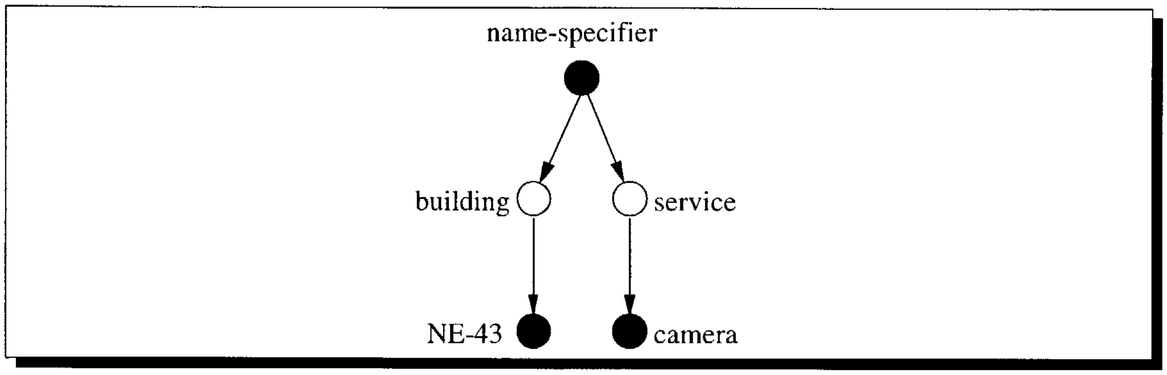


Figure 2-1: Name Specifier

The core of the naming scheme of INS constitutes the implementation of intentional names, and their advertisement and resolution. Intentional names are implemented in INS using *name-specifiers*. A name-specifier (Figure 2-1) is an arrangement of alternating levels of *attributes* and *values* in a tree structure. In Figure 2-1, 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 wild-card may be used in place of a value to show that any value is acceptable. 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 name-specifier in Figure 2-1, thus, describes an object in building NE-43 that provides a camera service.

An INR stores its information in a database called a *name-tree*. This database maps names to *name-records*, which include the IP addresses of services advertising the name. A name-tree can be viewed as a superpositioning of many name-specifiers, storing the correspondence between name-specifiers and name-records. Figure 2-2 shows an example name-tree that stores two objects, one (i.e. $R0$) that provides a camera service in NE-43 and the other one (i.e. $R1$) that provides a printer service in the same building.

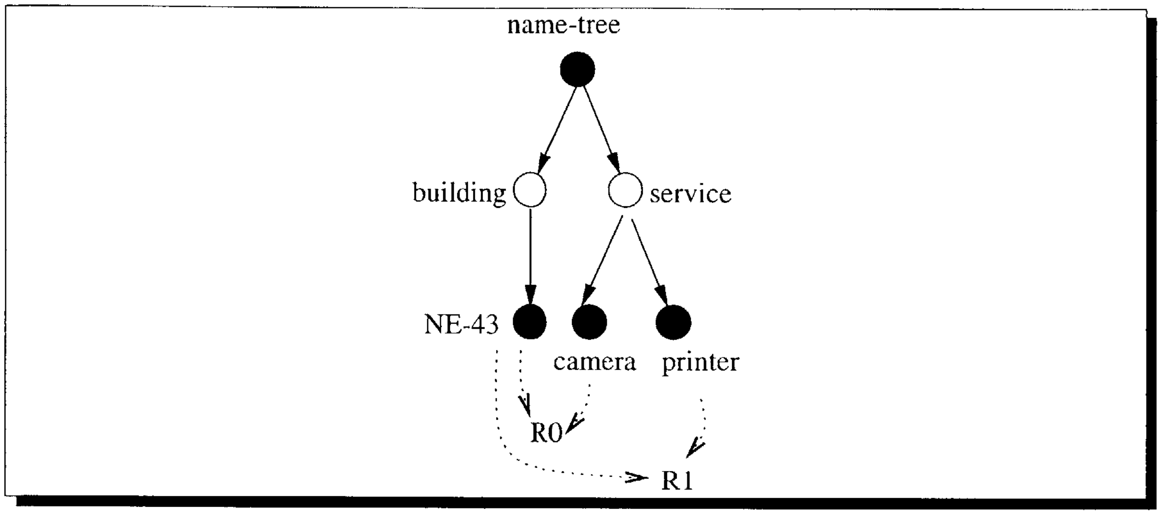


Figure 2-2: Name Tree

Analogous to the attributes and values in a name-specifier, a name-tree also has two fundamental building blocks, an *attribute-node* and a *value-node*. Like values, a value-node can have several attribute-nodes as its children that provide more specific classifications of that value. However, in contrast with attributes that have a unique corresponding value, an attribute-node can have several value-nodes as its children, each representing a distinct value the name-tree recognizes. A value-node that corresponds to a leaf av-pair of an advertised name-specifier also contains a pointer to the relevant name-record. In Figure 2-2 this is represented by broken arrows.

INRs interact with name-trees in two key ways: resolving name-specifiers to name-records and disseminating information about name-specifiers amongst themselves. The name-records for a name-specifier are retrieved using the *Lookup-Name* operation. An algorithm for this operation is given in pseudocode in the published description of INS [2], and is replicated in Section 4.4.

Resolving the name-specifier in Figure 2-1 in the name-tree of Figure 2-2 results in the name-record $R0$. Of the two name-records $R0$ and $R1$ in the name-tree, only $R0$ is returned, since the value of attribute 'service' sought by the client does not match that provided by $R1$. This is because $R1$ provides a printer service whereas the client seeks a camera object.

The *Lookup-Name* algorithm makes a series of recursive calls, but does not backtrack. Each call reduces the set of possible name-records by intersecting it with those contained in matching leaf nodes. The inventors of INS claim [2] that in the execution of the algorithm 'omitted attributes correspond to wild-cards'. Our analysis establishes this to be false (Section 5.2).

When a service advertises its availability to an INR, it is included in the name-tree stored by that INR using the *Add-Name* algorithm.

To periodically update adjacent INRs about new or expired services, an INR extracts name-specifiers from its name-tree using the *Get-Name* algorithm. This algorithm traces upwards from leaf nodes of a name-tree, reconstructing name-specifiers on its way to the root, and grafting along any previously reconstructed pieces.

In summary, INS achieves its expressiveness using a simple naming language based on attributes and values. It integrates name resolution and message routing to allow applications to be responsive to mobility and performance changes, and uses periodic service advertisements and soft-state name dissemination protocols between replicated resolvers to achieve robustness.

Chapter 3

Modeling tools

Our formalization of the core model of the naming scheme of INS (Figure 4-6) is written in Alloy, a first-order notation that attempts to combine the best of features of Z [22] and UML [19]. 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.

3.1 Alloy

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.

We describe the basic components of Alloy below (referring to the model in Figure 4-6 to explain the notions as they are introduced.) A detailed rationale for Alloy's design appears in [11].

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 name-specifier and a name-tree. *Record* models the set of name-records that exist in a name-tree. 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 name-records in a particular configuration, not the set of all imaginable name-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

$$r : S\ m \rightarrow T\ n$$

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 $recNT$, for example, maps at least one $Value$ to each *Record*, which informally means that all name-records appear in some value-node. 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 i.e. 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 wild-cards.

When several set components are declared together, the collection may be marked as *disjoint*, thus

$$disjoint\ Root, WildCard : fixed\ Value!$$

declares *Root* and *WildCard* to be disjoint (fixed) subsets of *Value* of size 1.

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: that is, navigating one or more steps of r. 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

$$\mathit{Root.attNS}\ \&\ \mathit{Root.attNT}$$

for example, denotes the set of attributes common to both the name-specifier and the name-tree 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: $s\ \mathit{in}\ t$, 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.

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

$$\mathit{no}\ \mathit{Root.attNS}\ \&\ \mathit{Root.attNT}$$

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

$$\mathit{all}\ v \mid v.\mathit{immFolNS} = v.\mathit{attNS}.\mathit{valNS}$$

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

$$\mathit{all}\ v : \mathit{Value} \mid v.\mathit{immFolNS} = v.\mathit{attNS}.\mathit{valNS}$$

In our Alloy models of INS that follow, 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 name-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.

Alloy also has a graphical notation for visualization of models. In the graphical representation, a box labeled S, denotes a set of objects, which are indivisible things, without notions of state or methods. Arrows with open heads denote relations. The set declaration $S : T\ m$ is represented as an arrow with a triangular head from a box marked $S\ m$ to a box marked T. An arrow with a triangular head and two tails represents disjoint subsets. If a set is fixed it has a vertical stripe down both the left and right-hand sides of the box.

3.2 Alcoa

We analyzed our model using Alcoa. Alcoa [14] 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, be developed incrementally, with Alcoa investigating whatever has been developed so far.

Alloy is not a decidable language, so Alcoa 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 name-specifier and name-tree are constructed from at most 4 attributes, 4 values and 4 name-records. This is a huge space (comprising about 2^{100} cases) that could not be covered by traditional simulation methods.

Alcoa's use requires first the creation and compilation of an Alloy model. The compilation takes a few seconds and finds superficial flaws, such as type errors. The user then selects a *schema* that is a paragraph of the model to be analyzed and starts a run. Alcoa'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. The user may then choose to edit the model, recompile and rerun, or to investigate the same schema further, by changing the scope or adjusting the solver parameters.

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.

Even in small scopes, the number of cases to consider is usually quite large. A relation in the scope of k has $2^{k \times k}$ possible values. For example, a model with only 3 relational state components in a scope of 3 thus has about a billion states. Clearly several of these will be ruled out by the constraints, and the search mechanism will prune away large parts of the space. For instance, in checking that an operation preserves an invariant, the search might exclude most of the post-states that do not violate the invariant, thus considering only 'bad' executions of the operation, effectively executing it backwards. Hence, by preemptively ruling out large classes that are not to cause problems, this scheme is able to account for billions of possible executions of the operation.

Alcoa works by translating the problem to be analyzed into a (usually huge)

Boolean formula. This formula is handed to an off-the-shelf SAT solver, and the solution is translated back by Alcoa into the language of the model. The algorithm is described in [12], and we outline the basic steps below.

The first step involves two simple manipulations on the problem: conversion to negation normal form and skolemization. Next it is translated into a boolean formula for the chosen scope along with a mapping between relational variables and the boolean variables used to encode them. This boolean formula is constructed so that it has a model exactly when the relational formula has a model in the desired scope. The boolean formula is then converted to conjunctive normal form, which is the preferred input of most SAT solvers. Next, the boolean formula is presented to the SAT solver. If the solver finds a model, a model of the relational formula is then reconstructed from it.

Alcoa comes with a suite of public domain SAT solvers including SATO [28] and RelSAT [17], whose parameters can be adjusted within Alcoa itself. However, in our analysis of INS that follows, these parameters did not require any adjustments.

Chapter 4

Alloy model of INS

There are two primary challanges to constructing a model of INS in Alloy. First the semantics of INS are only described informally. Pseudo-code for the main algorithms is provided in [2], but the definitions of data structures involved are informal. Second, Alloy's built in support for representing operations does not handle recursive functions; Alloy has not previously been used to model algorithms of this nature.

A Java implementation of the naming scheme of INS, however, is given in [20]. Our main model of INS in Alloy (Sections 4.3 and 4.4) is independent of the data structures used in this implementation and could be constructed from only an informal description of the data structures and pseudo-code for essential procedures. Thus, the impatient reader may jump straight ahead to Section 4.3.

Below we begin our discussion by informally describing how to extract Alloy models from Java code with language features similar to those used in the INS implementation. Then we explain various transformations that are performed on this model to reduce its complexity.

4.1 An initial model

Figure 4-1 transcribes parts of the Java code in [20], that are relevant to construction of an Alloy model.

The basic idea behind the construction of *domain* and *state* in an Alloy model is

```
class Attribute{
        String attribute; ...}
class Value {
        String value; ...}
class NameRecordSet {...
        Vector nameRecords; // Vector of NameRecord...}
class AVPair {
        Attribute a;
        Value v;
        Vector children; // Vector of AVPair ...}
class NameSpecifier extends AVPair {...}
class AttributeNode {
        Attribute a;
        Vector children; // Vector of ValueNode ...}
class ValueNode {
        Value v;
        Vector children; // Vector of AttributeNode
        NameRecordSet routeSet; ...
        NameRecordSet lookup(AVPair n) {...} ...}
class NameTree extends ValueNode {
        Vector nameRecords; // Vector of NameRecord ...}
```

Figure 4-1: Relevant data structures from Java implementation

as follows. For each of the (six) `class` declarations that `extend object`, introduce a new domain. Then for each field in such a `class`, define a relation that maps an element of the `class` to relevant value(s) of that field; Finally, if `class A extends B`, include a subset constraint in the model stating $A : B$, and for each field of `A`, define a relation that maps an element of A to element(s) of the domain corresponding to the type of that field. [1]

Figure 4-2 shows this construction and Figures 4-3 and 4-4 anotate it using graphical notation of Alloy. Method `lookup` in `class ValueNode` is modelled by defining an indexed relation *lookup*; for each element *av* of *AVPair*, *lookup[av]* is a relation from *ValueNode* to *NameRecordSet*. Thus, *v.lookup[av]* models the function call

[1]It must be stressed that this is an informal description. It is certainly not an attempt to precisely define rules that automate extraction of succinct Alloy models from arbitrary Java code; something left for future work.

```
model INS1 {
 domain{Attribute, Value, AVPair, ValueNode, AttributeNode, NameRecordSet}
 state{
  attributeAV : AVPair? -> Attribute?
  valueAV : AVPair? -> Value?
  childrenAV : AVPair? -> AVPair
  NameSpecifier : fixed AVPair!
  attributeAN : AttributeNode? -> Attribute!
  childrenAN : AttributeNode? -> ValueNode+
  valueVN : ValueNode? -> Value?
  childrenVN : ValueNode! -> AttributeNode
  routeSet : ValueNode -> NameRecordSet
  NameTree : fixed ValueNode!
  nameRecords : NameTree -> NameRecordSet
  lookup[AVPair] : ValueNode -> NameRecordSet
  immFollowsVN(~immPrecedesVN) : ValueNode -> ValueNode
  WildCard : fixed Value!
 }
}
```

Figure 4-2: Domain and State in initial model

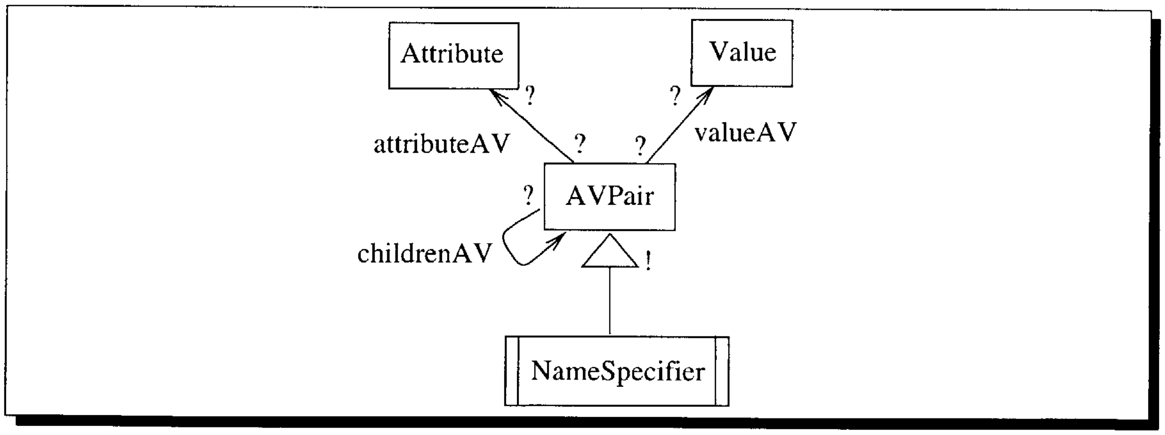


Figure 4-3: *NameSpecifier* in graphical Alloy

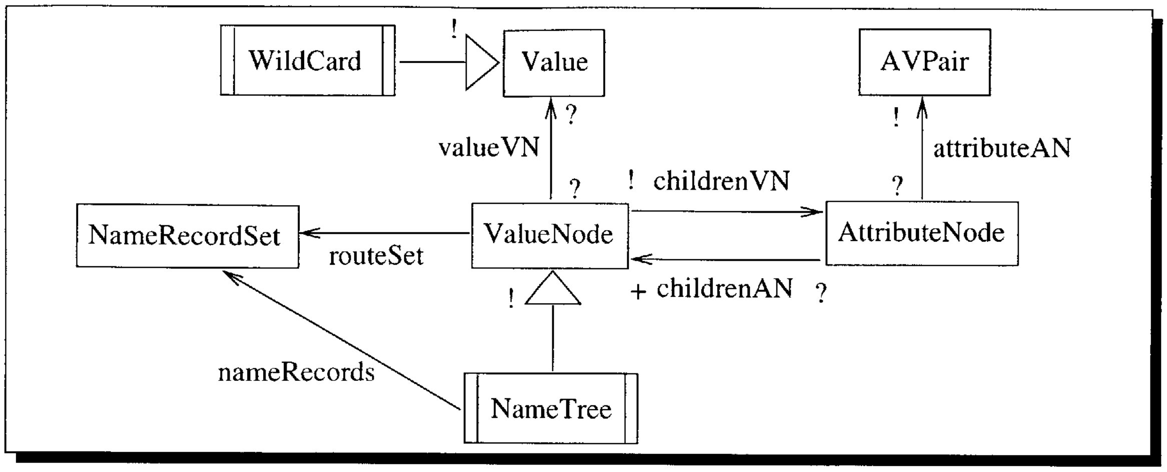


Figure 4-4: *NameTree* in graphical Alloy

`v.lookup(av)`.

The defined relation *immFollowsVN* maps a *ValueNode* to the possible *ValueNodes* its children *AttributeNodes* can take in the name-tree; *immPrecedesVN* is the transpose of *immFollowsVN*.

Wild-card is modeled as a *fixed Value*. This is slightly different from the Java implementation, which uses a flag.

4.2 Simplifying transformations

There are several simplifications that can be performed to the model above. We begin by noting that the relation *nameRecords* that explicitly stores the set of name-records in the name-tree is essentially redundant. This information can be stored indirectly by requiring that all elements of *NameRecordSet* must appear in the image of *routeSet*.

Recall that intuitively both name-specifier and name-tree are just alternating levels of values and attributes, arranged in a directed tree. We can use this intuition to remove several relations that appear in Figure 4-2 if we superimpose the two tree structures with each edge properly labeled, according to whether it belongs to the name-specifier or the name-tree. Moreover, we no longer need the indirection introduced by relations *attributeAV* and *valueAV* in the case of the name-specifier to access the attributes and values in it. Likewise, relations *valueVN* and *attributeAN* are not required to access elements of a name-tree.

It should be noted here that this simplification has a limitation. We can no longer represent repeated values or attributes in the name-specifier or the name-tree. This means that if some algorithm only breaks down when the same value or attribute appears multiple times in the name-specifier or the name-tree, then we would not be able to detect a flaw in that algorithm. In the case of *Lookup-Name* this is not a restriction since all the decisions it makes are based on local values observed and it is illegal to have sibling attribute (value) nodes that share an attribute (value).

Observe, further, that there is no need to have a separate root node each for the name-specifier and the name-tree. A single root can act as the root of both, with labeled edges pointing to the appropriate top level attributes.

Another thing to notice is that every recursive call to *Lookup-Name* is made by ensuring first that the values of the value-node and av-pair forming the parameters of the call are identical; both are `null` on the initial call. Hence there is no need to model *Lookup-Name* as an indexed relation.

Based on these observations we can remove the domains *AVPair*, *ValueNode*, and *AttributeNode* from our model

4.3 Model of INS in Alloy

The *domain* and *state* of our new model of INS are presented in Figure 4-6 and described below. Figure 4-5 illustrates this model using the graphical notation of Alloy, and replaces the model constructed earlier that appears in Figures 4-3 and 4-4.

The domain consists of three sets *Attribute*, *Value*, and *Record*[2].

Root and *WildCard* are distinct values, and *Root* acts as the root of both the name-specifier to resolve and the name-tree to search.

The relations *valNS* and *attNS* map an attribute to its child value and a value to its children attributes in the name-specifier, respectively. Thus the name-specifier is defined by the tuple (*attNS*,*valNS*). For example, the name-specifier in Figure 2-1 is

[2] *Record* is the same as *NameRecordSet* earlier, however we decided to simplify its name too!

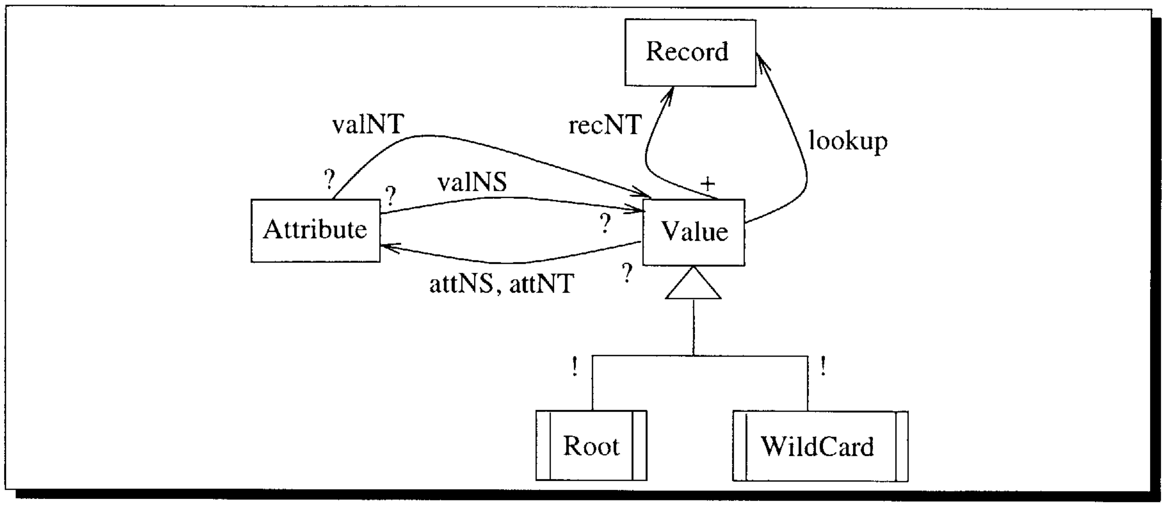


Figure 4-5: Graphical Alloy model of INS that replaces Figures 4-3 and 4-4

```
model INS {
 domain{Attribute, Value, Record}
 state{
  disjoint Root, WildCard : fixed Value!
  valNS : Attribute? -> Value?
  attNS : Value? -> Attribute
  valNT : Attribute? -> Value
  attNT : Value? -> Attribute
  recNT : Value+ -> Record
  immFolNS : Value -> Value
  immFolNT (~immPreNT): Value -> Value
  lookup : Value -> Record
 }
}
```

Figure 4-6: New domain and state of INS model

given by

$$\begin{aligned} attNS &= \{Root \rightarrow \{building, service\}\} \\ valNS &= \{building \rightarrow NE\text{-}43, service \rightarrow camera\} \end{aligned}$$

The relations *attNT* and *valNT* similarly map a value to its descendant attributes and an attribute to the possible values it can take respectively, but in the name-tree. The third relation that defines the name-tree is *recNT* which maps a value to name-records. The name-tree of Figure 2-2 would have

$$\begin{aligned} attNT &= \{Root \rightarrow \{building, service\}\} \\ valNT &= \{building \rightarrow \{NE\text{-}43\}, service \rightarrow \{camera, printer\}\} \\ recNT &= \{NE\text{-}43 \rightarrow \{R0, R1\}, camera \rightarrow \{R0\}, printer \rightarrow \{R1\}\} \end{aligned}$$

Thus, the name-tree is just a triple (*attNT*, *valNT*, *recNT*).

Defined relations *immFolNS* and *immFolNT* map a value to possible values its children attributes can take in the name-specifier and the name-tree respectively. For Figures 2-1 and 2-2

$$\begin{aligned} immFolNS &= \{Root \rightarrow \{NE\text{-}43, camera\}\} \\ immFolNT &= \{Root \rightarrow \{NE\text{-}43, camera, printer\}\} \end{aligned}$$

immPreNT is defined to be just the transpose of *immFolNT*.

The relation *lookup* models the *Lookup-Name* method, and maps each value v to a set of name-records; these records represent the return value of *Lookup-Name* when invoked on the value-node and the av-pair corresponding to v. Thus, *Root.lookup* is the result of resolving the name-specifier in the name-tree, and is $\{R0\}$ for Figures 2-1 and 2-2.

Figure 4-7 presents the constraints that model the wild-card value and the name-specifier, while Figure 4-8 contains the ones for the name-tree and *Add-Name* . They can be described as follows.

- The input name-tree and name-specifier are non-null (*NonEmpty*).
- *WildCard* is governed by the following three invariants:

- it does not appear in the name-tree (*WC1*);
- no attribute specializes it in the name-specifier (*WC2*);
- it does not contain any name-records (*WC3*).

- The structure of the name-specifier is constrained by the invariants:
  - no attribute maps to *Root* under the *valNS* relation, i.e. *Root* has no ascendants (*NS1*);
  - if a non-root value exists in the name-specifier, it has exactly one ascendant (*NS2*);
  - if an attribute exists in the name-specifier, it has exactly one descendant value (*NS3*), and one ascendant value (*NS4*);
  - the name-specifier data structure is acyclic (*NS5*); this is expressed using the transtive closure operator.
- Similar invariants (*NT1-5*) also hold for validating the name-tree.
- *Add-Name* is safely abstracted by modeling the constraints it imposes on the name-tree.
  - *Add1* just requires that no service satisfies all demands;
  - *Add2* says that a value not appearing in the name-tree does not contain any name-records;
  - when a name-specifier is added to a name-tree, its leaf nodes contain the corresponding name-record (*Add3*);
  - since only leaf nodes contain that name-record, any ascendant nodes do not contain it (*Add4*);
  - since each attribute has exactly one corresponding value in a name-specifier, sibling values do not share a name-record (*Add5*).

```
inv NonEmpty {some Root.attNS && some Root.attNT}

inv WC1 {no a | WildCard in a.valNT}
inv WC2 {no WildCard.attNS}
inv WC3 {no WildCard.recNT}

def immFolNS {all v | v.immFolNS = v.attNS.valNS}
inv NS1 {no Root.~valNS}
inv NS2 {all v : Value - Root | some v.attNS -> one v.~valNS}
inv NS3 {all a | some a.~attNS -> one a.valNS}
inv NS4 {all a | some a.valNS
                 -> (one a.valNS && one v | a  in v.attNS)}
inv NS5 {no v | v  in v.+immFolNS}
```

Figure 4-7: Constraints for wild-card and name-specifier

```
def immFolNT {all v | v.immFolNT = v.attNT.valNT}
inv NT1 {no Root.~valNT}
inv NT2 {all v : Value - Root | some v.attNT -> one v.~valNT}
inv NT3 {all a | some a.~attNT -> some a.valNT}
inv NT4 {all a | some a.valNT -> one v | a  in v.attNT}
inv NT5 {no v | v  in v.+immFolNT}

inv Add1 {no Root.recNT}
inv Add2 {all v | no v.~valNT -> no v.recNT}
inv Add3 {all v | some v.~valNT && no v.attNT -> some v.recNT}
inv Add4
    {all v | all r : v.recNT | no v1 : v.+immPreNT | r  in v1.recNT}
inv Add5 {no v1,v2 | v1 != v2 && (some v1.recNT & v2.recNT) &&
                     some v1p:v1+v1.+immPreNT, v2p:v2+v2.+immPreNT |
                          v1p != v2p && v1p.~valNT = v2p.~valNT}
```

Figure 4-8: Constraints for name-tree and *Add-Name*

4.4 Modeling *Lookup-Name*

Before we describe our model of *Lookup-Name* in Alloy, we explain its pseudo-code as it appears in [2] (Figure 4-9 transcribes it).

```
Lookup-Name(T,n)
   S <- the set of all possible name-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 name-records in the
                                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 name-records of Tv)
          else
             S <- S intersection Lookup-Name(Tv, p)
   return S union (the name-records of T)
```

Figure 4-9: *Lookup-Name* algorithm

Lookup-Name takes a name-tree and a name-specifier as its input parameters. It begins by initializing S to be the set of all possible name-records. Then, for each child av-pair in the name-specifier, it finds an attribute-node in the name-tree whose attribute is the same as that of the av-pair. If the av-pair contains a wild-card as its value, then the algorithm computes S' as the union of all name-records in the value-nodes that correspond to that attribute-node. If not, it finds the corresponding value-node in the name-tree. If it reaches a leaf of either the name-specifier or the name-tree, it intersects S with the name-records at the corresponding value-node. If not, it makes a recursive call to compute the relevant set from the subtree rooted at the corresponding value-node, and intersects that with S.

We model *Lookup-Name* by three mutually exclusive invariants (*Lookup1-3*), that are presented in Figure 4-10. *Lookup1-2* handle the case without wild-cards, while *Lookup3* adds the functionality for handling them. Despite their cryptic appearance, these invariants are fairly simple to understand.

```
cond indexedSubset(r:Record, v:Value)
     {all a1:v.attNS,a2:v.attNT,v1p,v2p |
          (a1 = a2 && v1p  in a1.valNS && v2p  in a2.valNT && v1p = v2p)
          -> r  in v1p.lookup + v.recNT}
cond indexedSuperset (r : Record, v:Value)
     {all v1 : v.immFolNT | v1.recNT + v.recNT  in r}

inv Lookup1
    {all v:Value - WildCard |
         (v.~valNS = v.~valNT && (no v.attNS || no v.attNT))
         -> v.lookup = v.recNT}
inv Lookup2
    {all v:Value - WildCard |
         (v.~valNS = v.~valNT && some v.attNS && some v.attNT)
         -> (indexedSubset(v.lookup,v) &&
             no r : Record - v.lookup | indexedSubset(v.lookup+r,v))}
inv Lookup3
    {all v:WildCard |
         some v.~valNS.valNT
         -> (indexedSuperset(v.~immFolNS.lookup, v.~immFolNS) &&
             no r : v.~immFolNS.lookup |
                indexedSuperset(v.~immFolNS.lookup - r, v.~immFolNS))}
```

Figure 4-10: Constraints modelling *Lookup-Name*

Lookup1 says that if v has the same parent attribute in both the name-specifier and the name-tree and it corresponds to a leaf value-node in the name-tree or to a leaf av-pair in the name-specifier, then *v.lookup* is just the name-records contained in that value-node.

Lookup2 is more subtle and uses the auxillary condition *indexedSubset*, which provides the functionality of taking the intersection over a finite collection of sets. It uses the simple mathematical equivalences

$$\begin{aligned}&S = \cap_{i \in I} S_i \ \equiv \\ &\forall i \in I \cdot S \subseteq S_i \ \bigwedge \ \forall s \notin S \ \exists i \in I \cdot S \cup \{s\} \not\subseteq S_i\end{aligned}$$

and

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

Lookup3 is similar to *Lookup2* and adds the functionality of handling wild-cards. It makes use of *indexedSuperset*, which behaves as a dual to *indexedSubset*.

To see the correspondence between invariants *Lookup1-3* and the pseudo-code in Figure 4-9, notice first that after its initialization, S is modified at 3 points in the pseudo-code. All these points are inside the *for* loop. Simply stated, each of our invariants corresponds to one of these points (*Lookup1* to the second, *Lookup2* to the third, and *Lookup3* to the first). The only difference is that we have incorporated the effects of repeated intersections and the union operations (while computing S' and on the last line) of the pseudo-code within our auxillary conditions *indexedSubset* and *indexedSuperset*. This was made possible by the two equivalences mentioned above.

Figures 4-6, 4-7, 4-8, and 4-10 form the core of our Alloy model of INS. We are now in a position to make assertions about this model and use Alcoa to test whether or not our beliefs about INS are sound.

Chapter 5

Analysis

We use Alcoa to perform four kinds of analyses on our Alloy model of INS. First, we test some basic properties that should hold in a valid naming scheme. Second,we analyze the published claims concerning the use of wild-cards in INS. Third, we observe the effect that addition of a new advertisement to the name-tree can have. Finally, we evaluate two fixes that were proposed by inventors of INS, intended to patch up the bugs revealed by analysis of Section 5.1.2 (that follows).

5.1 Testing fundamental properties

It is crucial for a naming scheme to have a name resolution algorithm that provides valid results only. Moreover, existence of valid services should result in clients seeking for them being served. Below we test some properties that we consider are fundamental to any naming scheme.

5.1.1 When INS works

We begin our analysis by testing the behavior of INS in the simple case when the name-specifier is added to the name-tree using *Add-Name* and exists there at the time of query resolution. Having made this assumption, we check the validity of the following three theorems.

```
cond NSExistsNT
     {all a,v | a.valNS= v
                -> (a.valNT = v && a.~attNS = a.~attNT)}
cond AlreadyAdded
     {NSExistsNT && some r | all v | some v.~valNS && no v.attNS
                                       -> r in v.recNT}
cond IsLeafAVPair(a:Attribute, v:Value){a.valNS=v && no v.attNS}
cond IsValidRecord(r:Record)
     {all a,v | IsLeafAVPair(a,v)
                -> r in v.recNT + v.+immPreNT.recNT}
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 5-1: Three basic theorems about *Lookup-Name*

```
Analyzing LookupOK2 ...
Scopes: Record(4), Value(4), Attribute(4)
Conversion time: 0 seconds
Solver time: 2 seconds
Total conversion time: 0 seconds
Total solver time: 2 seconds
No counterexamples exist in this scope
```

Figure 5-2: Alcoa output for analysis of *LookupOK2*

- *Lookup-Name* returns at least some name-record
- All name-records returned by *Lookup-Name* are valid
- All valid name-records are returned by *Lookup-Name*

For now, we consider a name-record to be valid if and only if it is included in the set of name-records of all leaf value-nodes, that match those of the name-specifier, or their parents. A more general treatment is presented in Section 7.1, where we argue that the validity semantics of INS need a reexamination to make the naming more versatile.

Figure 5-1 shows the Alloy code used to make the assertions above. Condition *NSExistsNT* ensures existence of the name-specifier in the name-tree. *AlreadyAdded* enforces this existence to be a result of addition. *IsLeafAVPair* determines whether the input parameters form a leaf av-pair in the name-specifier, and *IsValidRecord* formalises the definition of a valid record. *SomeRecordReturned* just asserts that the result of *Lookup-Name* is non-empty. *AllRecordsReturnedAreValid* and *AllValidRecordsAreReturned* have been named to describe their behavior. Finally the three assertions *LookupOK1*, *LookupOK2*, and *LookupOK3* test the properties listed above.

In all three tests (*LookupOK1*, *LookupOK2*, and *LookupOK3*), Alcoa completes its search without finding a counterexample. This gives us confidence that INS's resolution mechanism is sound when the name-specifier appears exactly in the name-tree as a result of an advertisement. Figure 5-2 shows sample Alcoa output resulting from the analysis of *LookupOK2*.

This concludes our analysis under the simplifying assumption that the name-specifier was added to the name-tree before lookup.

5.1.2 Problems with INS

Here, we consider relaxing the requirement that the name-specifier exists in the name-tree. This leads to discovery of several flaws in the *Lookup-Name* algorithm appearing in [2]. We conduct a series of tests that are presented in Figure 5-3 and described below.

```
assert LookupOK4 {no Root.attNT & Root.attNS -> no Root.lookup}

cond NoRecordOnAVMismatch
     {all a,r | (a in Root.attNS & Root.attNT && no a.valNS & a.valNT
                 && a.valNS != WildCard && r in a.valNT.recNT)
                -> not r in Root.lookup}
assert LookupOK5 {NoRecordOnAVMismatch}

cond SomeCommonAV {some a | a in Root.attNS & Root.attNT &&
                            some a.valNS & a.valNT}
assert LookupOK6{SomeCommonAV -> NoRecordOnAVMismatch}
```

Figure 5-3: More basic theorems

Our first test in the general case (*LookupOK4*) checks the claim that if the name-tree has no attributes in common with the name-specifier at the top most level, then *Lookup-Name* should return the empty set. Alcoa quickly generates a counterexample. Figure 5-4 shows the original Alcoa counterexample and Figure 5-5 presents a graphical illustration.

It so happens that the INS algorithm returns all name-records in the name-tree 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*, Figure 5-3) tests the case in which there exists some common top level attribute. It is natural to believe that if a value-node in the name-tree had no matching av-pair in the name-specifier, while its parent attribute-node had one, then the name-records of this value-node would not appear in the result of *Lookup-Name*. Once again, Alcoa produces a counterexample that is illustrated in Figure 5-6.

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 (Section 4.4).

We next pose the same question under the assumption that the name-tree and

```
Analyzing LookupOK4 ...
Scopes: Record(1), Value(2), Attribute(2)
Conversion time: 0 seconds
Solver time: 0 seconds
Counterexample found:
Domains:
  Attribute = {A0,A1}
  Record = {R0}
  Value = {Root,V0}
Relations:
  attNS = {Root -> {A1}}
  attNT = {Root -> {A0}}
  immFolNS = {Root -> {V0}}
  immFolNT = {Root -> {V0}}
  immPreNT = {V0 -> {Root}}
  lookup = {Root -> {R0}}
  recNT = {V0 -> {R0}}
  valNS = {A1 -> V0}
  valNT = {A0 -> {V0}}
Skolem constants:
  $1 = Root
  $50 = A1
  $51 = A0
  $81 = R0
```

Figure 5-4: Alcoa output for analysis of *LookupOK4*

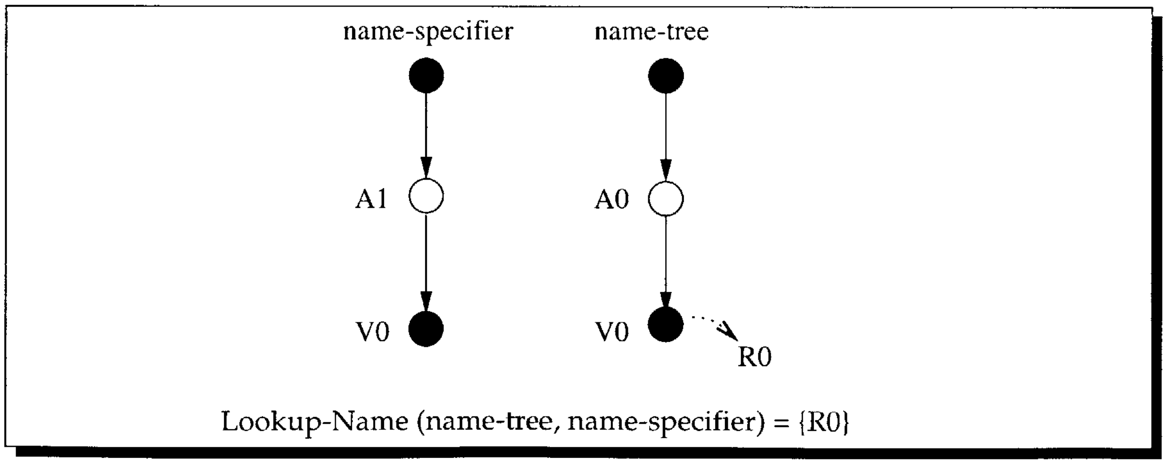


Figure 5-5: Graphical illustration of Alcoa counterexample to *LookupOK4*

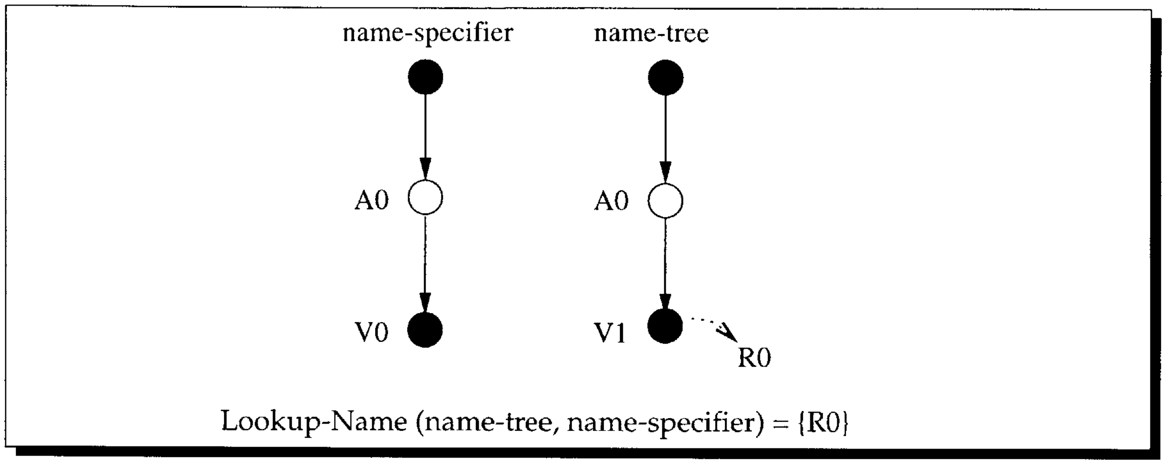


Figure 5-6: Counterexample to *LookupOK5*

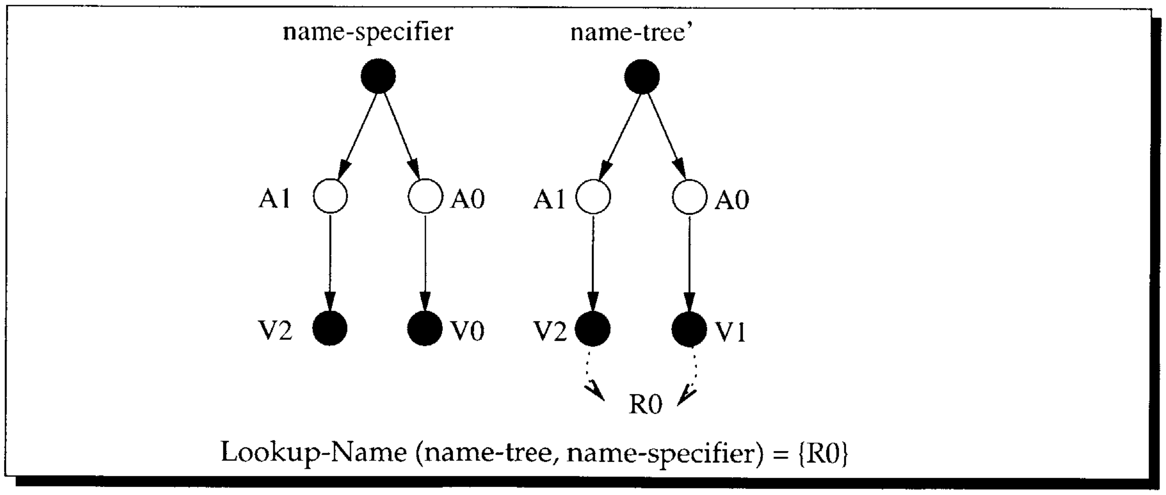


Figure 5-7: Counterexample to *LookupOK6*

the name-specifier have a common attribute and moreover one of the corresponding values also match (*LookupOK6*, Figure 5-3).

This time, not surprisingly though, Alcoa disproves the assertion with the counterexample displayed in Figure 5-7. The root of this bug is the same as that illustrated by *LookupOK5*.

5.2 Analyzing published claims

The published description of *Lookup-Name* [2] says:

> The algorithm uses the assumption that omitted attributes correspond to wild-cards; this is true for both queries and advertisements.

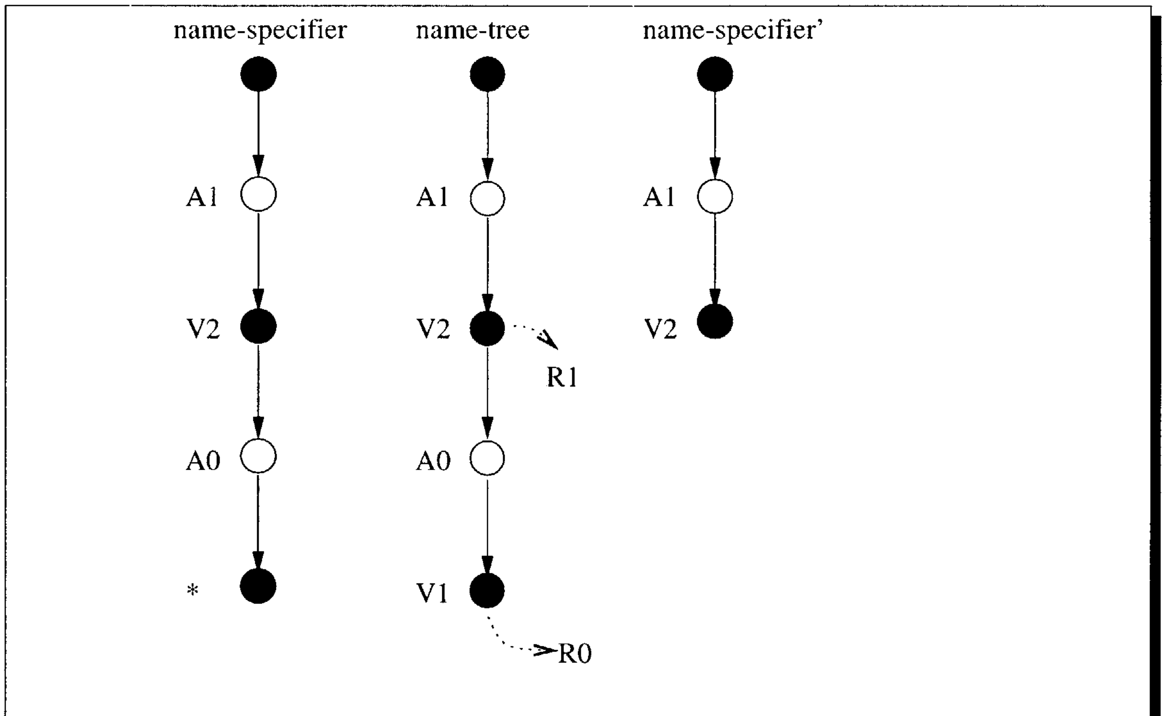
We put this claim to test (Figure 5-8) in the case of queries as follows. An Alloy operation *RemoveWildCard* is defined that removes wild-cards from a name-specifier. The operation mutates the name-specifier by removing the av-pair(s) containing wild-card(s), while maintaining the state of other av-pairs and the original name-tree. Our assertion, *LookupOK11*, (Figure 5-8) says that the result of *Lookup-Name* should be the same before and after this mutation.

This assertion is not valid; a counterexample is shown in Figure 5-9. Before mutation (i.e. with wild-cards) the name records *R0* and *R1* are returned; after mutation (i.e. with omission in place of wildcards), only *R1* is returned.

For the case of advertisements, one of our analyses (*LookupOK10*, Figure 5-10) already disproves the claim. Figure 5-11 illustrates the Alcoa counterexample. It also points out the difference between the name-specifier simply being 'present' by virtue of a correspondence in the data structures and it having been 'added' to the name-tree by an advertisement. If the contested equivalence were to hold, *Root.lookup* should be $\{R0, R1\}$, but it is empty.

```
op RemoveWildCard {
  all a | a.valNS != WildCard -> a.valNS' = a.valNS
  all a | a.valNS = WildCard -> no a.valNS'
  all v | all a:v.attNS | a.valNS = WildCard
                          -> v.attNS' = v.attNS - a
  all v | all a:v.attNS | a.valNS != WildCard
                          -> v.attNS' = v.attNS
  all v | v.attNT' = v.attNT && v.recNT' = v.recNT &&
          v.immFolNT' = v.immFolNT && v.immPreNT' = v.immPreNT
  all a | a.valNT' = a.valNT
}
assert LookupOK11 {RemoveWildCard  -> Root.lookup = Root.lookup'}
```

Figure 5-8: Alloy code for *LookupOK11*



Lookup-Name (name-tree, name-specifier) = {R0,R1}

Lookup-Name (name-tree, name-specifier') = {R1}

Figure 5-9: Counterexample to *LookupOK11*

```
cond NSMatchesNT {NSExistsNT &&
                    all a | some a.~attNS & a.~attNT}
assert LookupOK10 {NSMatchesNT -> some Root.lookup}
```

Figure 5-10: Alloy code for *LookupOK10*

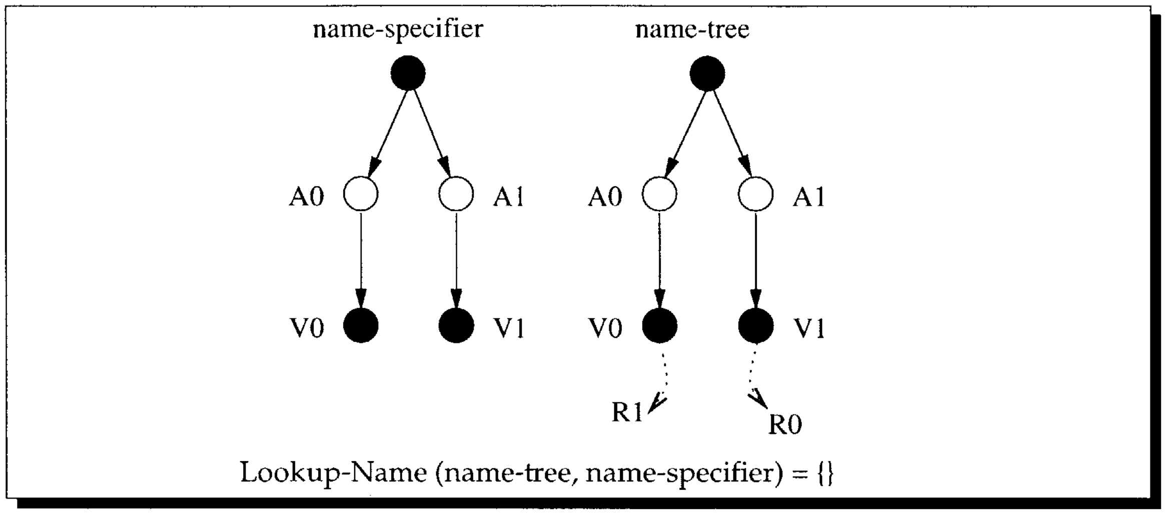


Figure 5-11: Counterexample to *LookupOK10*

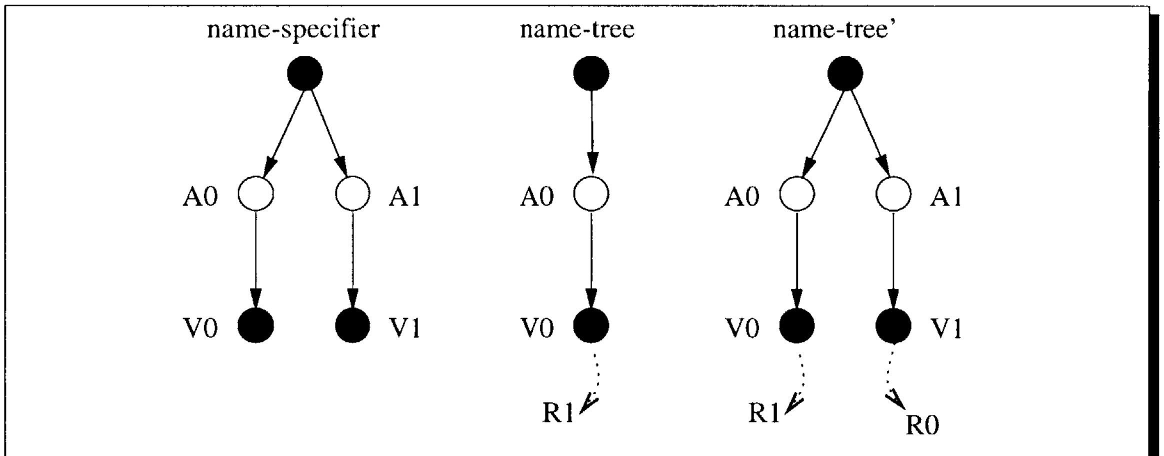
5.3 Behavior of new advertisements

In this section we analyze the effects that new advertisements can have on existing ones. Notice that we have not modeled the *Add-Name* algorithm in its full glory. Despite that, Alloy lets us ask questions about addition of new name-records in the name-tree! This is achieved by defining an Alloy operation *AddRecordTest* which mutates the state, the effect of which is that a new name-record, and some new attribute-nodes and value-nodes are added to the existing name-tree. Moreover, the original name-specifier is preserved, and the result of *Lookup-Name* before and after the mutation are compared.

Observe that *AddRecordTest* differs from *Add-Name* in INS, since *AddRecordTest* does not take a name-specifier and the corresponding name-record as its parameters and just mutates the state to represent a valid execution of *Add-Name* on some (essentially arbitrary) name-specifier and name-record.

```
op AddRecordTest{
  all v | some v.attNS && some v.attNT -> some v.attNS & v.attNT
  all a | (some a.valNT' && some a.valNS') -> a.valNT' in a.valNS'
  Attribute' = Attribute
  Value' = Value
  all a | a.valNS' = a.valNS
  all v | v.attNS' = v.attNS
  all a | a.valNT in a.valNT'
  all v | v.attNT in v.attNT'
  all v | some v.recNT -> v.recNT = v.recNT'
  all v | no v.recNT -> sole v.recNT' && v.recNT' in Record' - Record
 }
 assert LookupOK7 {AddRecordTest -> Root.lookup in Root.lookup'}
```

Figure 5-12: Theorem to analyze the effect of *Add-Name*



Lookup-Name (name-tree, name-specifier) = {R1}

Lookup-Name (name-tree′, name-specifier) = {}

Figure 5-13: Counterexample to *LookupOK7*

Figure 5-12 presents the Alloy operation and assertion that is tested. Note that in the operation the first two lines include code to cover up the bugs already discovered by *LookupOK4* and *LookupOK5*, since we would like to become aware of new flaws, if any. The assertion *LookupOK7* checks whether the name-records that result from the execution of *Lookup-Name* before the addition to the name-tree are included in the result after the addition.

Contrary to our expectations, Alcoa disproves this assertion with a counterexample that is illustrated in Figure 5-13. This is a very serious defect and implies that *Lookup-Name* is non-monotonic. The counterexample also points out that addition of new services do not just invalidate existing ones, but may also result in *Lookup-Name* returning no name-record at all, when in fact there are several services available, that were considered valid before the addition was performed!

5.4 Testing proposed fixes

Private communication with the inventors of INS revealed that they believed their implementation [20] successfully patches the bugs discovered by Alloy assertions *LookupOK4* and *LookupOK5*. We take these fixes and put them to test to see if they are valid, i.e. they fix the problems without introducing new errors.

We first evaluate the proposed fix for flaw revealed by *LookupOK4*. The implementation uses a flag to represent that the set S (Section 4.4) on initialization contains all name-records instead of actually inserting all name records into it. On returning from *Lookup-Name* a test is made to see if the flag is set and in that case the string $\{*\}$ is output to indicate no records. However, if the flag is set, then the union of S with any set results in S with the flag set.

We model this behavior in Alloy by defining a condition *ReturnStar* (Figure 5-14), which just checks if *Lookup-Name* returns all name-records[1]. Then we define an operation $AddRecordTestBugFix1$ which is similar in functionality to *AddRecordTest*

[1]Notice that this is insufficient, but our assertion that follows takes it into account and generates a counterexample that invalidates the actual proposed fix

```
cond ReturnStar {Root.lookup=Record}
op AddRecordTestBugFix1{
 all v | sole v.attNS
 all a | (some a.valNT' && some a.valNS') -> a.valNT' in a.valNS'
 Attribute' = Attribute
 Value' = Value
 all a | a.valNS' = a.valNS
 all v | v.attNS' = v.attNS
 all a | a.valNT in a.valNT'
 all v | v.attNT in v.attNT'
 all v | some v.recNT -> v.recNT = v.recNT'
 all v | no v.recNT -> sole v.recNT' && v.recNT' in Record' - Record
 some a | no a.~attNS
}
assert LookupOK8
       {AddRecordTestBugFix1
        -> ((some Root.lookup && not ReturnStar) -> not ReturnStar')}
```

Figure 5-14: Theorem testing proposed fix 1

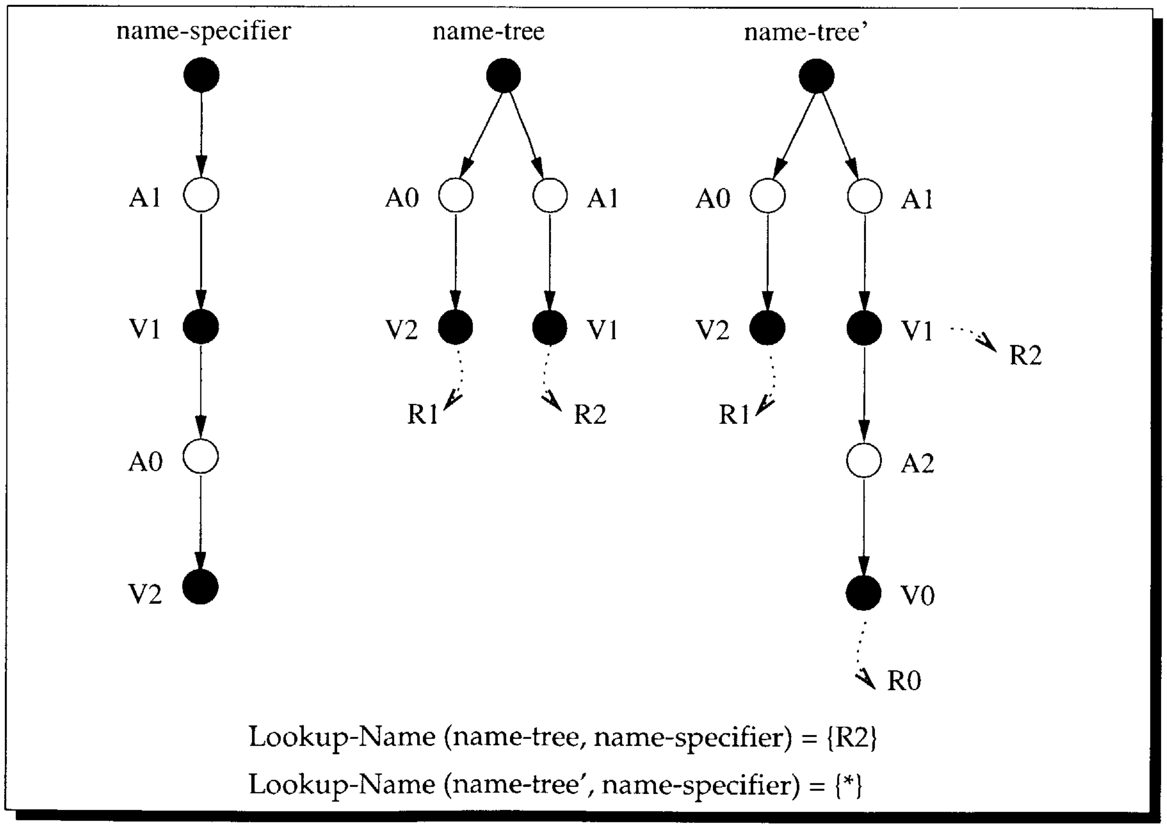

name-specifier
name-tree
name-tree'
A1
V1
A0
V2
A0
A1
V2
V1
R1
R2
A0
A1
V2
V1
R2
R1
A2
V0
R0
Lookup-Name (name-tree, name-specifier) = {R2}
Lookup-Name (name-tree', name-specifier) = {*}

Figure 5-15: Counterexample to *LookupOK8*

```
inv Lookup2p
    {all v | (v.~valNS = v.~valNT && some v.attNS &&
              some v.attNT && no a:v.attNS | a in v.attNT &&
              not a.valNS in a.valNT)
             -> (indexedSubset(v.lookup,v) &&
                 no r : Record - v.lookup | indexedSubset(v.lookup+r,v))}
inv Lookup2q
    {all v | (v.~valNS = v.~valNT && some v.attNS &&
              some v.attNT && some a:v.attNS | a in v.attNT &&
              not a.valNS in a.valNT)
             -> no v.lookup}

op AddRecordTestBugFix2{
 all v | sole v.attNS
 all v | some v.attNS && some v.attNT -> some v.attNS & v.attNT
 Attribute' = Attribute
 Value' = Value
 all a | a.valNS' = a.valNS
 all v | v.attNS' = v.attNS
 all a | a.valNT in a.valNT'
 all v | v.attNT in v.attNT'
 all v | some v.recNT -> v.recNT = v.recNT'
 all v | no v.recNT -> sole v.recNT' && v.recNT' in Record' - Record
 all a | sole a.valNT'
}
assert LookupOK9 {AddRecordTestBugFix2 -> Root.lookup in Root.lookup'}
```

Figure 5-16: Incorporating proposed fix 2 in our model

above, but removes the bug already discovered by *AddRecordTest* (line 1). Moreover, it also removes the bug indicated by *LookupOK5* (line 2).

To test the proposed fix we assert *LookupOK8*, which states that if *Lookup-Name* returned a name-record and the condition *ReturnStar* was false (i.e. a valid service was found) before the operation, then after the addition of a new service, *ReturnStar* should still be false (as otherwise either *Lookup-Name* returns nothing despite the existence of valid services or it returns a service that was considered invalid a priori to this addition but valid after the addition.)

Testing *LookupOK8* in Alcoa produces a counterexample which is illustrated in Figure 5-15. Notice that *Lookup-Name* considers $R2$ to be a valid service before the addition of $R0$, which due to a mismatch in the second level attribute ($A0$ versus $A2$) and a flaw in the proposed fix, renders $R2$ invalid after this addition.

Clearly detecting this subtle a flaw is non-trivial, but the use of Alcoa makes it feasible.

We now move on to evaluate the proposed fix to the bug discovered by *LookupOK5*. In this case the proposed fix can be modeled directly in Alloy as it concerns a change in the *Lookup-Name* algorithm. Figure 5-16 shows the Alloy code that mimics this change. The invariants *Lookup2p* and *Lookup2q* replace old invariant *Lookup2* (Figure 4-10) and implement the proposed fix.

We define an Alloy operation *AddRecordTestBugFix2*, which is once again similar in spirit to *AddRecordTest*, but removes the problems already discovered (first two lines). Our assertion *LookupOK9* is identical to *LookupOK7*, except for the operation it tests.

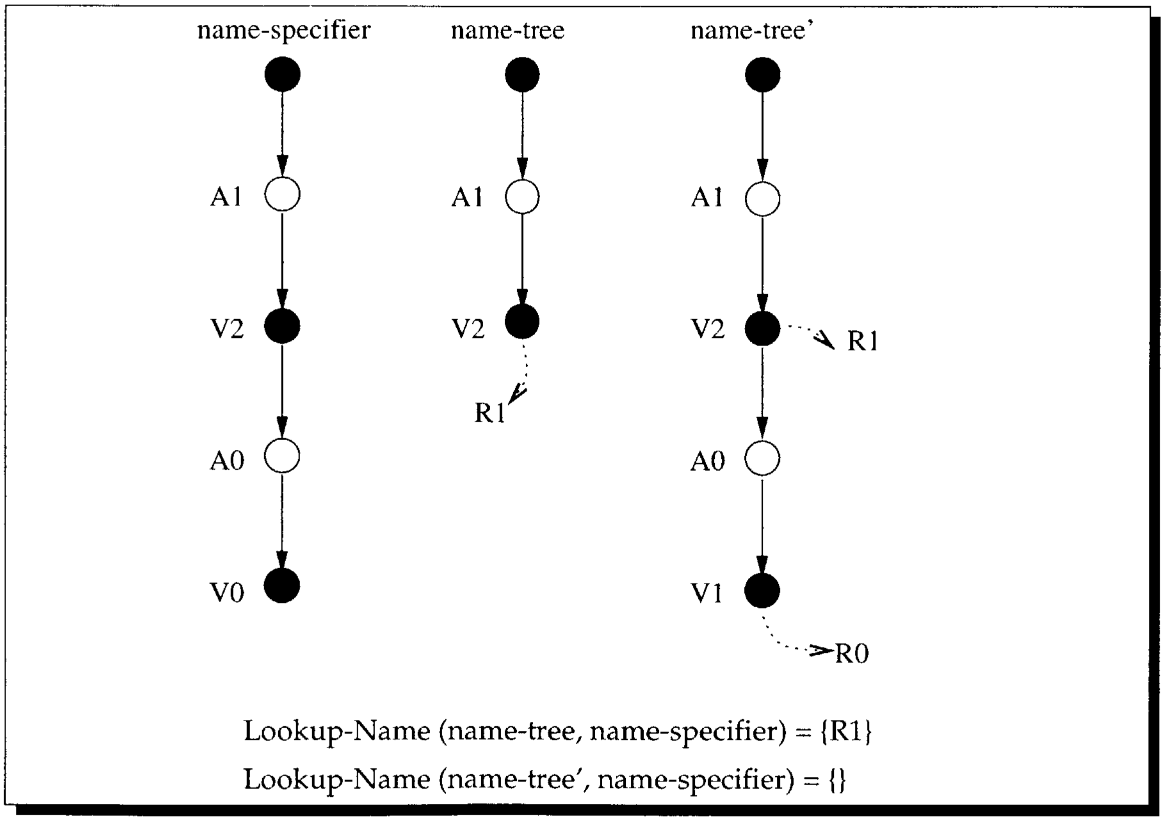

name-specifier
name-tree
name-tree'
A1
V2
A0
V0
A1
V2
R1
A1
V2
R1
A0
V1
R0
Lookup-Name (name-tree, name-specifier) = {R1}
Lookup-Name (name-tree', name-specifier) = {}

Figure 5-17: Counterexample to *LookupOK9*

As Figure 5-17 illustrates, Alcoa produces another counterexample, and invalidates this fix too! $R1$ that was considered to be a valid name-record by *Lookup-Name* is invalidated by the addition of $R0$. Notice, that all three counterexamples above resulting after addition are independent and point out distinct flaws!

Thus, both the proposed fixes are incorrect, and need to be reconsidered.

The problems exposed by *LookupOK4* and *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.

5.5 Summary of analyses

We did a comprehensive analysis of our model of INS in Alloy, thereby delineating the cases when the INS name resolution works fine, discovering several previously unknown flaws, and evaluating the proposed fixes. Table 5.1 summarizes the results of these analyses.

	Assertion checked in Alcoa	*Result*
Simple Case: name-specifier in name-tree	*Lookup-Name* returns something	Yes
	Every record returned is valid	Yes
	Every valid record is returned	Yes
Properties of any naming scheme	No record if no attributes match	No
	No record if no values match	No
	Record returned has matching value	No
	Addition does not invalidate existing records	No
Proposed fixes	Fix for bug found by LookupOK4 is sound	No
	Fix for bug found by LookupOK5 is sound	No
Particular claims about INS	Missing attribute $\sim$ wild-card (queries)	No
	Missing attribute $\sim$ wild-card (advertisements)	No

Table 5.1: Summary of analyses

Chapter 6

Evaluation

The Java implemetation of the naming scheme of INS appearing in [20] is 2300 lines long excluding *Add-Name* and *Get-Name*. About 900 of these lines constitute the testing code. Our model of the core functionality of INS (excluding *Get-Name*) consists of 55 lines of Alloy. All the analyses in Alcoa consist of another 85 lines of Alloy. Notice that these analyses include those performed to test the proposed fixes.

Scope / *Invariant*	3	4	5	6
LookupOK1	1 s	7 s	56 s	11 m 51 s
LookupOK2	1 s	4 s	41 s	6 m 26 s
LookupOK3	1 s	12 s	2 m 7 s	58 m 21 s

Table 6.1: Performance for exhaustive search

We evaluated the theorems presented in Section 5.1.1 using Alcoa with a scope between 3 and 6 (inclusive) in each domain. Alcoa reported back its analysis in a reasonable amount of time when the domains were restricted to be within 5. The times taken to perform these analyses are tabulated in Table 6.1. It is interesting to note that by performing the simplifications mentioned in Section 4.2, we were able to almost double the number of elements within the scope for each domain while

Invariant	*Time*
LookupOK4	2 s
LookupOK5	3 s
LookupOK6	3 s
LookupOK7	19 s
LookupOK8	24 s
LookupOK9	30 s
LookupOK10	3 s
LookupOK11	16 s

Table 6.2: Performance for counterexample detection using a scope of 4

performing an exhaustive search, and still got back the results in a reasonable amount of time.

Table 6.2 presents the performance analysis of counterexample generation by Alcoa. A scope of 4 elements in each domain is used to test assertions *LookupOK4-11*. The counterexamples to the analyses *LookupOK4*, *LookupOK5*, and *LookupOK6* (Section 5.1.2) took only a few seconds. Counterexamples to published claims (*LookupOK10*, and *LookupOK11*, Section 5.2) took 3 seconds and 16 seconds, respectively. The behaviour of new advertisements on existing ones (*LookupOK7*, Section 5.3) was analyzed in 19 seconds. Problems with proposed fixes (*LookupOK8* and *LookupOK9*, Section 5.4) were discovered in no more than 30 seconds. None of the counterexamples required more than 4 elements in any domain.

Observe that a uniform scope of 4 in each domain is not necessary to produce these counterexamples. We performed further tests by reducing the scope in each domain to get the minimum number of elements required to generate a counterexample. Table 6.3 shows the results produced. In only one case (i.e. *LookupOK8*) we require over 2 elements in more than 1 domain. Moreover, half of these counterexamples could have been produced by restricting the maximum scope to 3. This provides strong evidence in support of the "small scope hypothesis" – that most bugs can found in small scopes [13].

	Scope		
Invariant	*Att*	*Val*	*Rec*
LookupOK4	2	2	1
LookupOK5	1	3	1
LookupOK6	2	4	1
LookupOK7	2	3	2
LookupOK8	3	4	3
LookupOK9	2	4	2
LookupOK10	2	3	2
LookupOK11	2	4	2

Table 6.3: Minimum Scope Required

A 300 MHz Celeron processor running Windows NT with 128 MB of memory was used to perform all tests.

We were able to generate all counterexamples (except that for *LookupOK9*) without incorporating the functionality for wild-cards, which was added when *RemoveWildCard* was introduced[1]. This only emphasizes one of the various uses of incremental modeling.

[1]The performance analyses tabulated above reflect this

Chapter 7

Semantics issues

There are various issues regarding the development of a naming scheme based on intentional names that need clarification. In the following sections we discuss some of these issues and present our point of view on how to resolve them.

We begin with a discussion of what *conformance* should mean among name-records and name-specifiers in an intentional naming scenario. Then we discuss other issues that we consider to be of primary significance, and finally we point out the characteristics that in our opinion are vital for the validity of any naming scheme in general.

7.1 Conformance

The first step in the development of an intentional naming scheme is a formal definition of conformance. In the case of INS, the confusion about wild-cards is perhaps symptomatic of a lack of precise semantics.

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 conforming to 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 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

```
cond NRConformsNS (r:Record)
     {all a,v | IsLeafAVPair(a,v)
                -> some v1 | r in v1.recNT && v in v1 + v1.immPreNT}
assert LookupOK12
       {all r | NSExistsNT && NRConformsNS(r) -> r in Root.lookup}
```

Figure 7-1: Theorem testing conformance

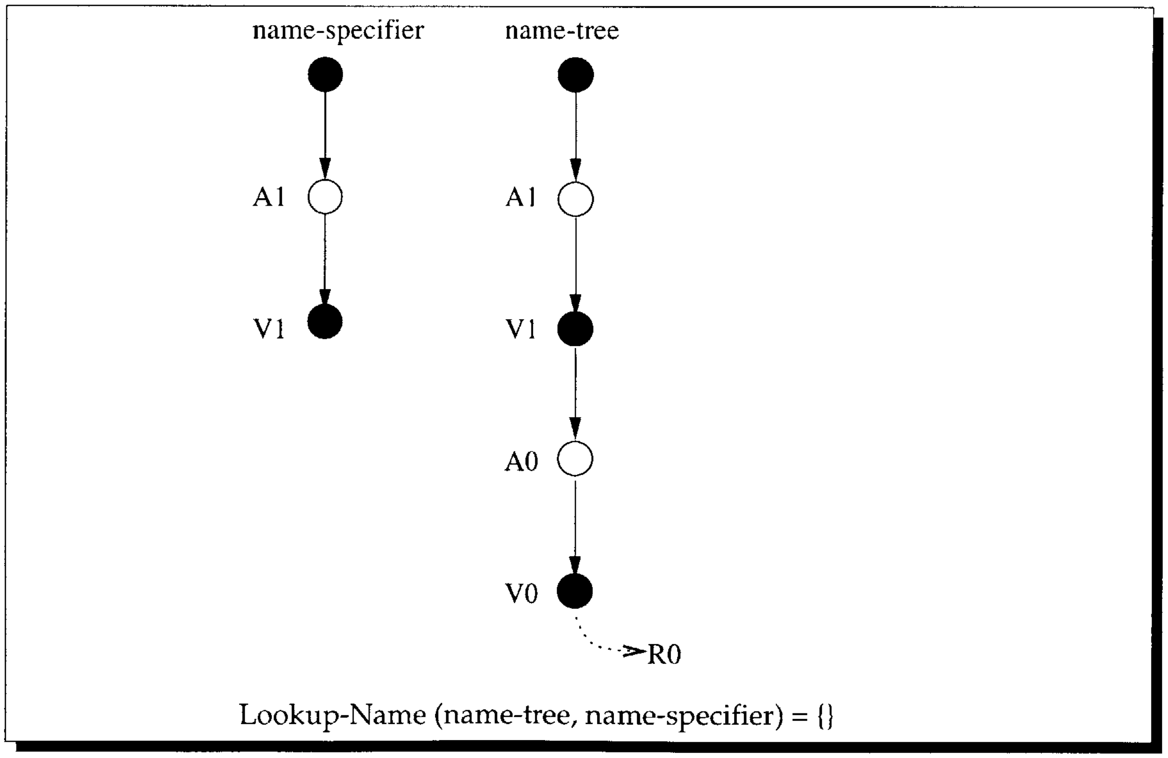


Figure 7-2: Counterexample to *LookupOK12*

know the exact details of the services 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 *LookupOK12* in Alloy (Figure 7-1). *NRConformsNS* defines a name-record to conform to the name-specifier if it appears in all the value-node(s) corresponding to each leaf av-pair or one of their descendant value-nodes. *LookupOK12* just tests if all the name-records that conform to the name-specifier according to this definition appear in the result of *Lookup-Name* .

Alcoa generates a counterexample in 5 sec to this assertion with a scope of 4 in each domain (Figure 7-2).[1]

Implementing the behavior of missing attributes as wild-cards is one way to provide this flexibility in INS, but that requires a significant alteration to *Lookup-Name* .

7.2 Other issues

Another issue that needs formalization is the meaning of missing attributes. It does not seem logical to treat any missing attribute as a wild-card. We suggest two orthogonal ways to classify its treatment. First missing attributes in queries should be dealt with separately from missing attributes in advertisements. Second the case in which all attributes are missing at a certain level should be separated from the case in which some attributes are missing but others are present.

Our analysis with Alcoa suggests that it would make sense to require that all top level attributes be specified by both the queries and advertisements. This could be achieved by defining a set of necessary attributes like *accessibility.* On the same lines, it sounds feasible to require that an attribute may only be ignored if all its orthogonal attributes are also ignored.

Another point that needs to be worked out is whether it is reasonable to assume that wild-cards only appear at the lowest level (an assumption made in [2]).

[1] A minimum scope of 2 *Attributes*, 3 *Values*, and 1 *Record* is required to produce a counterexample in this case. Again, wild-cards do not need to be implemented in order to observe this counterexample.

While specifying the semantics a naming scheme must also distinguish between the existence of the name-specifier due to a physical correspondence between data structures and its valid addition through an advertisement. We saw in Section 5.2 how the name-specifier 'existed' in the name-tree but the result of *Lookup-Name* was the empty-set.

A rather subtle issue of concern is how to handle applications that treat orthogonal attributes as specializing. Due to ambiguities in natural language and words having several (disparate) meanings, it is conceivable that the client considers an attribute to be orthogonal to another while the application in its advertisement treated them as one specializing the other.

Another issue that we believe needs attention is whether it is reasonable to require an attribute take one or all values. Notice that using the name-specifier there is no reasonable way to say, for instance, that a camera provides images in only 2 formats, namely JPEG and GIF. It is certainly desirable for a naming scheme to possess this flexibility. Thus it would make sense to have applications advertise their services using a name-tree instead of a name-specifier. A client could also use a similar data-structure to pose his query, however in his case when an attibute equals several values, it is more natural to treat them as a disjunction as opposed to the conjuction in the case of an advertisement. This, however, requires a major overhaul of the *Lookup-Name* algorithm.

Finally, it is our opinion that in case *Lookup-Name* is unable to return a service that advertised the name-specifier being resolved in its entirety, the client should be notified that the service(s) he is receiving are only approximate matches. This way he can take precautionary steps before sending his data to the service.

The semantics concerns expressed in this section are summarized in Figure 7-3.

7.3 Essential characteristics of a naming scheme

Our experience with analyzing INS in Alcoa has enabled us to capture certain features must be present in a naming scheme in general. These characteristics are summed up

S–I	Validity of name-records specializing a query.
S–II	Missing attribute $\sim$ wild-card?
S–III	Can wild-cards appear at any level?
S–IV	Existence versus addition (of a name-specifier in a name-tree).
S–V	Can a query specify specializing attributes as orthogonal?
S–VI	Must attributes be paired with either one or all values?
S–VII	Special treatment of partial matches?

Figure 7-3: Semantics issues

C–I	Addition monotonic
C–II	Addition commutative
C–III	Resolution idempotent
C–IV	Resolution only returns conforming objects
C–V	Resolution returns all conforming objects
C–VI	Resolution independent of *order* of prior operations
C–VII	All non-conforming objects have the same effect on resolution
C–VIII	For every non-empty catalog, some name resolves to an object

Figure 7-4: Essential characteristics

in Figure 7-4 and described below.

During the discussion that follows we make the following simplifying assumptions[2]. First of all, we assume a framework where a *catalog* stores the correspondence between *names* and *objects*. New objects are inserted into the catalog using an addition operation with parameters a name and the corresponding object to be inserted into the catalog, while existing objects are retrieved using a resolution operation that only has one parameter, namely the name that needs to be looked up. In INS, the name-tree acts as a catalog, while a name-specifier is a name and name-record an object; the addition operation being *Add-Name* and the resolution operation is *Lookup-Name* . Next, we assume that addition only involves insertions of new objects; names may, however, be re-used. Finally, we assume no temporal effects like time-stamping of object insertions and automatic deletion of non-renewed objects. Thus, insertions into the catalog exist indefinitely.

As displayed earlier, a foremost requirement is a formal definition of conformance of an object to a name. Once, this has been established, the naming scheme needs to possess the following properties.

C–I The effect of addition on name resolution should be monotonic. That is for an arbitrary name-specifier, the result of lookup before performing an addition operation should be a subset of that after performing this operation. As we saw in section 5.3, INS lacks this fundamental property.

C–II Addition should be commutative. This is to say that for an arbitrary name-specifier, the result of the resolution operation after any sequence of addition operations, should be the same for all permutations of that sequence.

C–III Resolution should be idempotent. In other words, the effect of executing this operation once should be the same as the effect of executing it multiple times, given an arbitrary name and no interleaving addition(s).

[2]These assumptions can be relaxed in a straightforward manner, however, in our current work we do not tackle the general case (with temporal effects) and it is left as future work.

C–IV The resolution operation must return objects that conform to the given name. This means that no object that fails to meet the comformance criteria is ever included in the result, and thus a camera never results when a printer is sought.

C–V The resolution operation must return all objects that conform to the given name. We believe, this condition can even be relaxed to speed up resolution. For example, it may be feasible to give priority to objects whose names match more closely to what is sought.

C–VI The result of the resolution operation given a name, must be independent of the order in which all previous operations are performed. In particular, this result should be the same for any possible interleaving or reordering of additions or resolutions that were carried out on the catalog prior to the execution of this resolution.

C–VII The behavior of addition of non-conforming objects should be identical. This is to say that for a given name, its resolution after the addition of a non-conforming object should result in the same objects, even if a different non-conforming object was inserted into the catalog (prior to this resolution).

C–VIII For any non-empty catalog there must be a name that results in a non-empty resolution. This condition is not guaranteed by $C - IV$ and $C - V$ and is necessary for liveness.

Chapter 8

Related work and Conclusions

A variety of models have been constructed in Alloy and analyzed in Alcoa, but Alcoa has not been used previously for the analysis of a recursive algorithm like *Lookup-Name* .

Jackson and Sullivan [15] recently recast a model of COM originally written in Z [24] into Alloy, and showed that its analysis can be automated. The resulting Alloy model is about 150 lines long, and has 8 relations, 1 indexed relation (i.e. function from a basic type to a relation), and 8 sets. Using Alcoa, they were able to generate automatically the counterexamples that Sullivan and his colleagues had found by hand analysis and extend the original analysis.

Vaziri and Jackson [25] translated the entire core metamodel of UML from OCL [27], into Alloy. The resulting model, which is about 400 lines long, is about half the size of the OCL version. It has 41 relations and 37 sets. They used Alcoa to show that the metamodel is consistent, by generating a sample UML model that satisfies all the constraints (with additional constraints that rule out the trivial empty model).

Formalizing software structures for purposes of precision and analysis is not a new idea. Abowd et. al [1] gave a formal semantics to informal architectural diagram using Z. Garlen et. al [6] displayed efficient development of tools for architecture design using such models. Various other work has demonstrated the use of a variety of mathematical frameworks to formalize and analyze models of software architecture. Inverardi and Wolf [10] used chemical abstract machines to formalize and analyze architectures.

Nor is fully automatic or interactive checking of formal specifications a new concept. Model checking applied to a finite state machine is well known. Alcoa is an analog, albeit incomplete, based on a systematic search of bindings for specifications in the setting of first-order logic and set theory. Such a framework is especially well-suited to modeling complex software structures (as opposed to protocols).

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

- Model checkers such as SPIN [9] provide similar exhaustive search to Alcoa. However, they are primarily designed for addressing the complexities that arise from concurrency, and their input languages therefore often parallel composition and communication mechanisms, and logics for describing event and state sequences. 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. In contrast, Alcoa addresses the complexity that arises from relational state structure. 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.

- Theorem provers such as PVS [18] can, unlike Alcoa, prove a theorem for all possible cases, thus offering greater confidence, but at greater expense. Modern theorem provers make extensive use of decision procedures and can thus automate several low level proof steps. Several theorem provers have been embedded in tools for specific languages (e.g. Z/EVES [5] for Z and LP [7] for Larch) to bridge the gap between proof obligations and assertions in the specification language. Despite these advances, they are still tools for experts only, since complex proofs often fail because of flaws in the proof strategy and not in the assertion being checked. Theorem provers tend not to fail gracefully, and do not generally provide counterexamples. Thus despite the greater confidence they can provide, theorem provers are often too demanding for everyday use.

- Specification animation tools, such as IFAD's VDM tool [3], 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. They typically execute operations from states constructed by the user, and can check, for example, that the states that follow satisfy invariants. Alcoa typically considers many more executions; while an animation tool is usually stepped through manually, Alcoa searches all possible executions - usually billions - within the given scope. Alcoa can even be induced to behave like an animation tool. For example, the user may first specify a condition and then request an execution of an operation that starts from a state satisfying the condition; the user may equally choose to constrain the post-state and 'execute' the operation backwards.

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

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 constructed automatically. We succeeded in trimming it down to three domains and less than half its original length. This final model was about one twentieth of the size of the implementation of the *Lookup-Name* operation and its test drivers. It remains a very interesting question if such remodelling can be automated, and we hope to explore this in future.

Moreover, we were able to formulate the *Lookup-Name* 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. Automation of this kind of overhauling seems to be very tricky since it requires advanced knowledge of the behavior of the operation involved.

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

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. Moreover, this use lies not only in finding flaws but also in simplifying existing designs. Incremental modeling together with interactive analysis are extremely useful in boiling down a model to encompass only the essential components, thus making it more amenable to both human understanding and automated analysis.

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.

Bibliography

[1] G. D. Abowd, R. Allen, and D. Garlan. Formalising style to understand descriptions of software architecture. In *ACM transactions on Software Engineering and Methodology,* Vol. 4, No. 4, October 1995.

[2] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *17th ACM Symposium on Operating Systems (SOSP 99)*, Kiawah Island, December 1999.

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

[4] R. J. Allan, D. Garlan, and J. Ivers. Formal modeling and analysis of the HLA component integration standard. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, November 1998.

[5] D. Craigen, I. Meisels, and M. Saaltink. Analysing Z specifications with Z/EVES. In *Industrial-Strength Formal Methods in Practice,* J.P. Bowen and M.G. Hinchey (Editors), September 1999.

[6] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.

[7] S. J. Garland and J. V. Guttag. A guide to LP: the Larch prover. Available as Research Report 82, Compaq Systems Research Center, Palo Alto, CA, December 1991.

[8] J. V. Guttag and J. J. Horning. Formal specification as a design tool. In *Conf. on Principles of Programming Languages (POPL 80)*, Las Vegas, Nevada, 1980.

[9] G. J. Holzmann. The model checker spin. In *IEEE Transactions on Software Engineering, Special issue on Formal Methods in Software Practice*, May 1997. Volume 23, Number 5.

[10] P. Inverardi and A. L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. In *IEEE Transactions on Software Engineering, SE-21,4*, April 1995.

[11] D. Jackson. Alloy: A lightweight object modelling notation. Technical Report 797, MIT Laboratory for Computer Science, Cambridge, MA, February 2000.

[12] D. Jackson. An automatic analysis for a first-order relational logic. Submitted for publication. Avaliable at: `http://sdg.lcs.mit.edu/ dnj/publications`, March 2000.

[13] D. Jackson and C. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. In *International Symposium on Software Testing and Analysis*, Jan 1996.

[14] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proceedings of the International Conference on Software Engineering*, Limerick, Ireland, June 2000.

[15] D. Jackson and K. Sullivan. Com revisited: Tool-assisted modeling and analysis of software structures. Submitted for publication, March 2000.

[16] D. Jackson and J. Wing. Lightweight formal methods. In *IEEE Computer*, April 1996.

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

[18] S. Owre, J. Rushby, N. Shankar, and F. Von Henke. Formal verification for fault-tolerant architectures: Prolegomena to design of PVS. In *IEEE Transactions on Software Engineering*, February 1995. 21(2).

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

[20] Elliot Schwartz. Design and implementation of intentional names. Masters Thesis, MIT Laboratory for Computer Science, Cambridge, MA, May 1999.

[21] K. Sklower. A tree-based packet routing table for Berkeley Unix. Technical Report, University of California, Berkeley, CA, 1993.

[22] J. M. Spivey. *The Z notation: A Reference Manual.* Prentice Hall, second edition, 1992.

[23] K. J. Sullivan, M. Marchukov, and D. Socha. Analysis of a conflict between interface negotiation and aggregation in Microsoft's component object model. In *IEEE transactions on Software Engineering*, Jul/Aug 1999.

[24] K. J. Sullivan, J. Socha, and M. Marchukov. Using formal methods to reason about architectural standards. In *Proceedings of the International Conference of Software Engineering (ICSE97)*, Boston, MA, May 1997.

[25] M. Vaziri and D. Jackson. Some shortcomings of OCL, the Object Constraint Language of UML. A response to Object Management Group RFI on UML, December 1999.

[26] P. Vixie. *Name Server Operations Guide for BIND - Release 4.9.5.* 1996. Internet Software Consortium.

[27] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1999.

[28] H. Zhang. SATO: An efficient propositional prover. In *Proc. of International Conference on Automated Deduction (CADE-97).*