This chapter describes the incremental construction and analysis of a small model. My intent is to explain just enough to impart the flavor of the approach, so don’t expect to follow all the details.

I’ve chosen an example that should be familiar to most readers: the design of an address book for an email client. Although I’ve kept the model small to simplify the presentation, this example isn’t atypical in the amount of effort involved. A ten-line program can’t do very much, and has almost nothing in common with a thousand-line program. But a ten-line model can be very useful, and doesn’t differ that much from a hundred-line model, which is often all that’s needed to explore a difficult design issue.

By developing the example in a series of small additions and modifications, I’ve attempted to convey the lightweight and incremental spirit of the approach. The immediacy of the feedback that the tool provides is much harder to get across; to experience this, you’ll need to try the example yourself, running analyses and seeing how they react to your own modifications.

An email client’s address book is a little database that associates email addresses with shorter names that are more convenient to use. The user can create an alias for a correspondent—a nickname that can be used in place of that person’s address, and which need not change when the address itself changes. A group is like an alias but is associated with an entire set of correspondents—the members of a family, for instance. When defining a group, a user will often insert aliases rather than actual email addresses, so that a change in a person’s email address can be corrected in just one place, even if it appears implicitly in many groups.

The tour starts with a simple address book with aliases and no groups. It shows how to declare the structure of the state of a system, and how to generate sample instances of the state (section 2.1). Then it adds dynamic behavior, and shows how to model an operation with constraints, how to simulate it, and how to check properties of operations (section 2.2).

The tour then takes a turn into more sophisticated territory. The state of the address book is elaborated to allow names (that is, groups and
aliases) to refer to other names, forming naming chains of any length (section 2.3). The model uses an idiom that design pattern afficionados call Composite. The analyses of the simple address book are reapplied, and now turn up some potential problems.

Finally, the model is extended with traces, so that now analyses and simulations show entire executions involving a series of operations, rather than single operation steps (section 2.4). I included this section to show the flexibility of the approach, especially for readers familiar with model checking, although in practice it’s often fine just to analyze operations one at a time.

2.1 Statics: Exploring States

We’re going to explore a simple address book for an email client that maintains a mapping from names to addresses. Here’s our first model:

```alloy
tour/addressBook1

module tour/addressBook1

sig Name, Addr {}

sig Book {
    addr: Name -> lone Addr
}
```

That’s a complete Alloy model. It introduces three signatures—Name, Addr, and Book—each representing a set of objects. The Book signature has a field addr that maps names to addresses. In fact, addr is a three-way mapping associating books, names, and addresses, containing the tuple \( b \rightarrow n \rightarrow a \) when, in book \( b \), name \( n \) is mapped to address \( a \). The expression \( b.addr \) denotes the mapping from names to addresses for book \( b \).

The keyword lone in the declaration indicates multiplicity—in this case that each name is mapped to at most one address. For now, we’re just modeling simple aliases; later we’ll consider groups.

This model contains no commands, so there’s no analysis that can be done (beyond simple static semantic and type checks). Our first analysis will be to get some samples of the possible states. To do this, we add a predicate, and a command to find an instance of the predicate:

```alloy
pred show {}
run show for 3 but 1 Book
```

The predicate has an empty body; later we’ll add some constraints. The command specifies a scope that bounds the search for instances: in this case, to at most three objects in each signature, except for Book, which
is limited to one object, since, for now, we’re only interested in seeing a single address book. The scope is for the purpose of analysis alone; the model doesn’t limit the size or number of address books.

Running the command produces the instance of fig. 2.1. Outputs can be shown in a variety of forms, textual and graphical. Here, I’ve chosen to have the output displayed as a graph, and I’ve instructed the analyzer to “project” the instance on Book, which means that it shows a separate graph for each book object.

You may wonder why this particular instance was chosen. In fact, the tool’s selection of instances is arbitrary, and depending on the preferences you’ve set, may even change from run to run. In practice, though, the first instance generated does tend to be a small one. This is useful, because the small instances are often pathological, and thus more likely to expose subtle problems. You can ask the tool to produce a series of instances without repeats, but in our tour, we’ll always make do with the first one.

This instance shows a single link from a name to an address. To see an instance with more than one link, we can add a constraint to the predicate:

```prolog
pred show (b: Book) {
    #b.addr > 1
}
```
So that we can talk about a particular book, I’ve added an argument \( b \) of type \( \text{Book} \) to the predicate. The expression \( b.addr \) is the mapping from names to addresses for this book, and \( \#b.addr \) is the number of associations in this mapping. So the constraint asks for an instance in which the book \( b \) has more than one name/address association.

Running the command again now gives the instance of fig. 2.2. We see that our model allows two names (three in this case!) to map to one address. How about the converse? Does our model allow one name to map to two addresses? If we add a constraint asking for such a name

\[
\text{pred} \text{ show} (b: \text{Book}) \{
\quad \#b.addr > 1
\quad \text{some} \ n: \text{Name} \ | \ \#n.(b.addr) > 1
\}\]

the analyzer tells us that the predicate \textit{show} is now inconsistent—at least in this scope—and has no instances. This is not surprising, since the constraint we added contradicts the multiplicity in the declaration of \( addr \).

Even if we can’t have one name map to two addresses, we would like to make sure that it’s possible to have more than one address in the address book. So we replace the inconsistent constraint by a weaker one:

\[
\text{pred} \text{ show} (b: \text{Book}) \{
\quad \#b.addr > 1
\quad \#\text{Name}.(b.addr) > 1
\}\]
Whereas the bad constraint used the expression $n.(b.addr)$ for looking up a single name $n$ in address book $b$, this constraint uses $\text{Name.}(b.addr)$ for looking up the entire set of names. This expression therefore denotes the set of all addresses that may result from lookups. One of the nice features of Alloy is that the operators are defined very generally, and any operator that can be applied to a scalar can also be applied to a set.

Running the command gives the instance of fig. 2.3. These little simulations are useful because, with minimal effort on the user’s part, they confirm that the model doesn’t inadvertently rule out obvious cases, and they present other cases that might not have been considered at all.

So far, we’ve defined a state space and generated some sample states. It’s time to look at some behaviors.

### 2.2 Dynamics: Adding Operations

Let’s add to the model a description of what happens when an entry is added to an address book:

```alloy
pred add (b, b': Book, n: Name, a: Addr) {
  b'.addr = b.addr + n -> a
}
```
The predicate \textit{add}, like the predicate \textit{show}, is just a constraint. In this case, though, it represents an \textit{operation}, and describes dynamic behavior. Its arguments are an address book before the addition ($b$), an address book after ($b'$), a name ($n$), and an address ($a$) the name is to be mapped to. The constraint says that the address mapping in the new book is equal to the address mapping in the old book, with the addition of a link from the name to the address.

The way this operation is described will probably strike you as odd if you’re used to imperative programming languages and haven’t seen modeling languages before. There’s no explicit mutation here; instead, the before and after states of the book are given different names ($b$ and $b'$), and the effect of the operation is captured by a property relating them. Whereas a procedure in a program is \textit{operational}, and describes how to \textit{produce} a change of state by modifying state components, Alloy is \textit{declarative}, and describes how to \textit{check} whether a change of state is valid, by comparing the before and after values.

Even though Alloy is declarative, it can still be executed much like an operational language. To execute the operation, we run a command such as

\begin{verbatim}
run add for 3 but 2 Book
\end{verbatim}

This time we’ve limited the scope to just 2 books (for the before and after values). The result, in fig. 2.4, shows the prestate (the state of the book before the operation) above, and the poststate (the state after) below. In the prestate, the book is empty; in the poststate, there is a new link from Name0 to Addr0.

Note how the name node is marked with the label \textit{add\_n} and the address node with \textit{add\_a} to show which objects are bound to the arguments $n$ and $a$ of the \textit{add} operation. These labels will become more important later when they show witnesses to the violation of an assertion.

Following the same strategy we used for states, we can explore more interesting transitions by adding constraints. We could elaborate the predicate \textit{add} itself, but it’s better to create a new predicate, making a clear distinction between the operation itself and constraints written for the purpose of exploration:

\begin{verbatim}
pred showAdd (b, b': Book, n: Name, a: Addr) {
  add [b, b', n, a]
  #Name.(b'.addr) > 1
}
\end{verbatim}
The new predicate `showAdd` “invokes” the existing predicate `add`. The effect is no different from including the constraints of `add` directly (but it’s more modular to do it this way). We’ve added a constraint that asks for a transition in which the address book after has more than one address mapped to (using the same constraint we used when simulating states). The result is shown in fig. 2.5. Note that it’s just as easy to constrain the state after as constraining the state before: the analyzer is “executing” this operation backward.
Let's move on, and write some more operations, for deleting entries, and for lookup:

```plaintext
pred del (b, b': Book, n: Name) {
    b'.addr = b.addr - n -> Addr
}

fun lookup (b: Book, n: Name): set Addr {
    n.(b.addr)
}
```

FIG. 2.5 A generated transition for `showAdd`. 
The deletion operation says that the after-book is the before-book with all links from the name $n$ to any address removed. The lookup operation is written as a function rather than a predicate: its body is an expression rather than a constraint, and says that the result of a lookup is whatever set of addresses the name $n$ maps to under the $addr$ mapping of $b$.

We could simulate these operations too, but let’s do something different, and write some assertions about how combinations of operations in sequence behave. Our first assertion says that deletion is an undo operation for addition:

```
assert delUndoesAdd {
    all b,b',b": Book, n: Name, a: Addr |
    add [b,b',n,a] and del [b',b",n] implies b.addr = b".addr
}
```

An assertion is a constraint that is intended to be valid—that is, true for all possible cases. This one says that an addition from book $b$ resulting in book $b'$, followed by a deletion using the same name $n$, results in a book $b"$ whose address mapping is the same as that of the original book $b$.

To check the assertion, we issue the following command to the analyzer:

```
check delUndoesAdd for 3
```

This instructs the analyzer to search not for an example, but for a counterexample—a scenario in which the assertion is violated. And indeed, it finds one, as shown in fig. 2.6. Strangely, there are only two distinct states in this scenario. As the diagram at the bottom shows (produced by the visualizer with different settings), $b$ and $b'$, the values of the book in the first and second states, are both $Book0$, shown above on the left. The reason is that the name/address link to be added is already present, so the execution of $add$ has no effect. The execution of $del$, on the other hand, removes the link, resulting in the empty book, shown on the right.

Sometimes the failure of an assertion will point to a flaw in the model proper. In this case, however, the model seems reasonable, and given our decision to allow additions for existing entries, it’s not surprising that deletion doesn’t act as an undo. (At least, it’s not surprising in retrospect. Many of the issues raised by analysis are like bugs in code—perfectly obvious once you’ve already seen them.) To check that our hypothesis is right, we can modify the assertion, restricting the claim to cases in which no entry already exists for the name $n$:
assert delUndoesAdd {
    all b,b',b"": Book, n: Name, a: Addr |
    no n.(b.addr) and add [b,b',n,a] and del [b',b"",n]
    implies b.addr = b"".addr
}

Executing the check now finds no counterexample. The assertion may still be invalid, though. Since the analyzer only considered cases involving three books, three names, and three addresses, it’s possible that there is a counterexample involving more.

So we crank up the scope. There’s no point considering more than three books, but we allow 10 names and 10 addresses:

    check delUndoesAdd for 10 but 3 Book

Executing this takes a bit longer than the previous analyses (but still only a fraction of a second on a standard desktop machine). As you increase the scope, the space of cases to consider grows dramatically. With 10 names and addresses, there are 11 possibilities for each name, so the starting state alone has $11^{10}$ possible values. And because the op-
erations don’t have to be written in an executable style, the tool has to search over the possible values of all three books, so there are over $10^{30}$ cases to consider.

Now you can see why this kind of analysis is more effective than testing. Of course, the analyzer doesn’t construct and check each case individually; even if it used only one processor cycle per case, $10^{30}$ cases would still take longer than the age of the universe. By pruning the tree of possibilities, it can rule out large subspaces without examining them fully.

We still haven’t proved the assertion is valid. But, intuitively, it seems very unlikely that, if there is a problem, it can’t be shown in a counterexample with 10 names and addresses. How far to go is a pragmatic judgment you have to make as a modeler. Eventually, as you increase the scope, the analysis becomes intractable.

The tradeoff is no different in principle from the one you face when deciding whether you’ve tested a program enough. In practice, though, exhausting a scope of 10 gives more coverage of a model than hand-written test cases ever could. Most flaws in models can be illustrated by small instances, since they arise from some shape being handled incorrectly, and whether the shape belongs to a large or small instance makes no difference. So if the analysis considers all small instances, most flaws will be revealed. This observation, which I call the small scope hypothesis, is the fundamental premise that underlies Alloy’s analysis.

There are many other examples of assertions in this “algebraic” style. Here are two. The first checks that \texttt{add} is idempotent—that repeating an addition has no effect:

```plaintext
assert addIdempotent {
    all b,b',b": Book, n: Name, a: Addr |
    add [b,b',n,a] and add [b',b",n,a] implies b'.addr = b".addr
}
```

The second checks that \texttt{add} is local; that adding an entry for a name \texttt{n} doesn’t affect the result of a lookup for a different name \texttt{n'}:

```plaintext
assert addLocal {
    all b,b': Book, n,n': Name, a: Addr |
    add [b,b',n,a] and n != n' implies lookup [b,n'] = lookup [b',n']
}
```

Checking these assertions gives no counterexamples.
module tour/addressBook1

sig Name, Addr {}
sig Book {addr: Name -> lone Addr}

pred show (b: Book) {
  #b.addr > 1
  #Name.(b.addr) > 1
}
run show for 3 but 1 Book

pred add (b, b': Book, n: Name, a: Addr) {b'.addr = b.addr + n -> a}
pred del (b, b': Book, n: Name) {b'.addr = b.addr - n -> Addr}
fun lookup (b: Book, n: Name): set Addr {n.(b.addr)}

pred showAdd (b, b': Book, n: Name, a: Addr) {
  add [b, b', n, a]
  #Name.(b'.addr) > 1
}
run showAdd for 3 but 2 Book

assert delUndoesAdd {
  all b,b',b": Book, n: Name, a: Addr |
  no n.(b.addr) and
  add [b,b',n,a] and del [b',b" ,n] implies b.addr = b" .addr
}

assert addIdempotent {
  all b,b',b": Book, n: Name, a: Addr |
  add [b,b',n,a] and add [b',b" ,n,a] implies b'.addr = b" .addr
}

assert addLocal {
  all b,b': Book, n,n': Name, a: Addr |
  add [b,b',n,a] and n != n'
  implies lookup [b,n'] = lookup [b',n']
}

check delUndoesAdd for 10 but 3 Book
check addIdempotent for 3
check addLocal for 3 but 2 Book

fig. 2.7 Final version of model for simple address book.
The final version of the model discussed in this section is shown in fig. 2.7. Note that it includes the simulation predicates and assertions and their associated commands. These play the same role that test drivers and stubs play for code; they are an integral part of the development. When you make a change to a model, you can recheck the assertions and rerun the simulations just as you would run regression tests after modifying code.

### 2.3 Classification Hierarchy

In a realistic address book application, you can create an alias for an address, and then use that alias as the target for another alias. And an alias can name multiple targets, so that a group of addresses can be referred to with a single name.

Rather than elaborating our existing model, we'll just start afresh and reuse fragments of the old model as needed. We start with a classification hierarchy showing the various sets of objects and their relationship to one another:

```plaintext
module tour/addressBook2

abstract sig Target {}

sig Addr extends Target {}

abstract sig Name extends Target {}
```