Software Abstractions

Volume 2: Patterns of Modelling & Analysis

Daniel Jackson

Preface

As a programmer working for Logica UK in London in the mid 1980's, I became a great advocate of formal methods. Extrapolating from a small success with VDM, I was sure that widespread use of formal methods would bring an end to the software crisis. I began spending my daily commute on the Tube reading about them.

One approach especially intrigued me. John Guttag and Jim Horning had developed a language, called Larch, which was amenable to a mechanical analysis. In a paper they'd written a few years earlier [, and which is still not as widely known as it deserves to be, they showed how questions about a design might be answered automatically. In other words, we would have real software 'blueprints' – a way to analyze the essence of the design before committing to code. I went to pursue my PhD with John at MIT, and have been a researcher ever since.

As a researcher though, I soon discovered that formal methods was not the silver bullet I'd hoped it would be. Formal models were hard to construct, and specifying every detail of a system was too hard. Theorem proving, the kind of analysis that Larch relied on, could not be fully automated. Even now, after 20 more years of research, it still requires the careful guidance of a mathematicaly guru, and is only practical for safety-critical developments for which the cost is justified. In my doctoral work, I took a more conservative route, and worked on automatic detection of bugs in code. But I kept an interest in the more ambitious world of formal methods and design analysis, and hoped one day to return to it.

In 1992, I visited Carnegie Mellon University. By then, I'd become enamoured, like many in the formal methods community, with the Z language. The inventors of Z had dispensed with many of the complexities of earlier languages, and based their language on the simplest notions of set theory. And yet Z was even less analyzable than Larch; the only tool in widespread use was a pretty printer and type checker.

On that visit to Carnegie Mellon, Ken McMillan showed me his SMV model checker: a tool that could check a state machine of a billion states in seconds, without any aid from the user whatsoever. I was awestruck.

With the invention of model checking, the reputation of formal methods changed almost overnight. The word 'verification' became fashionable again, and the adoption of model checking tools by chip manufacturers showed that real engineers really could write formal models, and, if the benefit was great enough, would do it of their own accord.

But the languages of model checkers were not suitable for software. They were designed for handling the complexity that arises when a collection of simple state machines interacts concurrently. In software design, complexity arises even in a single state machine, from the complex structure of its state. Model checkers can't handle this structure – not even the indirection that is the essence of all software design.

So I began to wonder: could the power of model checking be brought to a language like Z? Here were two cultures, an ocean apart: the gritty automation of SMV, reflecting the steel mills and smoke stacks of Pittsburgh, the town of its invention, and the elegance and simplicity of Z, reflecting the beautiful quads of Oxford.

This book is the result of a ten year effort to bridge this gap, to develop a language that captures the essence of software abstractions simply and succinctly, with an analysis that is fully automatic, and can expose the subtlest of flaws.

The language, Alloy, is deeply rooted in Z. Like Z, it describes all structures (in space and time) with a minimal toolkit of mathematical notions, but its toolkit is even smaller and simpler than Z's. Alloy was also strongly influenced by object modelling notations (such as those of OMT and Syntropy). Like them, it makes it easy to classify objects, and associate properties with objects according to the classification. Alloy supports 'navigation expressions,' which are now a mainstay of object modelling, but with a syntax that is particularly simple and uniform.

The analysis, embodied in the Alloy Analyzer, which is freely available online for a variety of platforms, actually bears little resemblance to model checking, its original inspiration. Instead, it relies on recent advances in SAT technology. The Alloy Analyzer translates constraints to be solved from Alloy into boolean constraints, which are fed to an off-the-shelf SAT solver. As solvers get faster, so Alloy's analysis gets faster and scales to larger problems. Using the best solvers of today (2005), the analyzer can examine spaces that are several hundred bits wide (that is, of 10^{60} cases or more). Hardware advances must also get some of the credit. Even had this technology been available ten years ago, an analysis that takes only seconds on today's machines would have taken an hour back then.

The experience of exploring a software model with an automatic analyzer is at once thrilling and humiliating. Most modellers have other people review their models; it's a sure way to find flaws and catch omissions. Few modellers, however, have had the experience of having their models analyzed by an automatic tool. Building a model incrementally with an analyzer, simulating and checking as you go along, is a very different experience from using pencil and paper alone. The first reaction tends to be amazement: modelling is much more fun when you get instant, visual feedback. When you simulate a partial model, you see examples immediately that suggest new constraints to be added.

Then the sense of humiliation sets in; you discover that there's almost nothing you can do right. What you write down doesn't mean exactly what you think it means. And when it does, it doesn't have the consequences you expected. Automatic analysis tools are far more ruthless than human reviewers. I now cringe at the thought of all the models I wrote (and even published) that were never analyzed, as I know how error-ridden they must be. Slowly but surely the tool teaches you, like Pavlov's dog, to make fewer and fewer errors. Your sense of confidence in your modelling ability (and in your models!) grows.

You can use analysis not only to make models more correct, but also more succinct and more elegant. When you want to rework a constraint in the model, you can ask the analyzer to check that the new and old constraint have the same meaning. This is like using unit tests to check refactoring in code, except that the analyzer typically checks billions of cases, and you don't need to provide them.

I have sometimes characterized my approach as 'lightweight formal methods' or 'agile modelling', since it relates to traditional formal methods as extreme programming relates to traditional programming. It emphasizes incrementality, being driven by perception of risk, and making the most of automated tools to find flaws as early as possible.

But my experience in the last decade, teaching software engineering to students at Carnegie Mellon and MIT, building tools with students, and consulting on industrial developments, has convinced me of the value of the attitude to programming expressed in the earliest work on formal methods. As Tony Hoare put it famously in his Turing Award lecture:

> *[T]here are two ways of constructing a software design: One way is to make it so simple there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. []*

A commitment to simplicity of design means addressing the essence of design – the abstractions on which software is built – explicitly, and up front. Abstractions must be articulated, explained, reviewed and examined deeply, in isolation from implementation concerns. I don't mean to imply a waterfall process, in which all design and specification precedes all coding. It's separation of concerns that counts: being able to think about the abstractions independently of the code. That's why a different language is needed: one that lays out abstractions in their rawest form, free from the entanglements of execution issues, and exposed to more powerful analyses than testing or type checking.

In this respect, the language and its analysis are a Trojan horse: an attempt to draw software developers' attention away from the lure of implementation language and tools, back to thinking deeply about underlying concepts.

That is why I have chosen the title 'Software Abstractions' for this book. Today's programmers are by and large a mechanistic crowd, more drawn to clever tricks and new technologies than to elegance of structure, clarity and simplicity. But I firmly believe that, in the long term, programmers will be recognized more for their appreciation of abstractions and skill in their design than for their mastery of transient technologies, to the great benefit not only of our software industry but also of society as a whole.

Daniel Jackson
Newton, Massachusetts

This version of the text: March 17, 2005.

Contents

1: Introduction

Software is built on abstractions. Pick the right ones, and programming will flow naturally from design; modules will have small and simple interfaces; and new functionality will more likely fit in without extensive reorganization. Pick the wrong ones, and programming will be a series of nasty surprises: interfaces will become baroque and clumsy as they are forced to accommodate unanticipated interactions, and even the simplest of changes will be hard to make. No amount of refactoring, bar starting again from scratch, can rescue a system built from a collection of flawed concepts.

Abstractions matter to users too. Novice users want programs whose abstractions are simple and easy to understand; experts want abstractions that are robust and general enough to be combined in new ways. When good abstractions are missing from the design, or erode as the system evolves, the resulting program grows barnacles of complexity. The user is then forced to master a mass of spurious details, to develop workarounds, and to accept frequent unexplained failures.

Software design at its essence, therefore, is the design of abstractions. An abstraction is not a module, or an interface, class or method; it is a structure pure and simple – an idea reduced to its essential form. Since the same idea can be reduced to different forms, abstractions are always, in a sense, inventions, even if the ideas they reduce existed before in the world outside the software. The best abstractions, however, capture their underlying ideas so naturally and convincingly that they seem more like discoveries.

The process of software development should be straightforward. First, you design the abstractions, from a careful consideration of the problem to be solved and its likely future variants. Then you develop its embodiments in code: the interfaces and modules, data structures and algorithms (or in object-oriented parlance, the class hierarchy, datatype representations and methods).

Unfortunately, this approach rarely works. The problem, as Bertrand Meyer once called it, is *wishful thinking*. You come up with a collection of abstractions that seems to be simple and robust. But when you implement them, they turn out to be incoherent and perhaps even inconsistent, and they crumble in complexity as you attempt to adapt them as the code grows.

Why are the flaws that escaped you at design time so blindingly obvious (and painful) at coding time? It is surely not because the abstractions you chose were perfect in every respect except for their realizability in code. Rather, it was because the environment of programming is so much more exacting than the environment of sketching design abstractions. The compiler admits no vagueness whatsoever, and gross errors are instantly revealed by executing a few tests.

Recognizing this advantage, and the risk of wishful thinking, the approach known as 'extreme programming' eliminates design as a separate phase altogether. The design of the software evolves with the code, kept in check by the rigours of type checking and unit tests. This is why Kent Beck recommends that, despite the advantages of design notations, all real design be done in code:

> *"Another strength of design with pictures is speed. In the time it would take you to code one design, you can compare and contrast three designs using pictures. The trouble with pictures, however, is that they can't give you concrete feedback. The XP strategy is that anyone can design with pictures all they want, but as soon as a question is raised that can be answered with code, the designers must turn to code for the answer." []*

But code is a poor medium for exploring abstractions. The demands of executability add a web of complexity, so that even a simple abstraction becomes mired in a bog of irrelevant details. As a notation for expressing abstractions, code is clumsy and verbose. It's a drag on the designer, who must make extensive edits, often across several files, to explore a simple global change. And pity the reviewer who has to critique design abstractions by poring over a code listing.

An alternative approach is to attack the design of abstractions head on, with a notation chosen for ease of expression and exploration. By making the notation precise and unambiguous, the risk of wishful thinking is reduced. This approach, known as *formal specification,* has had a number of major successes. Praxis, a British company that develops critical systems using a combination of formal specification and static analysis, offers a warranty on its products, boasts a defect rate an order of magnitude lower than the industry average, and achieves this level of quality at a comparable cost.

Why isn't formal specification used more widely then? I believe that two obstacles have limited its appeal. First is that the notations have had a mathematical syntax that makes them intimidating to software designers, even though, at heart, they are simpler than any programming language, and very much simpler than the notations of UML. A second, and more fundamental, obstacle is a lack of tool support beyond type checking and pretty printing. Theorem provers have advanced dramatically in the last 20 years, but still demand more investment of effort than is feasible for most software projects, and force an attention to mathematical details that don't reflect fundamental properties of the abstractions being explored.

This book presents a new approach. It takes from formal specification the idea of a precise and expressive notation based on a tiny core of simple and robust concepts, but it replaces conventional analysis based on theorem proving with a fully automatic analysis that gives immediate feedback. Unlike theorem proving, this analysis is not 'complete': it examines only a finite space of cases. But because of recent advances in constraint solving technology, the space of cases examined is usually huge – at least billions of cases – and it therefore offers a degree of coverage unattainable in testing.

Moreover, unlike testing, it requires no test cases. The user instead provides a property to be checked, which can usually be expressed as succinctly as a single test case. A kind of exploration therefore becomes possible that combines the incrementality and immediacy of extreme programming with the depth and clarity of formal specification.

This book is the first of a three-volume set. The subsequent volumes will be devoted to a catalog of modelling and analysis patterns, and a collection of case studies. This volume introduces the key elements of the approach:

- The *logic* provides the building blocks of the language. All structures are represented as relations, and structural properties are expressed with a few simple but powerful operators. States and behaviours are both described using constraints, allowing an incremental approach in which detail is added gradually as new constraints.
- The *language* adds a small amount of syntax to the logic for structuring descriptions. To support classification, and incremental refinement, it has a flexible type system that has subtypes and unions, but requires no downcasts. A simple module system allows generic declarations and constraints to be reused in different contexts.
- The *analysis* is a form of constraint solving that uses off-the-shelf SAT solvers as its engine. *Simulation* involves finding instances of states or executions that satisfy a given property. *Checking* involves finding a counterexample – an instance that violates a given property. The search for instances is conducted in a space whose dimensions are specified by the user in a 'scope', which assigns a bound to the number of objects of each type. Even a small scope defines a huge space, and thus often suffices to find subtle bugs.

The book is aimed at software designers, whether they call themselves requirements analysts, specifiers, designers or architects. It should be suitable for advanced undergraduates, and for graduate students in professional and research masters programs. No prior knowledge of specification or modelling is assumed beyond a high-school-level familiarity with the basic notions of set theory. Nevertheless, it is likely to appeal more to readers with some experience in software development, and some background in modelling.

Throughout the book, I use the term 'model' for a description of a software abstraction. It's not ideal, because a software abstraction need not be a 'model' of anything. But it's shorter than 'description', and has come to have a well established (and vague!) usage.

To keep the text short and to the point, I've relegated discussions of trickier points and asides to question and answer sections that are interspersed throughout the text. For the benefit of researchers, I've used these sections also to explain some of the rationale behind the Alloy language and modelling approach.

The source text for all the example models in the book is [will be] available at http://alloy.mit.edu so you can explore them yourself, along with supplementary material.

2: Static Models

This chapter is about *static models*. A static model describes a system's *states* but not its behaviours – how it moves from state to state. A dynamic model, in contrast, describes not only states, but also *transitions*.

The constraints that characterize a system's states are called *invariants*. An invariant is a property of a single state; in a thought experiment, you could check that a system satisfied its invariants by passing each possible state, one at a time, through an oracle that examines the state, but has no memory of other states it has seen. The constraints of a dynamic model, on the other hand, are properties of more than one state. They may be *operations*, for example, that relate the states before and after occurrence of some event.

> *Example*. The invariants of a file system may include: that each file belongs to a single directory; that no two files in a directory share the same name; that every directory except the root is contained in another directory; etc. Operating systems usually include a program that automatically checks the invariants of the file system (albeit usually at a lower level than this), and if they don't hold, 'repairs' the file system automatically.

You can learn a lot about a system from its state declarations and invariants alone. By analogy, think of the motion of a planet. A dynamic model describes how the planet moves, and can be used to compute where the planet will be at a given time. A static model describes the planet's orbit: it tells you what positions it can occupy in space, but not how it travels from one position to the next.

In Alloy, there is no built-in notion of states, so the language is equally well suited to describing systems that are *stateless* – that is, systems that have no time-varying behaviour. There are few truly stateless systems; most are stateless because we're not interested in how the states arise.

Example. You could model a software product family by describing all possible configurations of components. Such a model might be regarded as stateless, unless you subsequently explore how members of the family are assembled, in which case it will have turned out to be, in retrospect, a static model.

Example. You might think that the structure of airspace is stateless. In most busy terminal areas around airports, however, the division of the space into sectors (which governs the assignment of aircraft to controllers) changes dynamically, with larger sectors when there is less traffic, and a single controller can handle a larger area.

The difference between stateful and stateless models is therefore not one that matters much in practice. In addition to the language itself, the patterns described in this chapter apply equally well to both kinds.

The patterns in this chapter are divided into 3 categories:

- *Elementary Patterns.* These are the fundamentals of static modelling. They include both the use of the basic modelling elements – sets and relations – and, at a larger granularity, some ideas about organizing models and the relationships between models.
- *Declaration Patterns.* These are structural patterns. Because their essence can be expressed in declarations, they have nice graphical representations. Textual constraints add precision to their description.
- *Constraint Patterns.* These are more advanced patterns that arise only in textual constraints.
- *Analysis Patterns.* These are patterns of analysis strategies: how to use automatic analysis to explore a model. They tell you how to structure the model to exploit analysis, and how to select appropriate properties to analyze.

Discussion

Shouldn't a static model describe only the static configuration of a system?

In my approach, 'static' means *about state*. In other modelling approaches (such as UML), 'static' means *stateless* instead, so that a static model describes only the aspect of a system that doesn't change over time. According to this view, there's a fundamental difference between the configuration aspect of the system and the other structural aspects that change over time. I don't find this distinction helpful, and, as I argued in Section , stateless systems have a habit of become stateful.

Does static refer to the compile-time structure of a system?

No. There are gross structural properties of a system that persist from state to state, and these are usually the easiest to express in a static model, as invariants like any others. But whether an invariant is counted as 'compile-time' or 'run-time' is an implementation issue. If you expect an invariant to hold forever – that is, you're confident you won't discover later that it actually isn't an invariant – then the more of it you can express syntactically, in a way the compiler can check, the better. But what you choose express syntactically is an implementation decision, and will be limited by your choice of programming language. And for invariants about a system's environment, the notion of compile-time structure makes no sense at all. This is not to say, by the way, that it isn't useful to use object modelling notation to describe the syntactic structure of classes and fields in a program. Just make sure you don't let these implementation issues pollute and limit your abstract models.

2.1 Elementary Patterns

Micromodel

Build many tiny models rather than one big one.

Motivation

A real software design problem is invariably multifaceted. Rather than building a single model, it's better to build a series of 'micromodels' – tiny models that capture important aspects of the problem. The micromodels will not in general be independent, and will share sets and maybe even relations.

Description

What is a model *about*? This may seem a strange question. After all, when writing a program, we don't ask what the program is about (unless we're philosophers or sociologists); a program is just a device that performs some functions. But a model is a description, and how useful it is will depend on making sure that it's describing the right thing. Surprisingly, even experts can find themselves writing and discussing models without really understanding exactly what they describe. If it's not clear what the model is about, it won't be clear what it's for, and your effort will likely be wasted. On the other hand, careful thinking about what should be modelled is often an eye-opening process in itself, revealing subtleties of the problem previously unrecognized.

The beginning of wisdom in modelling is to realize that every model should be constructed with a purpose: to answer a hard question, or to clarify an important aspect. What makes a question hard or an aspect important will obviously depend on the problem at hand. Since the motivation for all precoding development steps is to reduce risk, it will also depend on the abilities and experience of the designers and implementers. Modelling offers a flexible and inexpensive way to explore design issues while mistakes are still easy to fix.

How many models should you build? A good starting assumption is that each aspect worth focusing on deserves its own separate 'micromodel'. Separating different aspects of a system, usually called 'separation of concerns', is a goal in itself. Achieving it is arguably the central challenge of software design. If you can decompose your problem into independent parts, you'll be able to solve the problem more easily. And if you can preserve the separation in the design, the resulting system will be easier to understand and more maintainable.

> *Example.* Suppose you're designing a media viewing application for digital photos, clipart, movies, fonts, and so on. Some of the aspects you might model independently are: the assignment of properties to assets (such as exposure for photos and frame rate for movies); annotation by the user (for example, listing the people in each photo); version control and derivation relationships (for example, tracking the fact that one photo is a retouched version of another); organization of assets into possibly overlapping sets; mapping of assets to their location on disk, and handling backups, DVD burning and moves between file systems; slide show parameters and execution; etc.

Here's another way to think about this. When starting work on the design of a system, ask yourself: what problem does this system solve? For any but the most trivial system, the problem is best understood as a collection of related subproblems. Each subproblem can be addressed separately, with one or more models.

> *Example.* You're designing a controller for a home automation system that will dim lights, activate lawn sprinklers, etc. At least 3 subproblems come to mind: the problem of creating, editing and storing schedules; the problem of configuring the system initially, determining which appliances are connected to which sockets, and giving them appropriate names; and the problem of executing a schedule of instructions, handling overrides, power failures, etc.

A common mistake made by novices is to rush into modelling before figuring out what should be modelled and why. Every noun that appears in the description of the system becomes an atom set; every relationship you can think of becomes a relation. A model built in this manner is likely to be both superficial and complicated at once.

Good models are usually small. If your model looks like a database schema, you've probably done something wrong. You should be able to justify the presence of each element of your model; later, I'll give some hints for identifying irrelevant elements that can be removed.

Examples

[To be completed. Address book and message store. Show you can be more general by splitting.]

Related Patterns

Database Schema: Often the result of a failure to separate concerns.

Questions

When you come to implement the system, won't you need to put the micromodels together?

The multiple models that you build must eventually be combined in a single implementation, of course. This may be tricky, but it's not a reason to build a single monolithic model. If you did that, you'd miss every opportunity to discover aspects of the problem that are truly independent (and can be implemented separately), and you'd be overwhelmed with the complexity of dealing with everything at once. Michael Jackson argues that the composition of fragmentary solutions should be regarded as a problem in itself, distinct from the problems each aspect poses ([]).

How do you select the aspects of the problem to model?

This question is addressed by Michael Jackson in his book *Problem Frames* []. A 'problem frame' is a kind of problem template. It shows the prototypical structure of a particular class of problem, and the models that are needed to describe it. Each problem frame has its own 'frame concerns' – tricky issues that arise in that class of problem – and a standard solution. Each frame thus corresponds to a standard class of problem with a known solution; it's like a design pattern at the system level. (Note, incidentally, that Michael Jackson uses the term 'model' to mean a representation inside the program of the state of the environment, and prefers the tern 'description' for what we are calling models.)

Where does the idea of separation of concerns come from?

Such a basic idea is fundamental to scientific thinking, and goes back a long way. But in computing, Edsger Dijkstra is usually credited with recognizing its importance, and he coined the term 'separation of concerns':

> *Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects.*
>
> *We know that a program must be correct and we can study it from that viewpoint only; we also know that is should be efficient and we can study its efficiency on another day [...] But nothing is gained –- on the contrary –- by tackling these various aspects simultaneously. It is what I sometimes have called 'the separation of concerns' which, even if not perfectly, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by 'focusing one's attention upon some aspect'; it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously. []*

How to exploit separation of concerns in software development has inspired much research, from work in requirements and specification on viewpoints [] and views [,], to more recent work on implementation strategies for maintaining separation, such as aspect-oriented programming [], which offers a syntactic mechanism for weaving together distinct programs, based on ideas from Common LISP, and subject-oriented programming [], which is more semantic in flavour, and works by synchronizing method calls across two distinct class hierarchies.

Approximation

Loosen the constraints of a model to allow approximations of reality..

Motivation

The representation of the real world inside a software system may not match the reality. This creates a need for looser models that accommodate representations that violate expected real world constraints. It also suggests paying careful attention to the nature of the approximation, and sometimes developing two separate models.

Description

A software system often maintains a representation of the state of the real world, which it updates with information provided by users or through sensors, and which it uses to answer queries about the world, or to control actuators. Because the information received by the system is not fully accurate, this representation may not obey the constraints you'd expect of the real world itself.

When you build a model for such a system, it's important to know what you're modelling: the real world itself, or the approximate representation. Often it will be good enough to model only the approximation. As you do so, you'll have to bear in mind that the constraints that apply in the real world don't necessarily apply in the approximation.

> *Example.* An application for recording a family tree must be based on a model of genealogical relationships that doesn't correspond exactly to reality. Because of uncertainty or a lack of information, there will be people recorded who have no parents, and who have more than one biological mother.

When only small parts of the model are approximate, you can create special features of the model to accommodate and contain the approximation.

> *Example.* In the family tree application, you might introduce a set Relationship to represent the possible relationships

amongst family members (mother, father, sibling, etc). This set could then be split into two subsets, corresponding to those relationships that are certain and those that are speculative. The genealogical invariants would then be enforced for the only the certain ones.

Sometimes it will be possible to impose real-world constraints on the approximation by making assumptions about what's happening without full information. It usually requires considerable domain expertise to figure out whether this makes sense.

Example. A network router maintains an approximate internal representation of the network as a whole. Connectivity in the real network must be transitive: if packets can flow from `A` to `B`, and from `B` to `C`, then they can also flow from `A` to `C`. But whether or not connectivity in the router's representation should be transitive is another matter.

For a critical system, the approximation itself can be the cause of failures unless handled correctly, so it's sensible to build two separate models, one for the real world and one for the approximation, and to explore their relationship carefully.

Example. A software system for controlling a radiotherapy machine may maintain a record of which patients, doctors and technicians are present in a treatment room. This record may not match the reality. A good design would attempt to ensure that the relation that approximates occupancy always contains the actual occupancy relation, so that it errs on the side of preventing treatment, rather than erroneously radiating an unknown occupant.

Examples

[To be completed.]

Related Patterns

Machine, Environment, Requirement: A more general view of the relationship between a machine and its environment.

Discussion

[To be completed.]

Classification

Organize the universe into a hierarchy of sets

Motivation

The first step in building a model is to consider which things are relevant, and how they should be classified. The result of this step is a collection of set names, each denoting some set of atoms, and a classification hierarchy relating the sets. Atoms can be classified not only by their static properties, but by dynamic properties too, so that an atom's membership in a set changes over time.

Description

The Form of the Hierarchy

It's easier to explain what the result of classification looks like than to explain how to get there, so let's start with that. A classification hierarchy consists of a collection of set names, arranged into one or more trees, with the following interpretation:

- the children of a set represent subsets (that is, sets whose elements also belong to the parent set):
- the subsets of a set at the same level are mutually disjoint (that is, they share no elements);
- if a set is marked as 'abstract', it's covered by its subsets (that is, each of its elements belongs also to one subset).

The key property of this structure is that every atom has a single, most precise classification: the set it belongs to that is lowest in the hierarchy.

> *Example.* The objects of a file system belong to a set `Object` with subsets `Directory` and `File`. The set `File` has subsets `Alias` and `Executable`. The set `Object` is abstract, but `File` isn't, since there are files that are neither aliases nor executables.

The classification hierarchy can be embellished with additional *sidesets*: subsets that classify the atoms orthogonally. Remember (from Section) that sidesets are not by default mutually disjoint (from each other or from the standard subsets), and that they are indicated by a dotted arrow (rather than a solid arrow) in the graphical notation and the keyword `in` (rather than `extends`) in the textual notation.

> *Example.* A subset of `Object`, the set of objects of a file system, might be `LocalObject`, the set of objects stored locally rather than on a remote server.

The Meaning of the Hierarchy

The meaning of a classification hierarchy is a collection of instances, each consisting of a universe of atoms, and, for each set name, a subset of the universe. These instances can correspond to possible worlds, in which case a set represents a fixed property of an atom. But they can also correspond to states, in which case a set represents a property of an atom in a given state. The classification is then 'dynamic', in the sense that an atom's membership in a set can change over time.

> *Example.* The set `File` might have a subset `OpenFile`, which a file belongs to when it has been opened for reading or writing but not yet closed.

Remember that even when you're modelling states, the model describes the set of possible states by giving constraints on a *single* state. Novices often get confused about this. Two sets are disjoint if they never share an element *in a given state*; an element can belong to one set in one state and another set in another state, and they are still disjoint.

Building A Classification

How do you build a classification hierarchy? Start by writing down a candidate list of top-level sets. Think first about any physical domains, and then consider more abstract ones. Review the list, checking each set against these criteria:

- *Relevance*: Do the atoms of the set really belong to the subject being modelled? In particular, if you're modelling a problem, are they part of the problem or part of the solution?
- *Meaning*: Is it clear what you mean by the set name? Can you visualize some atoms? Given some atom, would you be able to

tell whether it belongs to the set? Can you distinguish two elements of the set? Can you imagine counting them?

- *Coherence*: Do the members of a set share some basic properties?
- *Unary*: Is the set really a classification of atoms, or is actually about a relationship *between* atoms?

If a set fails any of these tests, drop it, or replace it with a better choice. Check that each set has at least some unique property, relevant to the model, that characterizes its elements; if not, consider merging sets. If two sets share a property, consider introducing a superset which they become subsets of.

> *Example.* Suppose you're modelling the storage and retrieval of messages in an email client. You might start with this list: `Email`, `Client`, `Message`, `Mailbox`, `ToAddress`, `IMAP`, `Alias`. The set `Email` fails the meaning test: what would its elements be? The set `Client` fails the relevance test, since the model isn't about version control, installation or distributed implementations, or anything that involves multiple clients. The set `IMAP` represents a particular atom and not a set (see the anti-pattern); if protocols are relevant, the set should be `Protocol`, but they're probably not. The set `ToAddress` fails the unary test: an address is only the 'to address' of a particular message, so replace it with `Address`. The sets `Message`, `Mailbox` and `Alias` are reasonable.

Now expand the list, and begin to construct a hierarchy, introducing new subsets, by asking youself these questions:

- *Refinement*: Is there an obvious classification of the elements of a set into further subsets?
- *Roles*: Do the atoms of a set play different roles that might be used to classify them?
- *States*: Do the atoms of a set have important properties that change from state to state?
- *Attributes*: Do the atoms of a set have attributes whose values are drawn from small sets?

- *Special members*: Are there particular members of a set that are special?

Don't worry if a subset has only a single element; that's fine (see). As you consider new subsets, explore their position in the classification hierarchy. If you find there are two orthogonal ways to classify the elements of a set, pick the one that is richer or more important.

> *Example.* Take the email client. The set `Message` can be refined into `SentMessage` and `ReceivedMessage`. Considering the state of a message suggests an additional subset `DraftMessage` for messages under construction. The set `Mailbox` has special members `Inbox` and `Outbox`. The members of `Alias` play two roles: some represent groups, and some represent individuals. So you might split the set into two subsets `Nickname` and `Group`. If you're not concerned about supporting multiple users, you might regard `ReplyAddress` as a special (singleton) subset of `Address`. Thinking about attributes, you might consider the format of a message relevant; if so, you might want to classify `Message` into `HTMLMessage` and `TextMessage`. This is an orthogonal classification that's less important than the draft/sent/received one, so it wouldn't be the primary classification, and you'd add these as sidesets. If every message is either text or HTML, you don't need both sets, so you drop `TextMessage`. Thinking about states, you realize that whether a message has been read or not is relevant, so you introduce a subset `UnreadMessage` of `ReceivedMessage`.

Now take a look at the emerging classification hierarchy. Ask yourself if it is balanced – whether sets are refined to a similar degree. It need not be, but in considering this you may find that you've focused your attention too much on one of the trees at the expense of the others, that you've missed some essential refinements, or that you've got carried away with gratuitous detail.

> *Example.* For the email client, the current hierarchy is shown in Figure . You notice that the set `Mailbox` seems to have received less attention than `Message`. Perhaps you should classify mailboxes according to whether they have been synchronized? You decide this is not relevant right now, and no other

classifications of mailbox come to mind, so the classification is reasonable.

Related Patterns

- *Designations*: To make your model precise, give each set a designation.

Multiplicity: It's useful to consider the multiplicity of each set as you construct the classification hierarchy.

Singleton: Sets with a single element.

Time confusions: Disjointness across states, eg.

Discussion

Does an abstract set contain no elements?

No, that's a common confusion. In a programming language like Java, an abstract class can't be the runtime type of an object, so it's common to think of the class as having no objects of its own. But in modelling terms, an abstract set contains all the elements of its subsets, but no additional ones.

Why do you talk about 'atoms' rather than 'objects' or 'entities'?

These other terms have extra connotations I want to avoid. Objects have local state, and 'entities' are sometimes distinguished from 'values'.

How does classification relate to inheritance?

Inheritance is a mechanism for sharing code fragments. Even at the implementation level, inheritance introduces complications that are often best avoided (eg, by using delegation instead). At the level of abstract modelling, it's is a red herring, and a term best avoided, along with 'class' and 'subclass'.

Are subsets subtypes?

The subsets in the primary classification are treated by Alloy as subtypes. This allows errors to be caught in which one subset is confused for another. But don't confuse Alloy's subtyping with that of programming languages. It's much simpler: there are no casts, and the type checker doesn't produce false alarms. Also, the notion of subtyping in programming languages is tied up with the idea of encapsulated mutable state. In Alloy, Square can be a subtype of Rectangle because every square is a rectangle. But in Java, if Rectangle offers a method resize that changes the aspect ratio, the subclass Square cannot be a subtype.

Must atoms that are classified dynamically have mutable state?

No, that's an implementation issue. Dynamic classification does imply the presence of mutable state somewhere, but not in any atom. Atoms are by definition immutable, and have no local state of their own. This is good, because it avoids premature implementation decisions. For example, the classification of files into open/closed might be implemented (as in Unix) with a global table, and not within the files themselves.

Why don't you divide atoms into 'entities' and 'values'?

Because it's not a distinction that can be easily sustained. You can't say that entities have mutable state, because no atoms do. You can't make distinctions based on notions of equality: two atoms are equal if they are the same atom (that is, there aren't two of them!), and you can define useful notions of equivalence over any set. You can't say that values are primitive datatypes, because truly primitive types (boolean, integer, real) are – surprisingly – almost never useful as sets in a model, and any important domain-specific set (name, dollar amount, social security number, shutter speed, distance) is likely to be implemented as an abstract datatype. Any attempt to come up with a plausible distinction gets mired in needless complexity, and doesn't really help anyway.

Can a set have two supersets?

Yes, but you can't declare a set as the subset of more than one set in Alloy. Just make it a subset of one, and write a textual constraint making it a subset of the other also.

Can a set have elements drawn from two supersets?

Yes. For example, in a model of an email client's synchronization, you might introduce a subset Available of Message+Mailbox containing the messages and mailboxes that are available when offline. In the Alloy textual notation, you would declare the set like this:

```
sig Available in Message+Mailbox {...}
```

In the graphical notation, you can draw a contour around the two sets Message and Mailbox, and draw a dotted subset arrow from the box labelled Available to this contour.

Is classification about is-a relationships?

The term 'is-a' has long been used in semantic data modelling to describe the subset relationship, and if it helps you, feel free to use it. Martin Fowler doesn't like it though [, p. 96]. He points out that, by transitivity, we should be able to infer from

- Shep is a Border Collie.
- A Border Collie is a Dog.
- A Dog is an Animal.
- A Border Collie is a Breed.
- Dog is a Species.

that 'Shep is a Breed', and 'Dog is a Species', and that the is-a notion is therefore dangerous. He blames this on a confusion between 'classification' (the object Shep is an instance of the set Border Collie) and 'generalization' (the set Border Collie is a subset of the set Dog).

I see no problem here. In Alloy, as we have seen, a scalar is indistinguishable from a singleton set, and there is nothing wrong with a classification hierarchy that includes the chain Shep-BorderCollie-Dog-Animal. Arguably, the conventional distinction between membership and subset is an artifact of typed set theory, and not the most natural way to think about sets and their elements.

The real problem here is a lack of designations (see designation-pattern). The term 'Dog' is used with two different meanings: to refer to the set of all dogs (as in 'A Dog is an Animal'), and to refer to a taxonomic notion invented by zoologists (as in 'Dog is a Species').

Where else can I find advice about classification?

A nice discussion of classification may be found in Roel Wieringa's book []. He gives some useful heuristics for evaluating a classification hierarchy. But I take issue with his 'principle of comparable size': that the sizes of the subsets of a set should be comparable. This may be valuable in database design, but for abstract modelling it's not compelling. In a file system, it surely makes sense to classify directories into the root directory and the rest, even though there are in practice thousands of directories but only one root.

Why don't you distinguish static and dynamic classifications linguistically?

Surprisingly, it's hard to come up with a general principle that makes the distinction precisely. On the other hand, it's usually easy to say exactly what changes are allowed over time for some particular sets; see .

Some languages (such as Z) allow you to say that the sets in a classification don't change over time, but that's too strong: it doesn't allow entities to appear and disappear. The informal claim that a set B forms a static classification of the set A might be formalized in Alloy like this. First we declare A and B as fields of a state signature:

```
sig State {A, B: set univ}
```

so that in a given state s, the sets s.A and s.B denote subsets of the universe. Now we can write a predicate that will hold for each state transition from s to s':

```
pred staticB (s, s': State) {
  all a: s.A & s'.A | a in s.B iff a in s'.B
  }
```

which says that every object a that is a member of the set A in both the pre-state s and post-state s' is a member of B in the pre-state if and only if it is a member of B in the post-state. In other words, an A object that is neither appears nor disappears in this transition does nto change its B membership. The subtleties of this definition begin to appear when you look more closely. Must A itself be static, or can you have a static classification of a dynamically classified set? And must the states s and s′ be consecutive (or can an object die and be reborn under a different classification)? Until someone finds a general, elegant and simple solutions to these problems, it's better just to handle each case on its own merits, and use the full power of the logic to write the dynamic constraints you want.

Can dynamic classifications be primary?

Yes. There is no reason to relegate dynamic classification to secondary status, with the primary classifications all being static, and dynamic subsets all appearing as sidesets. In constructing and analyzing static models, it doesn't matter whether a classification is static or dynamic, because we only consider one state at a time. For a dynamic model, even the static classifications will not usually be represented textually by signature extensions, since the value of a signature is fixed, and the value of a set in a static classification changes over time (as objects appear and disappear).

[To be completed: add diagrams to this section.]

Relations

Model relationships of all kinds with relations.

Motivation

All kinds of relationships can be expressed with binary relations. Using relations alone, rather than other kinds of structure, avoids implementation bias (because it's abstract), and results in a simple and uniform syntax for constraints.

Description

Because a relation can be viewed as a table, many people mistakenly think of relations only for representing stored information. But a relation is just a way of recording some properties about some set of things. Properties of individual things are expressed with relations with arity one (that is, sets); properties that involve relationships between pairs of things are expressed with relations of arity two; properties involving three things at a time are expressed with relations of arity three, and so on. This pattern is about relations of arity two – binary relations, or just 'relations'.

Orientation and Naming

Relations are directional, but the choice of direction is arbitrary – at least in theory. Given a `parents` relation from children to their parents, a tuple that maps `Alice` to `Bob` tells us that Bob is a parent of Alice, but a tuple that maps `Bob` to `Alice` makes Alice the parent and Bob the child. But you could equally well declare the relation `children`, containing the same tuples of `parents`, but reversed. And of course you could change the meaning of `parents`, and say that it maps parents to their children. In practice, however, it's not a good idea to choose relation directions arbitrarily, and there are some guidelines worth following:

- The name and direction of the relation should be chosen so that the navigation expression `x.r` can be read intuitively. Thus `Alice.parents` should be Alice's parents; `f.dir` should be the directory containing the file `f`; `m.hostName` should be the host name of the machine `m`, and so on. Requiring the relation always to be a noun is too draconian, though. Often a verb or very phrase works well, so that if the relation `r` contains a tuple from `a` to `b`, you can read 'a r b' as a sentence. For example, the relation that maps file system users to the files they can access might be called `canAccess`, so that `u.canAccess` becomes the files that user `u` can access.

- Some relations can be viewed naturally (but informally) as 'attributes' of objects, either because they assign properties that are viewed as intrinsic, or because the set of attributes is a target of other relations too. Such a relation should map from the object to the attribute. For example, a relation that associates names with files would be better declared as a relation `name` from files to names, than as a relation `files` from names to files. The Alloy textual notation encourages this viewpoint, since the relation is declared as a field of the signature corresponding to its domain.
- Ease or difficulty of finding an appropriate name can be a useful indicator. If a short and unambiguous name comes readily to mind, consider orienting the relation according to that name. For example, looking for a relation to describe ownership of files, the name `owner` naturally arises, suggesting a mapping from files to the users that own them.
- When you have several relations that form a group conceptually, you probably want to orient them in the same direction. For example, if you have a relation for file ownership and another for ability to access, you wouldn't want to declare them as `owner` from file to user and `canAccess` from user to file. It would be better to declare the first as `owns`, so they appear together in the same signature, and constraints such as

```
all u: User | u.owns in u.canAccess
```

can be written more uniformly.

Common Idioms

Here are some of the common kinds of properties expressed with relations:

- *Containment.* To express containment, declare a relation that relates `a` to `b` when `a` contains `b`. If contents can't be shared, the relation is injective (that is, has multiplicity lone-many). When the domain and range of the relation are not distinct sets, the relation is a self-relation (see).

Example. `contents: Directory lone -> Object` maps a directory in a file system to its contents.

- *Labelling.* To express the idea of attaching a label `l` to an entity `e`, declare a relation that maps `e` to `l`. Usually the relation will have a many-to-one multiplicity: every entity in some set will have a label.

  Example. `type: File -> one Type` maps a file to its type.

- *Naming.* A special case of labelling; the relation from an entity `e` to its name `n` will usually have multiplicity lone-to-one, so that names are unique.

  Example. `blockId: Car lone -> one EngineBlockId` maps a car to the identifier inscribed on its engine block.

- *Linking.* A relation often captures a relationship between peers in which neither plays a subordinate role; the tuples of the relation can be thought of as links, and the relation typically has weak multiplicity constraints.

  Example. `connects: TrackSegment -> TrackSegment` relates a track segment in a railway layout to the segments it's connected to.

- *Priority.* A relation can express different kinds of ordering, such as a priority ordering. The relation might relate `a` to `b` when `a` is the immediate predecessor (or successor) of `b`, or it might relate `a` to `b` when `a` is any predecessor (or successor) of `b`; the second relation is just the closure of the first.

  Example. `next: Job lone -> lone Job` maps a job in a print queue to the job that follows it.

- *Grouping.* A set can be grouped into subsets by a relation that relates `a` to `b` when they belong to the same subset. The grouping relation will be symmetric. If every atom belongs to a group, it will be reflexive; if groups don't overlap, it will be transitive; and if both conditions apply it becomes an equivalence relation.

Example. `similar: Photo -> Photo` relates photos that a matching feature deems to be visually similar.

- *Mapping to a set.* Although a relation relates individual atoms, if it relates more than one atom with a given atom, it can be viewed as mapping an atom to a set. This often allows a single relation to be used where novices might introduce unnecessary additional structure.

  Example. A camera might flash a warning light when a shutter speed is selected that is likely to result in a blurry image because of camera shake. The set of speeds that produce warnings for a given focal length might be modelled with a relation `shakes: FocalLength -> ShutterSpeed`.

Related Patterns

- *Non-Relations*: sloppy use of relations to model things that aren't relations.
- *Classification*: describe properties of single atoms, rather than pairs.
- *Self Relation*: a relation that relates elements of a single set.
- *Unordered Relation*: representing unordered relationships with a symmetric relation.
- *Multirelation*: relationships involving more than two atoms at a time.
- *Composite*: a common idiom for modelling composite structures.

Questions

Do relations always correspond to information that is stored?

No! This is a common misunderstanding, abetted perhaps by familiarity with relational databases. A relation is just a more convenient way of packaging a property than a predicate. A property `S` of a single entity can be modelled in logic by a one-place predicate `S(x)`, with `S(e)` being true when the particular entity `e` has the property `P`. In our relational logic, the predicate becomes a set `S`, and the property holds for the entity e when it is a member of the set: x in S. Similarly, a relationship R of two entities would be modelled in predicate logic as a two-place predicate R(x,y), with R(e1,e2) being true when entities e1 and e2 are related by the property. In our logic, the predicate becomes a relation R, with e1->e2 in R when e1 is related to e2. Don't be confused by the fact that a relation is viewed more naturally than a predicate as a table: the table might be very large or even infinite.

Should the relation direction correspond to the direction you expect to navigate it in the code?

No! Navigation in the code is a low-level implementation issue. There are many reasons a relation may never become a field in the code. It may not represent information that will need to be stored during program execution. Even for those relations that will end up in the program state, you should postpone thinking about how they'll be implemented. If you don't, you're likely to obscure and clutter your model with implementation details. Moreover, if you work hard to keep your relations abstract, and carefully separate the concern of designing your abstract model from the concern of how to implement it, you'll have more freedom in the end to choose a good implementation. For example, once you know which relations have values that change over time, you can organize them into fields so that as many classes as possible are immutable.

Aren't some relationships undirected?

Yes – the relationship between siblings, for example. See for how to model such cases.

Don't you need to worry about overloading when you choose direction?

No. Alloy's signature construct gives you a convenient way to organize relations by their domains. But it doesn't restrict your flexibility in how you use them. In Java, when you write an expression x.f, the field name f will be resolved according to the type of the expression x. In Alloy, the full context of a field reference is used to resolve overloading. So if you write x.f = y, for example, the type of y can be exploited in determining which field named f is meant.

Why doesn't Alloy represent attributes differently from other relations?

Because the distinction is not useful enough to hard wire into the language. The notion of attribute is intuitively appealing, but all attempts I know of to distinguish attributes rigorously from other kinds of relation fall down on close inspection. Sometimes, the argument goes, rather than relating two things, we want one thing to be regarded as a property of another: a person's name, a train's position, a photo's orientation. But how to make this intuition more precise? Is there a modelling principle that would tell you when something should be an attribute and when it should be a relation?

Let's try some candidates. Perhaps an attribute is something relatively unimportant: some data that will eventually be needed in an implementation, but not playing a major role in design. If so, best just to omit it. Modelling is all about focus; if it's important make it a relation, if it isn't, ignore it. Paradoxically, data that will be essential in the implementation can often be ignored completely in modelling. In the design of an application for viewing and organizing photos, for example, we might legitimately choose to ignore the actual picture content of the photos, knowing that rendering images is a solved problem, handled by a library call.

Perhaps attributes are for immutable objects: objects that are 'values' and don't have 'state'. This distinction is not a helpful one at all; which entities do and do not have local state is an implementation issue. Think about designing a domain name server. Is a domain name an attribute of a host machine? Or is the machine an attribute of the name that refers to it? In implementation terms, the latter is much closer to the truth. Even in a relatively low-level model, the distinction is hard to justify. Suppose you represent the name of an object as an attribute, intending to implement it in Java with a `String`. If you decide, for some obscure performance reason, to use a `StringBuffer` – a mutable version of `String` – instead, will it no longer be an attribute?

Naming, incidentally, is frequently relegated to 'attribute' status in modelling textbooks. But in practice, few issues are more important than naming. In fact, I'll stick my neck out and state Jackon's law of naming: either names matter a lot, or they don't matter at all. If they matter, use relations to model them; if they don't, just omit them. Take our photo application. The filename of a photo clearly matters a lot, and operations that rename photos in batch have to be designed very carefully to avoid nameclashes. But captions, another form of naming, are probably of no significance. Although a photo's caption may matter to the user, it is probably just like any other field that can be edited, displayed and used in sorting, and need not be modelled explicitly.

Perhaps attributes represent 'primitive datatypes'. Again, we're on shifting sands. The notion of a primitive datatype is programming-language dependent, and the very invention of abstract data types was intended to undermine this distinction. Are integers primitive if you implement your own abstract type for handling big numbers? And what about dates, which are anything but primitive (just take a look at the Java library)?

Michael Jackson prefers the term 'individual' to entity in order to get away from the entity/attribute distinction. 'The fact "Lucy's shoe size is 4" is symmetrically about the individual Lucy and the individual 4. It does not mean that 4 is the value of some attribute of Lucy any more than it means that Lucy is the value of some attribute of 4.' []. He admits in private, however, that it may be possible to make the notion of attribute precise, and suggests that those individuals that have no time-varying classifications (that is, one-place predicates) might qualify as attributes. So a person should be an entity and not an attribute because of properties like `married` and `employed`, whereas an integer should be an attribute, since its classifications, such as `prime` and `even` don't change from year to year. A name in a file system might have a property `used` and thus be an attribute. Perhaps such a distinction is tenable philosphically, but it seems too subtle to be useful as a modelling principle.

Complaints about attributes aren't new to object modelling. In his classic book on data modelling [], William Kent says: 'I don't know we should describe "attribute" as a separate construct at all. I can't tell the difference between attributes and relationships.' He suggests that a modelling language should have one construct, with two equivalent names. Then we can use whichever feels better at the time! Alloy actually does exactly this. The underyling construct is the relation, but in visualizing instances especially, you sometimes want to view a relation as an attribute. The Alloy Analyzer offers such an option in its visualizer; when you select it, the relation is shown as a label of the node, rather than an edge to another node. In practice, this is quite handy, and you do indeed find yourself wanting to show the very same relation with arrows on some occasions and as labels on others.

Is a relation the same as UML's assocation? And what about aggregations and compositions?

A UML association is a pair of relations, each the transpose of the other. The relations are called 'role names', and are written at the 'association ends'. For example, an association between Course and Student with role names courses (at the Course end) and students (at the Student end) corresponds to two relations

```
students: Course -> Student
courses: Student -> Course
```

related by the constraint

```
students = ~courses
```

In Alloy, we usually introduce only one name (say students), and then use the transpose expression (~students) for the other. Of course, if you find yourself writing many constraints involving both, it's convenient to introduce the second name as well, as a definition (see definition-pattern).

UML distinguishes aggregations and compositions from regular assocations. An aggregation is supposed to model some kind of ownership; composition is a stronger form of aggregation in which the component parts are not shared. These notions are hard to pin down precisely, and they don't seem to be very useful. Martin Fowler says that aggregation is 'one of my biggest betes noires' [] and Jim Rumbaugh describes it as a 'modelling placebo' []. Rather than complicating the basic notion of relationship, it's better to add constraints to say exactly what you want (see ,).

Multiplicity

Express basic counting properties of sets and relations

Motivation

Multiplicity is about counting – for a set, the number of elements it has, and for a relation, the number of atoms on one side associated with an atom on the other. Counting is fundamental, and when you think about the multiplicities of sets and relations, you often uncover important subtleties. It's useful to consider multiplicities of sets during construction of the classification hierarchy, and to consider multiplicities of relations when adding them to the model.

Description

Although the idea of multiplicities is simple enough, assigning multiplicities in a model can be tricky. It's all about counting, and when you count entities, you need to be able to distinguish one from another clearly.

> *Example.* In a unicode font, how many typographic symbols can a unicode value refer to? At most one, you might think – after all, there are 16 bits available for each symbol. If you asked a font designer this question, however, you'd discover that different typographic appearances of the same semantic character are given the same code. You might then recognize that your notion of 'typographic symbol' should be refined, and you need two distinct sets: `Character`, representing semantic symbols ('upper case A'), and `Glyph`, representing typographic images ('titling A', 'swash italic A', etc).

Moreover, important anomalies sometimes lurk within multiplicities. So elaborating your model with multiplicities should not be viewed as a trivial step: it usually requires some hard thinking, and raises productive questions.

> *Examples.* Zip codes, which are postal zones in the United States, are usually associated with unique states, but not always: some zipcode areas cross state boundaries. In most

file systems, a file belongs to a single directory, and aliases or links are distinct from the files they point to. Not in Unix; creating a hard link to a file results in a new inode entry indistinguishable from the original one.

Alloy's multiplicity notation uses the following keywords, each with an associated punctuation symbol that can be used in its place in diagrams:

lone	?	zero or one
some	+	one or more
one	!	exactly one
set	*	any number

Remember that the default multiplicity is set, except for textual declarations of the form v: e where e is unary, in which case the default is one; details are in Sections and . In a diagram, drawing a set as an oval gives it a multiplicity of lone.

As you construct a classification hierarchy, ask yourself how many members each set may have. Be skeptical about any set that you're tempted to constrain with a multiplicity. A multiplicity of some often represents an unnecessary limitation of functionality, and a multiplicity of lone or one may indicate a set that should really be a relation.

Examples. In a file system, you'd likely give the set Dir the multiplicity some, since every file system must at least have a root directory; the set Root itself would get the multiplicity one. The set File should not have a multiplicity of some: there's no need to require the file system to be non-empty. Considering assigning one to the multiplicity of the set Desktop, you might realize that the set is actually a relation, mapping each user to a desktop directory.

Two confusions are common. First, remember that a multiplicity constraint is semantically a constraint like any other. Saying that a set has a multiplicity of lone, for example, doesn't imply that there exists a world in which it has no elements; it just means that there are certainly no more than one. A model that hasn't yet been elaborated with multiplicities is just incomplete, not wrong.

Examples. The models of classification-pattern don't include multiplicities. You can't infer from the model of the email client, for example, that it must be possible to have no inbox.

Second, understand that multiplicities, like other constraints, describe a particular instance. If the model describes the states of a system, that means the multiplicity constraints talk about individual states. So a set with a multiplicity of lone can still contain different elements in different states.

Examples. A model of an email client might assign the multiplicity lone to the set ActiveServer, because the client can only handle one active server at a time, even if it can have many active servers over time.

Examples

Related Patterns

- *Designations*: Thinking about multiplicities can help you refine your designations.
- *Singleton*: A subset with a multiplicity of one.
- *Lowering*: Introducing a subset to allow more expressive multiplicity constraints.
- *Tight Declaration*: Using more elaborate expressions in declarations to exploit multiplicity.

Questions

You say that a multiplicity constraint doesn't imply that cases exist with all allowed counts. But mustn't the implementation support all these cases?

Yes. Eventually, when you implement a system based on your models, you will need to account for all the looseness in multiplicities that remain. If your file system model, for example, doesn't specify that there is a single root, you'll need to handle the possibility of multiple roots.

Aren't you abusing multiplicities to distinguish sets from scalars?

No, I'm taking advantage of the uniformity of Alloy's semantics. It might be tempting to think of relations and sets that differ only in their multiplicities as qualitatively different: for example, to view a singleton as a different kind of set from a set with two elements, or to view a relation that happens to be a total function (that is, mapping every element in its domain to exactly one element in the range) as different from all other relations. This turns out to be trouble though, and introduces a host of needless complications.

Most modelling languages treat sets and scalars differently, and pay the price with special syntax for converting between one and the other. OCL, the constraint language of UML, treats functions differently from other relations. When you navigate from an element outside the domain, you get a special undefined value in the case of a function, and an empty set in the case of a relation. So you need to use a different syntax for navigating different kinds of relation. And declarations are no longer just constraints: declaring a relation with multiplicities is different from declaring it without multiplicities and adding an explicit constraint.

Multirelation

Model relationships amongst three or more entities at a time with relations of higher arity.

Motivation

Some relationships involve more than two entities at a time. Just as a relationship involving two entities is expressed with a relation of arity two, so a a relationship involving three entities is expressed with a relation of arity three, and so on. Multirelations – relations of arity three or more – are easy to use, and more flexible than the special notions provided by some other modelling approaches (such as qualifiers).

Description

A multirelation is no different conceptually from a binary relation (or for that matter, a set): it's simply a collection of tuples. A three-way relationship, for example, will contain a triple such as `a -> b -> c` when the atoms `a`, `b` and `c` are related.

Example. In a file system, the relation `object` maps a name to an object in the context of a particular directory. The relation has arity three, with columns `Dir`, `Name` and `Object`, and can be shown as an arc from `Name` to `Object` labelled `Dir.object` (Figure 99), or declared textually as:

```
sig Name {}
abstract sig Object {}
sig Dir extends Object {object: Name -> lone Object}
```

Example. A network router translates a target address into an outgoing port. A network with many routers can be modelled with a ternary relation `port`

```
sig Address, Port {}
sig Router {port: Address -> lone Port}
```

shown graphically as an arc labelled `Router.port` from `Address` to `Port`.

Example. A matching feature for finding duplicates in a photo catalog, with the level of similarity required for matching set by the user, might be modelled as a relation matches declared as:

```
sig Photo {}
sig Level {matches: Photo -> Photo}
```

and shown graphically as an arc labelled Level.matches from Photo to Photo.

Ordering the Columns

As with binary relations, directionality is significant. But when there are more than two columns, direction is not the only choice: for a ternary relation, you have 6 ways to choose how to order the columns!

Example. To represent the format settings of styles in a word processor's stylesheet, you might declare a relation amongst three sets: Style, the set of style names (such as 'heading'); Attribute, the set of formatting attributes (such as 'type face'); and Value, the set of values of the attributes (such as 'Garamond'). The six options are:

```
sig Style {f: Attribute -> Value}
sig Style {f: Value -> Attribute}
sig Attribute {f: Style  -> Value}
sig Attribute {f: Value  -> Style}
sig Value {f: Attribute -> Style}
sig Value {f: Style -> Attribute}
```

With a binary relation, if you make the wrong choice, you can always access it backwards, writing r.x in place of x.r. With a multirelation, however, navigating via the internal columns can't be done so easily: the dot operator only matches on outer columns. So a good strategy is to start by selecting the column you expect to match on first.

- Choose the first column by considering which column you are most likely to access the relation with.

  Example. For the stylesheet example, most navigations are likely to start with a style. This reduces the choice to

```
sig Style {value: Attribute -> Value}
sig Style {attribute: Value -> Attribute}
```

- Now consider whether other accesses are likely, to determine the last column.

  Example. For the stylesheet example, it seems unlikely that we'll want to consider the relation that maps styles to values for a particular attribute, or that maps styles to attributes for a particular value. So this consideration is not relevant.

  Example. Consider a relation that records which courses students take in which terms. Having decided that the most likely access is by student (for talking about transcripts), we reduce the choices to

  ```
  sig Student {terms: Course -> Term}
  sig Student {courses: Term -> Course}
  ```

  The second may seem more natural. But we're more likely to want to talk about a particular term's registration (mapping students to courses) than about the roster for a particular course over time (mapping terms to students). So the first choice is better, because it lets us write the registration for a term t as terms.t. Note that the course taken by a student s in term t will be written s.terms.t – an idiom that is not as natural as a navigation expression, but is succinct and simple once you're familiar with it.

- If this does not resolve the order, or if there are more than 3 columns to consider, apply the rules for binary relations (see relation-pattern).

  Example. For the stylesheet, the principle of mapping from attributes to values (literally in this case!) suggests the order:

  ```
  sig Style {value: Attribute -> Value}
  ```

  Note that s.value will now be a relation that maps attributes to values for the style s, and that to access it with a standard navigation expression, you'd write something like a.(s.value) (and not a.s.value, which attempts to navigate from a through s, and will produce a type error). Alternatively, you can use the indexing form of the join operator, and write s.value [a], which has exactly the same meaning.

Related Patterns

- : Another way to represent a multiway relationship.
- *relation-pattern*: The basic pattern for two-way relations.
- : Bad use of multirelation.

Questions

How do multirelations compare to UML's qualified associations?

From an Alloy perspective, a qualified association from `X` to `Y` with qualifier set `Q` is a multirelation with columns `X`, `Q` and `Y`, and is accessed like any other multirelation, using the same operators, with the same semantics. UML, in contrast, treats the qualifiers differently, and they are accessed using different operators, in a more restricted way. To navigate from object x along qualified association r with qualifier q, you write x.r[q]. Exactly the same expression will work in Alloy, but it's not a special form for qualified associations, and you can write it in other ways. Multirelations can be manipulated with all the standard relational operators; the qualifier that maps x to y is x.r.y, for example; in UML, this cannot be expressed directly, and requires a comprehension expression.

How do multirelations compare to UML's association classes?

Association classes are another way to represent multiway relationships in UML. From an Alloy perspective, an association from `X` to `Y` with an attached association class `C` is a multirelation (r say) with columns `X`, `C` and `Y`, where `C` is an instance of . So it's just like a qualified association in which the qualifier is a tuple. The association class object for the link from x to y is thus x.r.y, and to access its attribute a, you could write x.r.y.a (or equivalently (x.r.y).a if that looks confusing). In UML, the set of all association class objects corresponding to an object x is written x.r[ry] where ry is the name of the association end at `Y`. Note that the square brackets in this expression and the qualified association access have completely different meanings! The syntax also depends on whether the relation is a self-relation; if not, the set can be written equivalently as x.r. The association object that links x and y cannot be expressed directly in UML without a set comprehension.

Can a multirelation relate 4 entities?

Yes, any number. The important point is that relations have fixed arity, but any arity (except zero and infinity) is allowed. Analysis becomes rapidly intractable as arity increases, but in practice, arities of 4 or more are rare (and are often a mistake – see).

Definition

Enrich a model with redundant components

Motivation

Defining new terms can make a model more succinct and understandable. Because definitions play a different role from other kinds of constraint, it's important to be clear about which constraints are definitions and which aren't. Unlike assumptions, definitions can't be wrong, so they don't need to be checked against knowledge of the problem domain. They can be ill-formed though, so you should write them in a stylized way or apply some simple sanity checks.

Description

A definition has two parts: the declaration of a new term, and a constraint defining it using existing terms. Often it's written as an equality:

```
new_term = some_expression_involving_old_terms
```

Logically, this is a constraint like any other. You can swap left and right hand sides, and its meaning won't change. But methodologically, definitions are special. They don't need any justification. A definition, in itself, doesn't say anything about the system to be built or the assumptions on which its design is based; it's not a claim that can be refuted. Of course, designers may disagree on whether the new notion is a useful one, but unlike the recording of a property of the environment, for example, a definition can't be *wrong*.

There's no special notation in Alloy for definitions, but because definitions play a different role from other kinds of constraint, it's useful to distinguish them informally. It's common, for example, to call the fact that defines a term `T` something like `Define_T`. You might also want to indicate in the name of the field or signature itself that it is a defined term, for example by adding the suffix _def.

Forms of Definition

The most common kind of definition is an equality defining a field in terms of other fields.

Example. A model of a network with a set of hosts and set of links might define the relation connects that associates two hosts when there is a link from one to the other:

```
sig Host {connects: set Host}
sig Link {from, to: Host}
fact Define_connects {connects = ~from.to}
```

Definitions of sets are less common, but no different in principle. The defined set can be a subtype in the primary classification, or a sideset.

Example. A subset StrandedHost might be defined as the set of hosts that have no connections:

```
sig Host {connects: set Host}
sig StrandedHost extends Host {}
fact Define_StrandedHost {StrandedHost = {h: Host | no
h.connects}}
```

A definition need not be an equality, however; any form of constraint can be used. A common idiom involves defining a relation by quantifying over the elements of its tuples.

Example. The definition of connects above can be written in a way that mirrors exactly the informal statement 'two hosts are connected if there is a link from one to the other':

```
fact Define_connects {
  all h, h': Host | h -> h' in connects iff some x: Link | x.from =
h and x.to = h'
  }
```

Implicit definitions like this bring a risk of incomplete or ambiguous definition, so it is usually better to use equality when possible. Often an equality can be obtained by turning a quantification into a comprehension – essentially moving the quantifier into the expression.

Example. The definition of the last example can be rewritten as:

```
fact Define_connects {
  connects = {h, h': Host | some x: Link | x.from = h and x.to =
h'}
  }
```

A definition of a field that uses a quantification over an element of the signature can be packaged as a signature fact. This is just a syntactic shorthand, but because the quantification hidden by the shorthand is over the same signature that the field is declared in, the risk of an incomplete definition is mitigated.

Example. The set of servers associated with a host might be defined as the hosts it is connected to that are classified as servers. With an explicit quantification, you might write:

```
sig Host {connects, servers: set Host}
sig Server extends Host {}
fact Define_servers {
  all h: Host | h.servers = h.connects & Server
  }
```

but this implicit form is error prone. Quantifying over the wrong set – Server instead of Host – like this

```
sig Host {connects, servers: set Host}
sig Server extends Host {}
fact Define_servers {
  all h: Server | h.servers = h.connects & Server
  }
```

now leaves servers undefined for non-server hosts. Writing the same definition as a signature fact, like this

```
sig Host {connects, servers: set Host}{servers = connects &
Server}
sig Server extends Host {}
```

eliminates the risk of making this mistake.

Criteria

A definition is not an arbitrary constraint, even though it can take any form syntactically. It must meet two criteria. For every possible combination of values of the existing terms, the definition must give to the defined term

- at least one value (ie, the definition is *complete*); and
- at most one value (ie, the definition is *unambiguous*).

If the definition is not complete, it will constrain the values of the existing terms, so that possibilities that existed before will no longer exist. When you analyze your model, this might mean that important cases are missed. An ambiguous definition is a contradiction in terms.

Example. Suppose you attempt to define a network host's essential neighbours as the minimal subset of neighbours that can reach every host:

```
2.1 sig Host {connects, essentials: set Host}
2.2 fact Define_essentials {
2.3   all h: Host | let es = h.essentials {
2.4     Host in es.*connects
2.5     es in h.connects
2.6     no h': h.essentials | Host in (es - h').*connects
2.7   }
```

The crucial constraint is the minimization constraint (2.6) which says that you can't take a host away from the set of essential neighbours and still satisfy the condition that every host is reachable from the set.

This definition is both incomplete and ambiguous. It's incomplete because it actually requires every host to be reachable from every other host. And it's ambiguous because there can be two incomparable sets of neighbours that both reach every host, and are both minimal.

The Alloy Analyzer can help you check that a definition is well formed. First, you need to pull the definition out into a predicate. This is good practice in general, although you won't want to do it all the time because it's clumsy for small definitions. Suppose the defined term is `x` and has type `t`, and that the defining constraint is `D`, appearing in a fact:

```
fact Define_x {D}
```

Declare a predicate `define_x` that takes an argument `x` with type `t` and has body `D`, and replace the original occurrence of `D` by an invocation `define_x (x)`:

```
fact Define_x {
  define_x (x)
  }
pred define_x (x: t) {D}
```

Example. The example above becomes:

```
sig Host {connects, essentials: set Host}
fact Define_essentials {
  define_essentials (essentials)
  }
pred define_essentials (essentials: Host -> Host) {
  all h: Host | let es = h.essentials {
    Host in es.*connects
    es in h.connects
    no h': h.essentials | Host in (es - h').*connects
  }
```

This refactoring has no semantic effect, but it allows you to 'turn off' the definition by just commenting it out:

```
fact Define_x {
– define_x (x)
  }
pred define_x (x: t) {D}
```

Now you can experiment with the definition constraint. The two basic checks can be written as assertions:

```
assert Complete_x {
  some x: t | define_x (x)
  }
assert Unambiguous_x {
  no x, x': t | define_x (x) and define_x (x') and x != x'
  }
```

Remember that every assertion is a claim about every instance, so you don't need to say explicitly 'for every combination of values of the existing terms'.

Example. For the network example, the assertions, along with commands to check them, are:

```
assert Complete_essentials {
  some essentials: Host -> Host {
    define_essentials (essentials)
    }
  }
check Complete_essentials
assert Unambiguous_essentials {
  no essentials, essentials′: Host -> Host {
    define_essentials (essentials)
    define_essentials (essentials′)
    essentials not = essentials′
    }
  }
check Unambiguous_essentials
```

and when you run the checks in the default scope, you get the counterexamples shown in Figure 2.1. [To be completed.]

The completeness check is actually higher-order: for each combination of values of the existing terms, it needs to look at each possible value of the defined term. Higher order analyses are supported by the Alloy Analyzer, but are intractable except for very small problems – for example, when the defined term has arity of at most two, and the scope is two.

In practice then, a different approach is often necessary for completeness checks. A strategy that works well to find egregious cases of incompleteness is simply to simulate the model with the definition under some interesting cases.

Example. For the network example, you might write a simulation predicate that asks for some host without connections:

```
  pred show () {
    some h: Host | no h.connect
    }
  run show
```

Executing the run command (with the definition in place, not commented out), will not find an instance, and the Analyzer will report a possible inconsistency – which of course there is, since a host without connections certainly has no subset of neighbours that can reach every other host.

```
sig MemorySystem {
  main: Addr -> one Data,
  cache: Addr -> lone Data,
  stale, dirty: set Addr
  }
  stale = {a: Addr | a.cache not in a.main}
```

Related Patterns

[To be completed.]

Questions

Does an expression used to define a term always have exactly one value?

Yes. In Alloy, every expression has exactly one value for a given binding of its variables. This is by design. In particular, by representing scalars as singleton sets and using relational image in place of function application, Alloy avoids the kinds of undefinedness that arise is most other modelling languages. UML, for example, has a special undefined value that an expression can take, and has a three-valued logic for reasoning about constraints in which such expressions occur. Z is simpler – it has no special values and uses a standard logic – but you can write an expression that will cause the constraint containing it sometimes to have no meaning.

So are definitions by equality always complete and unambiguous?

Unfortunately not. First, cyclic definitions can be ambiguous: just consider adding two new terms, each defined as equal to the other. Cycles are easily detected though (see below). Second, a multiplicity constraint implicit in the declaration of a defined term can combine with the definition to compromise completeness. To avoid this, either declare the term without multiplicities, or first check an assertion saying that the term has the multiplicities you expected.

Alloy's treatment of scalars as singleton sets makes it easy to use equality even for implicit definitions; you can write

```
d = {x: t | F}
```

to say that the `d` is the `x` such that condition `F` holds. In this form, the totality and determinism properties are both easily checked, by asserting that `d` is always a scalar (that is, a set with a cardinality of one).

Are cyclic definitions possible?

Yes, but they are always a mistake. In fact, my claim about definitions by equality being unambiguous only applies if there are no cyclic definitions: just consider adding two new terms, each defined as equal to the other.

To check that your definitions aren't cyclic, you can draw a *definition graph*. Create a node for each defined term, and draw an arc connecting it to a node for each of the terms used in its definition. The definition graph is useful not only for detecting cycles, but also for getting a high level view of the structure of the model's vocabulary. Definitions are often built on top of one another in layers, and the graph shows this clearly.

In a large project, when you have many models and many terms, a definition graph is essential. You can label each arc with the location of the definition itself too. And if you stick to the kinds of lexical convention I recommend, it should not be hard to write a script that constructs such a graph automatically.

Are definitions a good thing?

Generally, yes. Definitions make constraints more succinct, and good definitions can be powerful, changing the way you think about a system. A definition is always preferable to a designation (see designation-pattern), because it can be understood purely formally, with reference to the subtleties of the problem domain. But of course you can overboard, and if you introduce too many new terms, your model will be incomprehensible. A sure sign of poor definitions is inconsistent use of defined terms, with some constraints using a defined term and other similar constraints using the original terms. If it's not completely obvious when to use a defined term, it's probably not a good definition.

How is `let` *related to definitions?*

Alloy's `let` construct gives you a way to make definitions locally to an expression or a constraint. It's always good to localize a definition when possible, because it results in both cleaner models, and better analysis performance.

Isn't this all a bit pedantic?

I don't think so. If you've ever had to wade through a typical requirements document, you'll know how painful it is to make sense of a description in which definitions are poorly organized, and it's not even clear which terms are the defined ones. Of course, a large part of the problem with such documents is the sheer volume of informal text. But even if you replace most of the text with crisp models, it still won't help if the reader can't grasp easily the significance of each term.

Designation

Give meaning to the terms in a model that represent real world phenomena

Motivation

When designing a software system that interacts with the physical world, it's important that the terms used in models that refer to phenomena in the world have clear and unambiguous interpretations. To do this, you write *designations* that connect the formal terms to their (necessarily) informal meaning.

Description

Many software systems are designed either to affect the physical world, or to produce information about it. Some systems have quite direct interaction with physical phenomena, even if mediated by sensors and actuators: an elevator controller, for example, determines the elevator's position from floor sensors, and adjusts it by activating a motor. But for most systems, there is a larger gap between the software and the physical world.

When the gap is small, it's easy to see why it's important to understand how the states and events of the executing software relate to states and events in the real world. This means that any terms used in a model that reflect real world phenomena must have clear and precise interpretations.

> *Example.* The design of an elevator controller involves a relation `atFloor` that relates elevators to floors. The relation is assumed to be updated by sensory input, and is used to determine appropriate actions. But what exactly does `atFloor` mean? It might mean that the elevator is level with the floor, so that the doors can be opened safely. With this interpretation, an elevator will not be associated with any floor at all during its travel between floors. On the other hand, it might mean that the elevator is nearest a particular floor. With this interpretation, an elevator will always be associated with one floor. Both interpretations are plausible, and you could reasonably develop a model with either in mind. What's crucial,

however, is to use the same interpretation consistently. If the updating of atFloor by the sensors uses the first interpretation – level with the floor – and you implement the floor display under the second, your display will be mostly dark, and will just flicker with a floor number when the elevator passes it. Worse, if the updating uses the second interpretation, and the door control uses the first, you'll end up opening the doors between floors.

When the gap is larger, it's easy to forget about the real world, even though the very purpose of the software is to interact with it! So, paradoxically, it becomes even more important to be clear about the interpretation of terms.

Example. In the design of a word processor, you might be tempted to use the term Character without a precise designation. You might end up with an implementation in which the same datatype is used for the 'characters' of a font and the 'characters' typed on the keyboard. Cleanly supporting any feature in which the mapping between the two is not one-to-one – converting character sequences such as ffi to ligatures, selecting alternate versions of alphabetic letters, etc. – will now be impossible. The remedy is to introduce as many distinct terms as you need (character, glyph, symbol, etc.), designate them carefully, and then use them according to the designations.

Writing Designations

Writing designations can be easy, especially in a domain that is well understood. For each set or relation, you give a condition under which a particular atom or tuple would belong to the set or relation.

Example. A model of a system for trading stocks may have a set of companies Company, a set of ticker symbols Ticker, a relation ticker that maps a company to its ticker symbol, and a relation value that maps a ticker symbol to the stock's current value. We might write designations like this:

```
c in Company : c is a public company incorporated in the US
t in Ticker : t is a ticker symbol of the New York Stock Exchange
c -> s in ticker : company c has ticker symbol s
```

```
t -> v in value: current ask price of ticker t in dollars is v
```

But more often, writing designations is tricky. As you try and formulate a clear designation, all kinds of nasty complexities come to mind. Paradoxically, this is a *good* thing. If you postpone the challenge of coming up with a good designation, you might pay a heavy price later, when you realize that your system depends on untenable assumptions, or that different parts of the system make different, incompatible assumptions.

Example. A model of a railway safety system may have sets `Train` and `TrackSegment`, and a relation `on` that maps a train to the track it is currently riding on. But what exactly is a train? What if a train starts out on a journey, and splits in two, with some carriages following one engine, and the rest another? And what does it mean to be on a track segment? When a train is passing from one segment to another, how many segments is it on? One, none or both?

Sometimes even the most innocent-looking names turn out to be remarkably resilient to designation. Often this is because a name that seems to represent a simple physical phenomenon has actually come to be used in a complex human context, so it should be understood instead as a legal or social construct.

Example. In a model of a travel reservation system, what is a *flight*? Experienced travellers know that it certainly doesn't correspond to the most interpretation: a leg of a journey in a plane, involving one takeoff and one landing. A single flight may span multiple legs, and because of code sharing, two passengers sitting next to one another may not even be on the same flight. So it probably makes sense to demote the notion of a flight – to regard it as nothing more than an identifier used in various ways by an airline – and to introduce a range of simpler notions that are less likely to be misunderstood, such as *journey*, *leg*, etc. This example also illustrates how the interpretation of a term is highly dependent on the problem domain; the notion of a *flight* in air-traffic control would be completely different.

Example

An example of a full model with designations.

Related Patterns

- *multiplicity-pattern*: Thinking about multiplicity often helps refine a designation.
- *definition-pattern*: A new term can be introduced by definition, rather than by designation.

Questions

Is writing designations really worth the trouble?

How much effort to invest in writing designations is a matter of judgment. For a lot of exploratory modelling, it's good enough to choose names that are clear and relatively unambiguous. For a safety critical system, pinning down designations is essential. In any serious development, what matters most is recognizing the risk of using names that are not properly designated. I recommend always including a glossary with a model that designates any subtle names, and gives a placeholder for adding and refining designations later.

Is a glossary of designations the same as a data dictionary?

Not exactly. A designation conveys no information about the domain; any assumption about the domain appears as a constraint

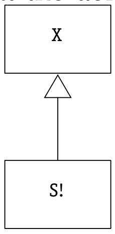


Figure 2.1: Counterexamples showing a bad definition

in a model instead. Data dictionaries often contain implicit constraints, and they mix definitions and designations. The purpose of designations is to separate the model proper, which is completely formal (and meaningless when standing alone), from the interpretation of its terms. Aside from making it easier for readers, this ensures that different models can share the same designations, and that analyses applied to a model take into account all relevant constraints.

Should you designate many or few terms?

The fewer terms that need designating, the better. Every designated term is an opportunity for confusion and error. Terms that are introduced by definition rather than designation are easier to understand. You don't need to worry about what they mean in any deep sense, since you can pass the buck, explaining the defined name using other names. But eventually, the buck stops somewhere, and you have to explain a name that refers directly to some phenomenon. Crafting a minimal collection of designations that gives just enough connection to the domain is hard but rewarding.

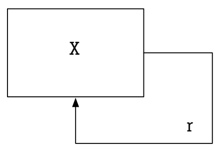
If I understand what a term means, do I really need to write a designation down?

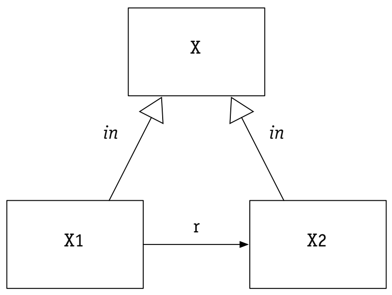
Designations are like comments in code. Novices especially are loath to write them, and when they do, they pick the ones that are easiest to write (and of course the least useful). Experienced developers know never to underestimate the creative ability of readers (of code or any kind of artifact) to invent for themselves new interpretations of terms, inconsistent with everyone else's, while thinking, like the writer, that the term's meaning was completely obvious.

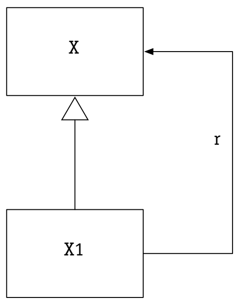
Where can I find more guidance on writing designations?

The idea of making designations explicit is due to Michael Jackson. His book *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices* [] contains several articles on designation. His more recent book *Problem Frames* [] covers the idea in more depth.

2.2 Declaration Patterns


X
r


X
in
in
X1
r
X2


X
r
X1

Singleton

Motivation

In some sets, there are particular elements that play important roles. Such an element is called a *Singleton*, and is modelled as a singleton set – that is, a set containing one element. For example, in the directories of a file system, the root directory is a singleton.

Basic Form

The basic form of Singleton has a set X with a singleton subset S, as shown in Figure 2.2. The singleton is never a top-level set, because if it were, there would be no other elements of the set, and modelling the singleton would have no use (see Section). Singletons often arise from projection. For example, in a model of a file system projected onto a single user, the desktop folder would be a singleton; in the full model, it would be a relation that maps each user to a folder.

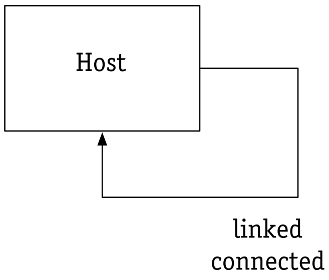
Variants

An *option* is like a singleton, but represents a set that may have one element or no elements at all. Options tend to arise only under projection. For example, a model of book structure projected onto a particular book may include a set of chapters, in which the preface, forward and index are optional chapters.

Example

A traffic light protocol has a starting state start and a relation next that maps a state to its successor. In each state, some colours are showing; there are three singletons corresponding to the standard colours. Two particular protocols, for the US and the UK, are also singletons.

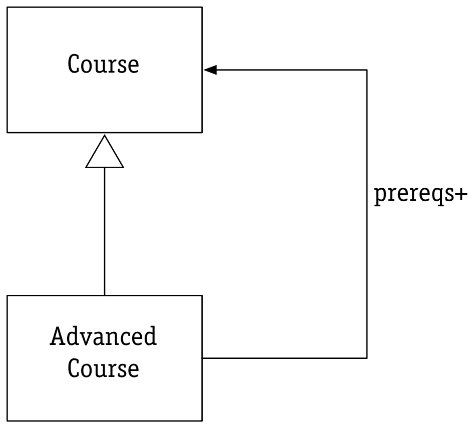
```
sig Colour {}
one sig Red, Yellow, Green extends Colour {}
sig State {on: some Colour}
sig Protocol {start: State, next: State -> one State}
```

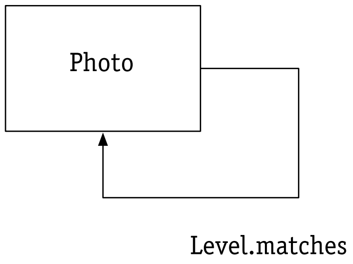


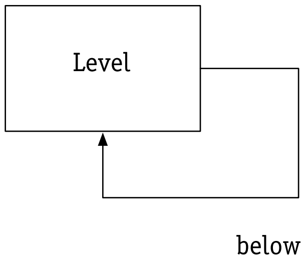
```
    {start.on = Red}
one sig US_Protocol extends Protocol {
  all s: State, s′: s.next |
    s.on = Red and s′.on = Green
    or s.on = Green and s′.on = Yellow
    or s.on = Yellow and s′.on = Red
    }
one sig UK_Protocol extends Protocol {
  all s: State, s′: s.next |
    s.on = Red and s′.on = Red + Yellow
    or s.on = Red + Yellow and s′.on = Green
    or s.on = Green and s′.on = Yellow
    or s.on = Yellow and s′.on = Red
    }
fact CanonicalState {no disj s, s′: State | s.on = s′.on}
```

A diagram for this model is shown in Figure . Projecting onto a particular protocol makes the start state also a singleton (Figure). [To be completed.]

State is an example of the *SetObject* idiom.


Course
prereqs+
Advanced Course





Self Relation

Motivation

A relationship amongst elements in a single set can be modelled as a self relation: a relation from a set to itself. It's sometimes useful to declare a relation this way even if it isn't truly a self relation, simply to avoid declaring new sets for the domain and range.

Basic Form

The characteristic feature of *Self Relation* is a relation whose domain and range may overlap in some instances. The relation, r say, is declared with the same set, X say, as its domain and its range (Figure 2.3).

Variants

It's often convenient to apply *Lowering* (Section), and declare the domain and range explicitly even if they might overlap (Figure 2.4), or to just name one of the two (Figure 2.5).

Constraints

There's a collection of standard mathematical properties for self relations. A relation r is

- *reflexive* if it relates every element to itself:

  ```
  all a: X | a->a in r
  ```

- *transitive* if, whenever it relates a to b and b to c, it also relates a to c:

  ```
  all a, b, c: X | a->b in r and b->c in r = a->c in r
  ```

(or more succinctly r.r **in** r);

- *symmetric* if, whenever it relates a to b, it also relates b to a:

  ```
  all a, b: X | a->b in r => b->a in r
  ```

(or more succinctly ~r **in** r).

If a relation has all three properties then it is called an *equivalence*, and it defines a partition of the set: a division into disjoint subsets, where two elements are in the same subset if they are related.

To a model a relationship that is unordered, use a symmetric self-relation. Alternatively, you can use *SetObject*.

A relation is *connected* if there is a path between any pair of elements:

```
all a, b: X | a->b in *r
```

More often, there is some special element (a *Singleton*; see Section singleton-section) from which all other elements can be reached:

```
X in Root.*r
```

A relation is *acylic* if there no element can be reached from itself:

```
no x: X | x in x.^r
```

A relation is *irreflexive* if it never maps an element to itself:

```
no a: X | a->a in r
```

and *anti-symmetric* if, it never maps a to b and b to a (unless a and b are the same element):

```
all a, b: X | a->b in r and b->a in r => a = b
```

A relation that is reflexive, transitive and anti-symmetric is called a *partial order*. If, in addition, every pair of elements is comparable

```
all a, b: X | a->b in r or b->a in r
```

the relation is a *total order*. If it's just reflexive and transitive, it's a *preorder* or *quasi-order*.

Examples

Network Connectivity

The connectivity of a network can be modelled as a self relation linked on a set Host (Figure 2.6) with the designation:

```
h1 -> h2 in linked : there is a direct, bidirectional link between
h1 and h2
```

Define a second relation connected as the reflexive-transitive closure of linked, so it maps a host to the set of hosts it is connected to, directly or indirectly – assuming that every host can forward packets, and that a host can communicate with itself.

The relation linked is symmetric and antireflexive (since there are no links from hosts to themselves). The relation connected is symmetric, transitive and reflexive, and is therefore an equivalence – it partitions the network into subnets of connected hosts.

Course Prerequisites

The prerequisite structure of courses can be modelled as a self relation `prereq` on a set `Course` (Figure 2.7). Lowering on the domain, we define the set `AdvancedCourse` to consist of those courses that have prequisites.

Designating the relation `prereq` is important to dispel two likely confusions:

> `c1 -> c2` **in** `prereq : course c1 requires c2` **as** `an immediate prerequisite`

Describing the prerequisites as 'immediate' means that we're not including in a course's prerequisites all the prerequisites of its prerequisites, and so on – in other words, distinguishing `prereq` from its transitive closure. The word 'requires' means that these prerequisites are conjunctive; to model a choice of previous courses, a more complex structure is needed (see Section).

The relation `prereq` is irreflexive – you can't be expected to have taken a course before you take it! It must in fact be acyclic, since otherwise there would be nowhere to start.

Photo Matching

An application for organizing photos offers a function that helps locate photos similar to a given photo. The user picks a photo and sets a matching level, and the application displays a set of photos that match it at the given level.

This is readily modelled as a ternary relation `matches`. For a given matching level `l`, the expression `l.matches` is a self relation on `Photo` (Figure 2.8). The matching levels in the set `Level` are ordered by a self relation `below`, which maps a level to all the levels below it.

Which properties `matches` satisfy is a fundamental question in the design of this feature. If `l.matches` defines an equivalence for a given level `l`, it will be possible to organize the photos into groups. The more sophisticated the matching algorithm, however, the less likely it is even to be transitive. Think of those morphed photos you've seen: each gradual change goes from one photo to an almost identical photo, but the photo at the end is nothing like the photo at the beginning. The properties of below are easier: presumably it's a total ordering.

There's a nice instantiation of *Analogy* in this model also. As the matching level increases, we expect more photos to match. Here's a constraint expressing this:

```
all l: Level | l.below.matches in l.matches
```

It says that the pairs of matches at all levels below a given level l are included in the matches for l: in other words, when you increase the matching level, you never lose matches. It can be written more succinctly as:

```
below.matches in matches
```

[To be completed. Make relation properties predicates and use in examples.]

```
pred Reflexive (r: univ -> univ, s: set univ) {s <: iden in r}
pred Irreflexive (r: univ -> univ) {no iden & r}
pred Symmetric (r: univ -> univ) {~r in r}
pred Transitive (r: univ -> univ) {r.r in r}
pred Antisymmetric (r: univ -> univ) {~r & r in iden}
pred PartialOrder (r: univ -> univ, s: set univ) {
  Reflexive' (r, s)
  Transitive (r)
  Antisymmetric (r)
  }
pred TotalOrder (r: univ -> univ, s: set univ) {
  PartialOrder (r, s)
  all x, y: s | x->y in r+~r
  }
```

Tuple

Motivation

There are several circumstances in which it's useful to create an explicit set of objects that represent tuples of other objects. What distinguishes a tuple from a standard object is that it has no identity distinct from its value. For example, a position on a gameboard might be represented as a tuple consisting of a row and a column.

When you want to attach an attribute to the links of a relationship, you can represent the relationship as a set of tuples, and the attribute as a relation mapping each tuple to an attribute value. For example, pairs of slides in a slide show might be represented as a tuple with an attribute for the transition effect to be used on switching from the first slide to the second.

that might be modelled with a tuple

pair of characters and kerning – nice because optional?

keyboard combinations

use to associate property with an association? another way to get a ternary.

Basic Form

The essence of *Tuple* is a set of tuple objects Tuple, with some named components such as x and y (Figure):

```
sig X, Y, A {}
sig Tuple {x: X, y: Y, a: set A}
```

Note the multiplicity on the component relations; each component must be present. There must be some relations on the tuple set itself, such as the attribute a, since otherwise a simple relation between the components would have sufficed.

A tuple has no identity distinct from its value. To express this, you record a *canonicalizing constraint* saying that two distinct tuples cannot have the same components:

```
no disj t, t': Tuple | t.x = t'.x and t.y = t'.y
```

If this axiom doesn't apply, you don't have an instance of *Tuple*. For example, marriages aren't tuples because a couple can be married, divorced and then married again, and the two marriages would have different properties.

The canonicalizing constraint is a nuisance to write, so the Alloy library contains predefined predicates. In this case, you'd just invoke the predicate tuple like this:

```
tuple (x, y)
```

Because tuples are artifacts of the model, they generally have no designation in the problem domain; they represent nothing more than the combinations of their components.

Variants

A tuple set is *complete* if it contains a tuple for each possible combination of components:

```
pred complete () {
  all x: X, y: Y | some t: Tuple | t.x = x and t.y = y
```

If you can avoid this axiom, you should, because it explodes the state space to be analyzed: the scope of Tuple will be the product of the scopes of the components X **and** Y.

Examples

A kerning pair consists of two typographic glyphs, and some kerning amount that indicates how the spacing between those two glyphs should be adjusted:

```
sig Glyph, Kern {}
sig KerningPair {fst, snd: Glyph, kern: Kern}
fact {tuple (fst, snd)}
```

A slide show consists of a set of transitions, each with an optional transition effect:

```
sig Slide, Effect {}
sig Show {first: Slide, transitions: set Transition}
sig Transition {from, to: Slide, effect: lone Effect}
fact {tuple (from, to)}
```

A slide show of photos in which each photo can appear more than once would not be an instance of *Tuple*, since transitions would not be canonical.

The automatic exposure system of a camera associates with each exposure level a collection of acceptable settings, modelled as tuples, each consisting of a shutter speed and an aperture:

```
sig ShutterSpeed, Aperture {}
sig Exposure {settings: set Setting}
sig Setting {speed: ShutterSpeed, aperture: Aperture}
fact {tuple (speed, aperture)}
```

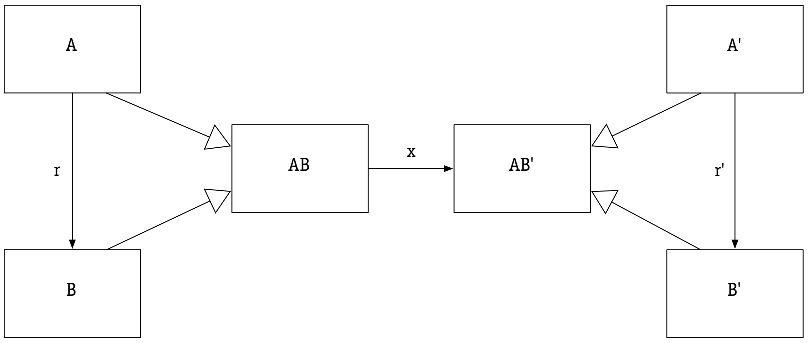
The state of a camera uses two instances of *Tuple*, one for the exposure (a combination of speed an aperture), and another for the overall setting (a combination of exposure and focal length), and identifies some of the settings as requiring a camera shake warning

```
sig ShutterSpeed, Aperture, FocalLength {}
sig LightLevel {settings: set Setting}
sig Exposure {speed: ShutterSpeed, aperture: Aperture}
sig Setting {exp: Exposure, length: FocalLength}
sig ShakeWarning extends Setting {}
fact {tuple (...)}
```

Predicates and Functions

```
–true when the arguments are the component relations of a tuple
pred tuple (fst, snd: univ - univ) {
  fst.~fst & snd.~snd in iden
}
```

$$\begin{tikzcd}
\boxed{A} \arrow[r, "xa"] \arrow[d, "r"'] & \boxed{A'} \arrow[d, "r'"'] \\
\boxed{B} \arrow[r, "xb"] & \boxed{B'}
\end{tikzcd}$$

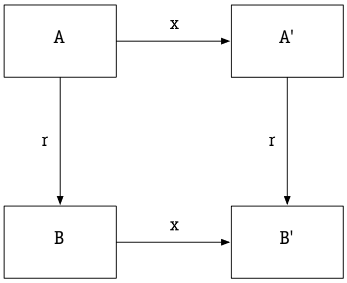


Set Object

Motivation

Higher order relationships, such as relations that associate sets rather than just elements, cannot be expressed directly in a first order language like Alloy. You can get around this most of the time by representing sets explicitly as objects. In a university curriculum, for example, the prerequisites of a course might be satisfied by various combinations of courses; each such combination could be modelled as a set object.

Set objects can also be used to model unordered relationships. For example, a network of computers with bidirectional links between hosts could be modelled with a set object for each link.



$$\begin{tikzcd}
A \arrow[r, "x"] \arrow[d, "r"'] & A' \arrow[d, "r"'] \\
B \arrow[r, "x"] & B'
\end{tikzcd}$$

Basic Form

The basic form consists of some set objects, Set say, and a relation elements that maps set objects to the objects they contain.

Set objects, like tuples (see *Tuple*, Section tuple-section), have no identity distinct from their values:

```
no disj s, s′: Set | s.elements = s′.elements
```

Variants

Examples

A university curriculum has courses each with a choice of prerequisites consisting of one or more courses:

```
module alloy/idioms/courses

sig Course {prereqs: set Prerequisite}
   {this not in prereqs.courses}
sig Prerequisite {courses: some Course}
fact {setObject (Prerequisite, courses)}
```

The signature fact says that a course cannot belong to one of its own prerequisites. A fuller characterization of what makes a prerequisite structure reasonable is quite tricky, and

A computer keyboard assigns events to combinations of keys held down together, each consisting of at exactly one regular key, and zero or more special keys:

```
module alloy/idioms/keyboard

abstract sig Key {}
sig Special, Regular extends Key {}
one sig Control, Shift, Option, Command extends Special {}
sig KeyPress {keys: set Key, event: Event} {one keys & Regular}
sig Event {}
fact {setObject (KeyPress, keys)}
```

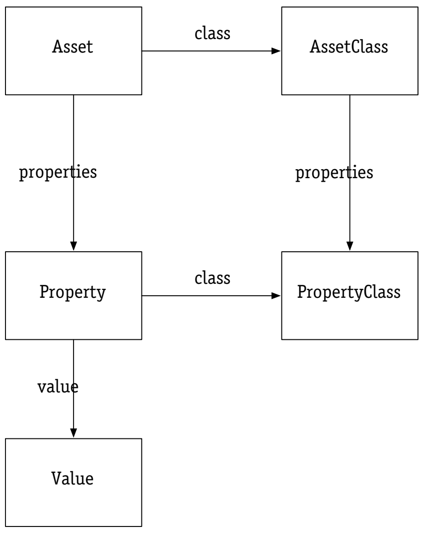
Note the use of *Singleton* to model the special keys.

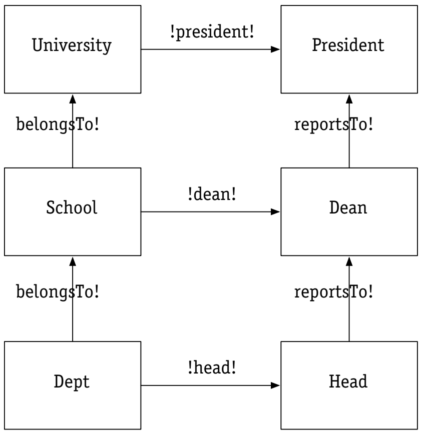
A prison database system tracks pairs of inmates who must not be placed in the same cell (because they belong to opposing gangs, for example):

```
module alloy/idioms/prison

sig Inmate {cell: Cell}
sig Conflict {enemies: set Inmate} {#enemies = 2}
sig Cell {occupants: set Inmate}
   {no c: Conflict | some occupants & c.enemies}

fact {setObject (Conflict, enemies)}
```

Predicates and Functions

```
pred setObject (sets: set univ, elts: univ -> univ) {
  no disj s1, s2: sets | s1.elts = s2.elts
  }
```

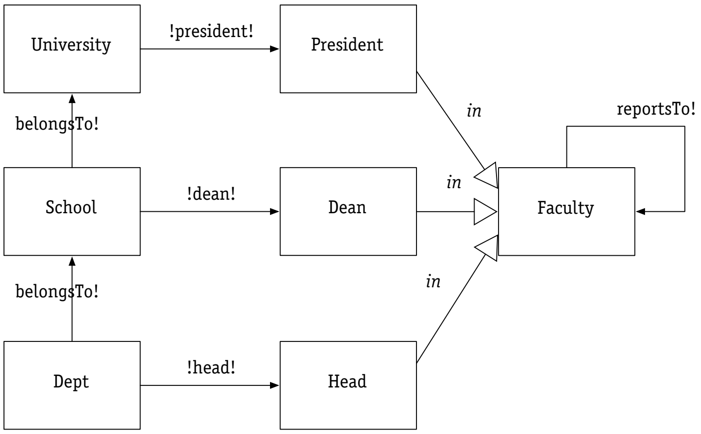

Asset
class
AssetClass
properties
properties
Property
class
PropertyClass
value
Value



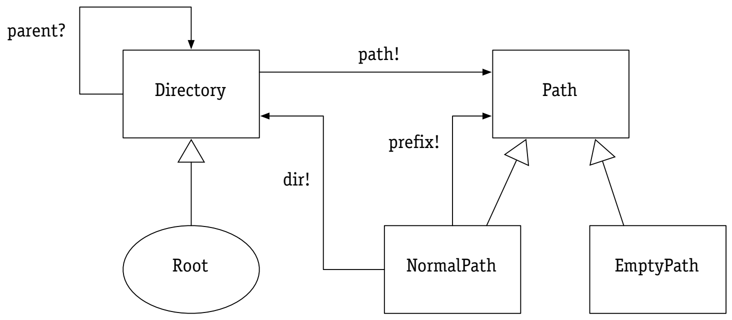
Analogy

Motivation

Sometimes a structure amongst one set of objects mirrors a struc-



ture amongst another. A company's organization into divisions might mirror its reporting hierarchy, for example. *Analogy* makes explicit the parallels between the two structures. The mirroring of the two structures may be a property of a domain, or it may be enforced by the system being designed.



Basic Form

The characteristic feature of Analogy is a partitioning of some sets into two groups. Each set in one group is related to a corresponding set in the other group, and a relation between two sets in one group is mirrored by a relation amongst the corresponding sets in the other group. In Figure 2.9, A and B form a group on the left, and A′ and B′ form a group on the right. A corresponds to A′ and B corresponds to B′. The relation r from A to B is mirrored by the relation r′ from A′ to B′. The relations xa and xb map the elements of A and B to their corresponding elements in A′ and B′.

Variants

A common degenerate form of Analogy (Figure 2.10) has only one set in each group, with the relations r and r′ being self-relations (see Self Relation). Sometimes the two sets collapse together in only one of the groups. It is common for the mapping relations not to be one-to-one: some elements may not be mapped, and two elements may be mapped to the same element. It is rare for them to be many-to-many, however.

It is often convenient to use the same names for the two mapping relations, or for the two analogous relations (Figure 2.11). This is not the same as merging: the two relations with the same names are distinct relations, and Alloy will resolve the overloading according to the context of use.

Constraints

The analogous structures often mirror each other element-by-element. For example, the mapping relations may be 'structure preserving', meaning that two elements are related on the left whenever their corresponding elements are related on the right:

```
all a: A, b: B | a->b in r <=> a.xa->b.xb in r′
```

If the mapping relations are total functions (that is, each left element is mapped to one right element), this situation is called *homomorphism*. If the mapping relations are one-to-one, it's called *isomorphism*, and the structures on the left and right are essentially identical.

A common, weaker condition is that whenever two elements are related on the left, their corresponding elements are related on the right (but not necessarily vice versa):

```
all a: A, b: B | a->b in r => a.xa->b.xb in r′
```

When the mappings are total functions, this can be captured more succinctly by the constraint

```
r.xb in xa.r′
```

which says that if you start at an element in A, follow the relation r and then the mapping xb, you get the same set of elements you would get if you first mapped the element with xa and then followed r′.

Many variants of this kind of constraint arise in practice, such as

```
r.xb = xa.r′
```

and

```
xa.r′ in r.xb
```

Examples

Media Asset Properties

An application for organizing photos and other media files stores properties for each file. The particular properties used depend on the kind of file; photos will have exposures, for example, and movies will have frame rates. The value a property has for a given file must be one of a set of possible values for that property.

To model this (Figure 2.12), we declare a set Asset for the actual assets – the photos, movies, and so on that are stored, and map it to a set AssetClass that has elements such as ‘photo’, ‘movie’, etc. The set Property models the properties associated with assets (using the *Tuple* idiom; see Section), and is mapped to a corresponding set PropertyClass representing the abstract properties – ‘exposure’, ‘frame rate’, etc.

The relation name class is overloaded, and is used for both mappings; likewise, the relation name properties is used on both sides of the analogy. In the fact, the names are resolved by context.

The fact might instead have been written as a signature fact, attached to Asset:

```
sig Asset {...} {properties.@class = class.@properties}
```

which is short for

```
    all a: Asset | a.properties.class = a.class.properties
```

Note the @ symbol needed to preempt the implicit dereferencing of fields of Property and AssetClass, which might otherwise have been interpreted as fields of Asset.

University Hierarchy

Universities are divided into schools, and schools are divided into departments. There is a corresponding managerial hierarchy of presidents, deans and department heads.

The model (Figure 2.13) overloads the name belongsTo for the relations on the left amongst organizational units, and the name reportsTo for the relations on the right amongst personnel. This example is an isomorphism: the two structures mirror each other exactly in every instance.

If a dean can also be a department head – that is, the sets Dean and Head are no longer disjoint – we introduce a superset Faculty, and merge the two reportsTo relations into one (Figure 2.14).

Path Names

In a file system, the path names associated with directories mirror the structure of the directory structure itself (Figure 2.15). Note that the root directory needs special treatment.

Composite

Motivation

Structures that can grow to arbitrary depth, but which take the same form at each level, are modelled with *Composite*. Examples include: directory hierarchies, grouping of objects in a drawing application, organizational hierarchies, and family trees.

Basic Form

Each element of the structure is classified as either a composite object or a leaf (Figure). It is common for there to be properties associated with all elements (such as ep), and properties associated only with leaves (lp); there may also be properties (not shown) associated only with composites. The key to *Composite* is the uniformity of the structuring mechanism across levels, expressed by the relation elements that maps a composite object to the elements it contains.

Variants

Usually, the structure contains no cycles:

```
acyclic (elements)
```

Sharing of substructures may be prohibited; this constraint can be expressed with a multiplicity marking on the diagram, or textually:

```
elements in Composite lone -> Element
```

If there are no cycles or sharing, each element in the structure can be assigned to a level, which can be useful in some applications. Here is one way to do this (as an instance of *Analogy*), using levels ordered with the ordering provided by Alloy's library:

```
open util/ord[Level] as order
sig Level {elts: set Element}
fact {
  order/first().elts = Element - elements.Composite
  all l: Level | next(l).elts = l.elts.elements
```

```
    }
```

The most restricted, but a common, case of this idiom has all the elements in one tree, reachable from a single root:

```
sig Element {ep: EP}
sig Composite extends Element {elements: set Element}
sig Leaf extends Element {lp: LP}
one sig Root extends Composite {}
fact {
   acyclic (elements)
   elements in Composite lone -> Element
   Element in Root.*elements
   }
```

Examples

Test suites for a testing tool:

```
sig Name {}
sig Test {name: Name}
sig Case extends Test {}
sig Suite extends Test {tests: set Test}
sig Run {result: Test -> lone Result}
sig Result {}
sig Success, Failure extends Result {}
```

[To be completed. Explain how model raises questions about how failure reports map to suite structure.]

An address book for an email client, which associates a set of targets with a given name:

```
module addressBook
open alloy/util/relations
abstract sig Target {}
sig Addr extends Target {}
abstract sig Name extends Target {}
sig Alias, Group extends Name {}
sig Book {
  addr: Name -> Target
  } {
  all a: Alias | lone a.addr
2.23 all n: dom (addr) | some n.^addr & Addr
2.24 no n: Name | n in n.^addr
2.25 }
```

Names are either aliases or groups (2.18). An alias is mapped to a single target (2.22), and a group to any number of targets. A target may be an address (2.16) or another name (2.17), so there may be many levels of indirection, with groups containing groups, aliases pointing to aliases, and so on. Cycles are not allowed (2.24), and any name that is mapped must correspond to at least one address (2.23). In this instantiation of *Composite*, Target plays the role of Element, Address plays Leaf, and Name plays Composite. The addr relation plays the role of elements, but in this case is ternary, so that more than one address book can exist over the same sets of targets.

Split Relation

Motivation

A relation between two sets may be split into subrelations on disjoint domains and ranges. For example, a company's computer inventory might classify employees according to their role (sales, design or hacking), and computers by their operating systems. The relation that maps an employee to a computer might have the property that it can be split into separate relations, one from sales personnel to Windows machines, one from hackers to Linux boxes, and one from designers to Macs.

Basic Form

SplitRelation can be expressed in two ways. You can declare a relation on the supertypes, and write constraints for the splitting. Alternatively, you can exploit overloading, and declare the subrelations instead, using the same name for all of them.

```
sig X {r: set Y}
sig X1 extends X {} {r in Y1}
sig X2 extends X {} {r in Y2}
sig Y {}
sig Y1, Y2 extends Y {}
```

```
sig X {}
sig X1 extends X {r: set Y1}
sig X2 extends X {r: set Y2}
sig Y {}
sig Y1, Y2 extends Y {}
```

Variants

Examples

```
sig Employee {computer: Computer}
sig Salesman extends Employee {} {computer in WindowsCom-
puter}
sig Hacker extends Employee {} {computer in LinuxComputer}
sig Designer extends Employee {} {computer in MacintoshComputer}
sig Computer {}
sig LinuxComputer, MacintoshComputer, WindowsComputer extends Computer {}
```

A library's assets consist of books and sound recordings, each of which is assigned an identifier – ISBN's for books, and ISRC's for recordings:

```
sig Asset {}
sig Book extends Asset {id: ISBN}
sig Recording extends Asset {id: ISRC}
sig Identifier {country: Country, publisher: Publisher, title: Title}
sig ISBN extends Identifier {}
sig ISRC extends Identifier {year: Year}
sig Country, Publisher, Title, Year {}
```

2.3 Constraint Patterns

Navigation

A class of expressions commonly used in writing constraints

Motivation

A useful class of expressions can be viewed as navigations through an object model. Novices in particular tend to find navigation expressions easier to read and write than those that involve more ad hoc combinations of relations.

The path expressions used in navigation expressions are just regular expressions over the language of binary relation names.

Description

A navigation expression takes the form

```
navigation-expr ::= set-expr . path-expr
```

where set-expr is an expression denoting a set, and path-expr is an expression made from relational and set operators. The set expression gives a set of starting points, and the path expression gives a set of paths to be followed, obtaining a set of ending points (which may be empty).

A set expression appearing in a navigation expression is either just a set-valued variable name, or another navigation expression, or a combination of set expressions using the set operators:

```
set-expr ::= set-name | navigation-expr | set-expr set-op set-expr
set-op ::= + | & | -
```

A path expression is either a relation name, or a constant, or an operator applied to one or two path expressions:

```
path-expr ::=
   binary-relation-name
   | iden | univ | none
   | unary-op path-expr
   | path-expr binary-op path-expr
binary-op ::= + | & | - | .
unary-op ::= ~ | * | ^
```

The meaning of a navigation expression doesn't need any new explanation; you can just apply the rules given in Chapter . But it's helpful to think of navigation expressions in terms of following paths through a graph:

- to follow a path `p`, where `p` is the name of a binary relation, just follow any of its arcs;
- to follow a path in `p1.p2`, follow a path in `p1`, **then** a path in `p2`;
- to follow a path in `p1 + p2`, follow a path in `p1` **or** a path in `p2`;
- to follow a path in `p1 & p2`, follow a path in `p1` **and** a path in `p2`;
- to follow a path in `p1 - p2`, follow a path in `p1` that is **not** a path in `p2`;
- to follow a path in `~p`, follow a path in `p` **backwards**;
- to follow a path in `*p`, follow a path in `p` **zero or more times**;
- to follow a path in `^p`, follow a path in `p` **one or more times**;
- to follow a path in `iden`, **stay** where you are;
- to follow a path in `univ`, go **anywhere**;
- there are **no paths** in `none`.

[To be completed.]

Discussion

Is `r.s` *a navigation expression, for relation* `r` *and set* `s`*?*

No. As a navigation expression, it would have to be written `s.~r`.

Isn't the grammer ambiguous?

Yes, corresponding to two legitimate (and semantically equivalent) ways to view certain expressions.

[To be completed.]

Similarity

Define equality of structure as a predicate

Motivation

A structure has identity and content. Two kinds of equality can therefore be defined. Often useful to have a predicate that says when two structures are structurally equivalent. Like equality in OOP.

Basic Form

Variants

Examples

Related Patterns

Canonicalization is the other option.

Discussion

2.4 Analysis Patterns

Guided Simulation

Simulate a model to check consistency and explore typical cases

Motivation

[To be completed.]

3: Dynamic Models

This chapter will be about *dynamic models*. A dynamic model describes a system's behaviour.

It will include, for example:

- *Incremental State*. The state of the system is factored out into one or more signatures, and built-up incrementally (using signature extension). This corresponds to the 'established strategy' of Z.
- *Local State*. Fields of a signature are extended with a column representing time. Each atom of a signature can then be viewed as an object with time-varying behaviour. This scheme works nicely for loosely coupled, asynchronous systems.
- *Heap*. The state of the system is represented by a heap that maps object references to object values. This corresponds most directly to the view of state found in languages such as Java, and is useful for modelling features such as views, in which an explicit representation of object sharing is desirable.
- *Events*. The events that occur during a system's execution are represented explicitly as objects in their own right, and associated with operations explicitly. The merit of this pattern is that common features of events can be described without repetition. It also admits frame conditions in Reiter's style. This pattern is orthogonal to the others.

Abstract Machine

Model a system as operations acting on a state

Motivation

Many dynamic systems can be modelled in a very simple way, by treating actions performed on the system and by the system as *operations,* each specified separately in terms of its effect on a global *state.* All control is embedded in the operations themselves, without the need for any kind of additional control program, making the description highly modular and easy to reason about.

Basic Form

Here is an outline showing the general form of the *abstract machine* pattern:

```
sig State {...}
pred invariant (s: State) {...}
pred init (s: State) {...}
pred op1 (s, s': State) {...}
...
pred opN (s, s': State) {...}
```

The state is declared as a signature, with fields representing the state components. The rest of the model consists of a series of predicates: the *invariant,* a constraint that characterizes a well-formed state, which is expected to hold for every reachable state of the machine; the *initialization* that gives the conditions that hold in the initial state; and then a collection of *operations,* each of which describes the effect of a particular action as a constraint relating the state before and the state after.

In a particular execution of an operation, the state before is called the *pre-state,* and the state after is called the *post-state.* It's common to label the post-state with a prime mark, but there are other conventions too. There's no significance to the names of the pre- and post-states; an operation could equally well be declared like this:

```
pred op1 (pre, post: State) {...}
```

The initialization predicate can hold for more than one state: that is, the machine can be started in any one of a set of states. The operations can be *non deterministic*, meaning that when invoked in a given pre-state, one of several different post-states might result. They can also be *partial*, meaning that there are pre-states in a which an operation cannot be applied, because there is no post-state for which the pre/post combination satisfies the operation's predicate.

Mathematically, an instance of this pattern describes a *state machine*, consisting of one or more initial states, and a transition relation that maps states to states. You can imagine the state machine as a graph, with the states drawn as nodes, the initial states highlighted, and the transition relation drawn as arrows between nodes. For any machine you're likely to describe, the graph is too big to draw, but it's still a useful picture to have in mind. Think of the initialization predicate as a test on state nodes that determines whether or not they should be highlighted, and each operation as some subset of the arrows, connecting pairs of nodes that satisfy the operation constraint.

Although the operations have names, you should think of them as anonymous. The names are just useful ways to refer to the operations in the model, but they don't appear in any instance resulting from an analysis. In terms of the graph, the transition arrows are *unlabelled*.

The operations often have additional arguments, corresponding to inputs and outputs. An operation can have just an input, just an output, or both an input and an output. In general, it can have any number of inputs and outputs (but the same number for every invocation).

Arguments are often also used to expose values that are selected internally by the operation. For example, a flushing operation in a cache may reveal as an argument the set of cache lines that are flushed, even though these are not an input but are in fact chosen non-deterministically by the operation itself. When the operation is used (in an invocation), such arguments are usually existentially quantified.

Variants

Examples

A model of a web browser cache:

```
module browser

sig URL, Page {}
sig State {
   web, cache: URL -> lone Page
   }

pred init (s: State) {
   no s.cache
   }

pred get (s, s': State, u: URL, p: lone Page) {
   some s.cache[u] =>
      p = s.cache[u] and s.cache = s'.cache,
      p = s.web[u] and s'.cache = s.cache ++ u->p
   }

pred flush (s, s': State) {
   s'.cache in s.cache
   }
```

The state consists of the state of the browser cache itself, and the state of the web; each is a simple mapping from URL's to web pages. The cache is initially empty. The get operation takes a URL and returns a page, which is either the page in cache (if present), or the page obtained from the web (otherwise). If the page is not in cache, the cache is updated. The flush operation has no arguments; its effect is simply that the new cache holds some entries from the old cache. Note that neither operation constrains the state of the web, which is assumed to be capable of changing at any time. There is no invariant.

This is a typical way to model flushing in caches; it's assumed that the flush operation is called at arbitrary times. We could have specified flush with an argument to expose the set of URL's that are flushed:

```
pred flush (s, s': State, urls: set URL) {
    s'.cache = (univ - urls) <: s.cache
    }
```

A model of an address book program:

```
module book

sig Addr, Name {}
sig Book {
    addr: Name -> (Name + Addr)
    }

pred inv (b: Book) {
    let addr = b.addr |
        all n: Name {
            n not in n.^addr
            some addr.n = some n.^addr & Addr
            }
    }

pred add (b, b': Book, n: Name, t: Name + Addr) {
    b'.addr = b.addr + n->t
    }
pred del (b, b': Book, n: Name, t: Name + Addr) {
    b'.addr = b.addr - n->t
    }
fun lookup (b: Book, n: Name): set Addr {
    n.^(b.addr) & Addr
    }
```

Say it actually violates the invariant. Includes *observer operation.*

Related Patterns

Local state, important variant of abstract machine.

Split operation, etc, often used with it.

Discussion

invariants in decls?

How can a named operation be anonymous?

Names of Alloy predicates are never semantically significant. The name of the predicate is just a shorthand for the predicate's constraint, and isn't part of the predicate itself. If you renamed a predicate and all its uses, you would not have changed the meaning of the model. The anonymity of predicates can be seen in their combination too. If you define a predicate `P` as the conjunction of two predicates `Q` and `R`, there would be nothing in `P` that would indicate its origins in `Q` and `R`. It could equally well have come from two other predicates with different names but the same constraints.

Are anonymous operations a feature of Alloy?

No, anonymity of operations is the norm in most modelling languages: Z, VDM, Larch and OCL, for example.

Aren't operation names used by clients to determine which operations to call?

Yes! Something fishy going on here.

Is the priming of the post-state like the priming of variables in Z?

Yes and no. Alloy adopts the same convention of priming post-state variables, but the notion of priming is not built in. The prime mark is just part of the variable's name, and the names of a predicate's arguments are not significant. In Z, however, priming of operators is a built-in notion, and adherence to the priming convention is required for certain operators (such as the precondition operator and sequential composition) to work correctly.

Why aren't input and output arguments distinguished?

There's no need.

Is this pattern only good for API's?

No, can represent all kinds of autonomous systems too.

Explicit Precondition

Split an operation specification into an explicit precondition and postcondition

Motivation

Some operations are *partial*, and cannot be applied in all states. If so, it can be useful to separate the *precondition*, which describes the set of applicable states, from the *postcondition*, which describes the effect of the operation in those states.

Basic Form

The basic form of the pattern is to introduce two additional predicates for each operation, corresponding to the pre- and post-conditions:

```
sig State {...}
pred op (s, s': State) {
   pre_op (s)
   post_op (s, s')
   }
pre_op (s: State) {...}
post_op (s, s': State) {...}
```

For this separation to make sense, there must be no hidden constraint `C` on the pre-state lurking in the postcondition for which

```
post_op (s, s') ⇒ C (s)
```

holds. If such a tautology were to hold, `C` would be an additional precondition, and it would not be true that `pre_op` alone determines whether the operation can happen.

Variants

A common variant has the pre- and postconditions separated, but doesn't introduce extra predicates. Instead, the operation is simply split into two sets of constraints, labelled by appropriate comments. As when predicates are used, it's important that the postcondition not hide an implicit precondition; an operation specification satisfying this is said to be 'honest' in Z.

Examples

Address Book

An address book model, in which the add operation has a precondition intended to ensure preservation of the invariant:

```
module book

sig Addr, Name {}
sig Book {
    addr: Name -> (Name + Addr)
    }

pred inv (b: Book) {
    let addr = b.addr |
        all n: Name {
            n not in n.^addr
            some addr.n => some n.^addr & Addr
            }
    }

pred add (b, b': Book, n: Name, t: Name + Addr) {
    pre_add (b, n, t)
    post_add (b, b', n, t)
    }
pred pre_add (b: Book, n: Name, t: Name + Addr) {
    n not in n.*(b.addr)
    t in Addr or some lookup (b, t)
    }
pred post_add (b, b': Book, n: Name, t: Name + Addr) {
    b'.addr = b.addr + n->t
    }
```

As an example of a hidden precondition, suppose the state signature had instead declared the address map so that it associates at most one name or address with each name:

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

Now the add operation has a precondition induced by the postcondition, requiring that the mapping not already map the name being added to some other target:

```
pred post_add (b, b': Book, n: Name, t: Name + Addr) {
    b'.addr = b.addr + n->t
    }
pred pre_add_hidden (b: Book, n: Name, t: Name + Addr) {
    no b.addr & n->(univ - t)
    }
assert HiddenPre {
    all b, b': Book, n: Name, t: Name + Addr |
        post_add (b, b', n, t) => pre_add_hidden (b, n, t)
    }
```

The assertion claims the hidden precondition; uses such as this make it convenient to package pre- and postconditions in separate predicates.

To eliminate the hidden precondition, we could change the postcondition to override any old entries for the name:

```
pred post_add (b, b': Book, n: Name, t: Name + Addr) {
    b'.addr = b.addr ++ n-t
    }
```

Discussion

Is this like VDM etc? B?

Yes, but VDM doesn't conjoin. But this does give us access to each condition separately.

Operation Simulation

Simulate an operation's executions

Motivation

Having written an operation, it's a good idea to sample some executions to check that the operation has the intended behaviour. *Simulation* involves simply running the operation predicate, usually with some additional constraints chosen to force selection of more interesting cases.

Basic Form

Given an operation of the form

```
sig State {...}
pred op (s, s′: State) {...}
```

a simulation can involve nothing more than running the operation predicate:

```
run op for 3 but 2 State
```

(Note the scope setting: there is usually no point in considering more than two states, which will correspond to the pre- and post-states of the operation.)

A series of sample executions can be obtained in the Alloy Analyzer using the 'next solution' function. Often, however, it's more convenient to guide the analyzer towards more interesting executions. In that case, you'll want to wrap the operation predicate in a simulation predicate:

```
show_op (s, s′: State) {
   op (s, s′)
   ...
   }
run show_op for 3 but 2 State
```

Additional constraints are then inserted into the simulation predicate. The typical forms of constraints that are added are:

- Constraints on the pre-state, forcing execution from interesting states; you can think of this is simulating *forwards*;
- Constraints on the post-state, forcing execution that result in interesting states; you can think of this is simulating *backwards*;
- Constraints on both pre- and post-states, forcing certain effects to occur.

A common form of pre/post constraint asserts that the pre- and post-states differ in some interesting way. The constraint

```
s not = s'
```

is usually not enough; it may even give an execution in which the pre- and post-states are structurally identical. Instead, you'll usually want to assert a difference between particular components of the states:

```
s.c not = s'.c
```

Variants

Examples

Browser Cache

A model of the get operation in a browser cache:

```
module browser

sig URL, Page {}
sig State {
   web, cache: URL -> lone Page
   }

pred get (s, s': State, u: URL, p: lone Page) {
   some s.cache[u] =>
      p = s.cache[u] and s.cache = s'.cache,
      p = s.web[u] and s'.cache = s.cache ++ u->p
   }
```

The state consists of the state of the browser cache itself, and the state of the web; each is a simple mapping from URL's to web pages. The get operation takes a URL and returns a page, which is either the page in cache (if present), or the page obtained from the web (otherwise). If the page is not in cache, the cache is updated.

Running the operation without additional constraints, like this

```
pred show_get (s, s': State, u: URL, p: lone Page) {
   get (s, s', u, p)
   }
run show_get for 3 but 2 State
```

gives an execution in which neither the web nor the cache maps any URL's, so no page is returned.

So we try a simulation forwards, constraining the cache and the web to be non-empty in the pre-state:

```
pred show_get (s, s': State, u: URL, p: lone Page) {
   some s.web and some s.cache
   get (s, s', u, p)
   }
run show_get for 3 but 2 State
```

This now gives a simple execution in which the URL requested is in the cache, and no state change occurs. Curious to see an execution in which the URL is not in the cache, we try

```
pred show_get (s, s': State, u: URL, p: lone Page) {
   some s.web and some s.cache
   no s.cache[u]
   get (s, s', u, p)
   }
run show_get for 3 but 2 State
```

which gives an execution in which the URL is not mapped to a page by the web either! So we strengthen the pre-constraint to require that the URL be mapped by the web:

```
pred show_get (s, s': State, u: URL, p: lone Page) {
   some s.web and some s.cache
   no s.cache[u] and some s.web[u]
   get (s, s', u, p)
   }
run show_get for 3 but 2 State
```

Now, as expected, we get an execution in which the cache is updated. Interestingly, in that very transition, the mapping of the requested URL is dropped in the web itself (which is quite plausible), alerting us to the fact that changes to the web are unconstrained in our operation.

Trying a simulation backwards, we ask for an execution in which no page is returned, but there is a mapping for the requested URL in the web:

```
pred show_get (s, s': State, u: URL, p: lone Page) {
    some s'.web[u] and no p
    get (s, s', u, p)
    }
run show_get for 3 but 2 State
```

The execution generated is quite plausible: the mapping appeared as the request was being handled, so it wasn't available. In constrast, an attempt to simulate backwards for an execution in which no page is returned, but there is a mapping for the requested URL in the cache

```
pred show_get (s, s': State, u: URL, p: lone Page) {
    some s'.cache[u] and no p
    get (s, s', u, p)
    }
run show_get for 3 but 2 State
```

yields no instance, and is inconsistent.

Finally, a simulation using a constraint on both pre- and post-states, asking for an execution in which the cache grows:

```
pred show_get (s, s': State, u: URL, p: lone Page) {
    some s'.cache - s.cache
    get (s, s', u, p)
    }
run show_get for 3 but 2 State
```

In contrast, a request for an execution in which the cache shrinks

```
pred show_get (s, s': State, u: URL, p: lone Page) {
    some s.cache - s'.cache
    get (s, s', u, p)
    }
```

```
run show_get for 3 but 2 State
```

fails, returning no instance, since the get operation doesn't allow for any flushing of the cache.

Discussion

Why a simulation predicate? Why not just add the extra constraints to the operation itself?

You can indeed add the constraints to the operation, but it's not good practice because you may forget to take them away. Declaring the simulation predicate gives you a convenient place to store the constraints that aren't part of the operation proper, but are there solely to obtain interesting executions.

Invariant Preservation

Check that an operation preserves an invariant

Motivation

To check that every reachable state in a dynamic system satisfies an invariant, you can check that the invariant holds for all initial states, and that it's preserved by every operation. If so, the invariant holds by induction in all reachable states. This method is easy to apply in practice, and, because it treats each operation separately, makes a robust case that unsafe states are never visited.

Basic Form

Given an abstract machine, with a declaration of the state, an invariant, an initialization and operations:

```
sig State {...}
pred inv (s: State) {...}
pred init (s: State) {...}
pred op (s, s': State) {...}
```

the basic form of an invariant preservation check is a collection of assertions, one that the initialization predicate implies the invariant, and one for each operation saying that it preserves the invariant:

```
assert initEstablishes {
   all s: State | init (s) implies inv (s)
   }
check initEstablished
assert opPreserves {
   all s, s': State |
      inv(s) and op(s, s') implies inv (s')
   }
check opPreserves
```

A counterexample for the former is a initial state that violates the invariant. A counterexample for the latter is a transition from a state s satisfying the invariant to a state s′ that does not satisfy the invariant, but which is a possible outcome of the operation: in other words, a transition that breaks the invariant.

It's easy to see that, by induction, if the assertions are valid, the invariant must hold in every reachable state. The initialization assertion ensures that the invariant holds in every initial state. The preservation assertions ensure that, if the invariant holds after some execution of *k* steps, then it must also hold after *k*+1 steps. Since every state is reached by an execution of a finite number of operations, the invariant must hold in every reachable state.

Variants

??

Examples

Address Book

A model of an address book program, in which names can be mapped to both names and addresses, in multiple levels:

```
module book

sig Addr, Name {}
sig Book {
    addr: Name -> (Name + Addr)
    }

pred inv (b: Book) {
    let addr = b.addr |
        all n: Name {
            n not in n.^addr
            some addr.n = some n.^addr & Addr
            }
    }

pred add (b, b': Book, n: Name, t: Name + Addr) {
    b'.addr = b.addr + n->t
    }
pred del (b, b': Book, n: Name, t: Name + Addr) {
    b'.addr = b.addr - n->t
    }
```

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

The invariant states that no name is mapped to itself, directly or indirectly, and that if a name is itself mapped to, then the name is mapped, directly or indirectly, to at least one address.

A check that add preserves the invariant

```
assert AddPreserves {
   all b,b′: Book, n: Name, t: Name + Addr |
      add (b,b′,n,t) and inv (b) implies inv (b′)
   }
check AddPreserves for 3
```

fails, generating, for example, a scenario in which a name is mapped directly to itself. To make add preserve the invariant, we can constrain it with a precondition saying that the name being added is not directly or indirectly mapped to itself, and that the target of the name being added is either an address or a name that resolves to at least one address:

```
pred add (b, b′: Book, n: Name, t: Name + Addr) {
   -- precondition
   n not in n.*(b.addr) and (t in Addr or some lookup (b, t))
   -- postcondition
   b′.addr = b.addr + n-t
   }
```

Discussion

Does invariant checking always work?

No. Sometimes an invariant holds in every state, but it can't be checked by this method. The problem is that the preservation check allows the operation to be invoked in any pre-state satisfying the invariant. It may well be that the operation is never in fact invoked in those states from which it would behave badly and break the invariant. If so, checking invariant preservation will seem to give false alarms. In theory, you can workaround this problem by strengthening the invariant with constraints that characterize the reachability. In practice, however, it's hard to characterize the reachable states succinctly.

So why bother with invariant checking?

First, it's a simple and effective technique that often does work. Second, because it's *modular*, it can be applied to very partial abstract machine descriptions: you can do an invariant preservation check when you only have one operation. Third, the weakness of invariant checking – its inability to account for reachability – turns out to be a strength. Because each operation is checked separately, and a change to one operation cannot undermine another operation (so long as the change does not cause the invariant to be violated), the method is inherently robust. In contrast, a check based on reachability, in which reachability is determined automatically, is fragile, and a small change to one operation can dramatically change the set of reachable states, and thus result in invariant violations many steps later.

Relation to rep invariants?

Declarative Operation

Specify an operation with free-form constraints

Motivation

Roughly speaking, there are two ways to describe an operation. In the *operational* idiom the operation is itself viewed as a collection of smaller operations, with a constraint setting each field of the post-state. In the *declarative* idiom, the operation is instead characterized by a collection of free-form constraints that relate the values of fields in the pre-state and post-state. An operational modeller asks 'how would I make *X* happen?'; a declarative modeller asks 'how would I recognize that *X* has happened?'.

The declarative idiom is more expressive, because it can describe *non-deterministic behaviour*, in which a single pre-state can result in any of several different post-states. Sometimes it is also the most direct way to describe the behaviour.

Non-determinism is a kind of intentional incompleteness, in which details of behaviour are left unspecified. It has two different uses:

- *Describing unpredictable components.* Sometimes the behaviour cannot be predicted, so it makes no sense to attempt to describe it. The behaviour of the environment of a system, in particular, is usually unknown, and, even if we could predict it to some degree, we'd prefer to make no assumptions that might turn out to be wrong, and undermine the system's correctness.
- *Deferring implementation decisions.* It is often useful to omit implementation details, deferring them until a later stage of development. This lets you explore fundamental properties of behaviour first, before considering implementation strategies, and it also improves modularity, since one part of the systen will not come to depend on another part being implemented in a particular way. The lack of information that results from omitting implementation details is sometimes called 'under-determinedness' to distinguish it from true 'non-determinism' in which the post-state resulting from an operation at runtime is not determined by the history of prior states.

Basic Form

Declarative descriptions in Alloy are characterized more by absence of form than presence. In the operational idiom, an operation is usually given as a collection of constraints, one for each field of the post-state:

```
sig State {f1: T1, ..., fn: Tn}
pred op (s, s': State) {
    s'.f1 = e1
    ...
    s'.fn = en
    }
```

Although the constraints aren't assignment statements – their order doesn't matter and they have no side-effects – they are reminiscent of imperative programming. Each component f_i of the post-state is assigned a value given as an expression e_i in terms of some combination of values of the components of the pre-state.

A declarative operation may violate this form in any number of ways:

- Some fields may have no constraint at all that mentions their value in the post-state;
- Expressions about the post-state may appear on the right hand side of comparisons, and expressions about the pre-state may appear on the left;
- Subset comparisons may be used in place of equality;
- The constraints on fields may be inside quantifiers.

A common form uses constraints on fields that are defined in terms of other fields; the base fields themselves are not constrained.

Examples

Web Cache

A model of a web browser cache has an operation flush which simply eliminates some URL/webpage mappings from the cache:

```
module browser
```

```
sig URL, Page {}
sig State {
  cache: URL -> lone Page
  }

pred flush (s, s': State) {
  s'.cache in s.cache
  }
```

This is an example of non-determinism introduced for implementation freedom. A particular implementation will of course have some strategy for determining which pages to keep – based for example on their age and frequency of access – and the behaviour at runtime will be quite predictable.

Non-deterministic operations can often be written in a slightly more operational form, using existential quantification to capture the non-determinism:

```
pred flush (s, s': State) {
  some drop: set URL |  s'.cache = (univ-drop) <: (s.cache)
  }
```

A common trick is to make the quantified variable an explicit argument of the operation

```
pred flush (s, s': State, drop: set URL) {
  s'.cache = (univ-drop) <: (s.cache)
  }

The operation is now no longer non-deterministic, but its uses
will reintroduce the non-determinism:

...
some drop: set URL | flush (s, s', drop)
...
```

Railway Train Motion

A railway system is described as a network consisting of track segments, each with a set of successors:

```
module railway
sig Segment {
  next: set Segment
```

```
    }
```

and a dynamic state in which each train is on a particular segment:

```
sig State {
    on: Train -> one Segment
    }
sig Train {}
```

The behaviour of trains can be described by a non-deterministic operation that says that, in a given step, some set of trains move from their segments to arbitrarily chosen successors, and the rest remain on the same segments:

```
pred trainsMove (s, s’: State) {   some movers: set Train |
all t: Train |                     t in movers =>
s’.on[t] in s.on[t].next ,                      s’.on[t] = s.on[t]
}
```

This example illustrate the use of non-determinism to describe environmental behaviour. A design of a railway switching system will have to make some assumptions about how trains move (for example, that they don’t jump over entire track segments), but clearly, the fewer assumptions made, the better.

Address Book Addition and Deletion

A multilevel address book, constrained to forbid cycles:

```
module book

sig Addr, Name {}
sig Book {
    addr: Name -> (Name + Addr)
    } {
    no ^addr & iden
    }
```

Suppose we want to specify a deletion operation that removes a name, both as something that maps to other names and addresses, and as a target for a name. We probably don’t want to just remove all mappings from and to the name, like this:

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

but instead want to make sure that any name that mapped to an address via this name will still map to it. This can be expressed directly as a declarative operation:

```
pred del (b, b': Book, n: Name) {
    let ad = ^(b.addr), ad' = ^(b'.addr) |
        all x: Name |
            x = n =>
                no x.ad' + ad'.x ,
                x.ad' - n = x.ad - n
    }
```

This rather formidable looking constraint is not as complex as it looks. It introduces local names ad and ad' for the indirect mappings before and after, and then says that, for any name x, if x is the name n being deleted, then in the mapping after, it maps to nothing and is not mapped to; and if x is not n, it maps to the same set of names and addresses before and after, ignoring n.

Simulating this version of the operation (or running a *Determinism Check*) shows some rather strange behaviours. Our description allows the book to be rearranged so long as the effective mapping is the same. So, for example, if the entry for name N0 maps it to another name N1, and to an address A0, and the entry for name N1 also includes A0, our operation may remove the superfluous entry, leaving N0 pointing only to N1, and N1 to A0. This is not desirable behaviour; the user may well have intended the redundancy in order to ensure that N0 continues to map to A0 whatever changes are made to the mapping for N1. Worse, our operation can add superflous entries!

This declarative operation is thus not suitable in its current form: it expresses a necessary condition on the behaviour, but not a sufficient one. We could refine it to eliminate the unacceptable effects, or, alternatively, we can reformulate it operationally. Taking the latter approach, we can equate the new address mapping to the old one, with the mappings to and from the deleted name omitted, and with a new mapping from each name that mapped to the deleted name, to each name or address it mapped to:

```
pred del1 (b, b′: Book, n: Name) {
    let others = univ - n, ad = b.addr {
        b′.addr = others <: (b.addr) :> others + ad.n->n.ad
        }
    }
```

We can then perform a *Refinement Check* to make sure that this operation does indeed satisfy the necessary conditions of the declarative operation:

```
assert DelOK {
    all b, b′: Book, n: Name |
        del1 (b, b′, n) => del (b, b′, n)
    }
check DelOK for 2
```

Related Patterns

A declarative operation may constrain a field introduced by *Definition.*

Discussion

Determinism Check

Check that an operation is deterministic

Motivation

An operation may be written with the intent that it's deterministic, but it may in fact be non-deterministic because of a simple mistake, or a deeper misunderstanding about the system being modelled. The determinism check is easy to apply and readily exposes such problems.

Basic Form

An operation defined as a predicate on states

```
sig State {...}
pred op (s, s': State) {...}
```

is *deterministic* if it associates at most one post-state with each pre-state. This can be checked with an assertion:

```
assert op_deterministic {
   all s, s', s": State |
      op (s, s') and op (s, s") => s' = s"
   }
```

Most operations will fail to meet this strict definition of determinism because states themselves are atoms, so checking the assertion may give a counterexample in which the distinct post-states are indeed distinct state atoms, but are structurally identical. So often the notion of determinism is weakened, to insist only that the operation determines the structure of the post-state. If the state signature has fields f_1 through f_n

```
sig State {f_1: T_1, ..., f_n: T_n}
```

the determinism check becomes

```
assert op_deterministic {
   all s, s', s": State |
      op (s, s') and op (s, s") =>
         {
         s'.f_1 = s".f_1
```

```
            ...
            s'.f_n = s''.f_n
            }
    }
```

Variants

Examples

A multilevel address book:

```
module book

sig Addr, Name {}
sig Book {
    addr: Name -> (Name + Addr)
    } {
    no ^addr & iden
    }
```

Suppose we want to specify a deletion operation that removes a name, both as something that maps to other names and addresses, and as a target for a name. We probably don't want to just remove all mappings from and to the name, like this:

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

but instead want to make sure that any name that mapped to an address via this name will still map to it. This can be expressed directly as a declarative operation:

```
pred del (b, b': Book, n: Name) {
    let ad = ^(b.addr), ad' = ^(b'.addr) |
        all x: Name |
            x = n =>
                no x.ad' + ad'.x ,
                x.ad' - n = x.ad - n
    }
```

To check that this is deterministic, we write

```
assert del_deterministic {
    all b, b′, b‶: Book, n: Name |
        del (b, b′, n) and del (b, b″, n) =
            b′.addr = b‶.addr
    }
check del_deterministic for 2
```

which gives a counterexample shown in Figure X.

Related Patterns

Declarative Operation explains when non-determinism is useful.

Canonical Set.

Similarity.

Discussion

Operation Refinement

Check whether one operation refines another

Motivation

It is often useful to model an operation at different levels of detail. You might start, for example, by characterizing the essential properties of the operation's behaviour, and then flesh out the details. Or you may write a detailed descripion of several operations, and only later discover a general property you expect them all to share. In these cases, the more detailed operation is expected to *refine* the more partial operation, by reducing its allowable transitions (and never adding new transitions). Refinement is easily formulated as an assertion to be checked.

Basic Form

An operation op′ *refines* an operation op if the transitions allowed by op′ are also allowed by op′. Given operations defined like this

```
sig State {...}
pred op (s, s′: State) { ... }
pred op′ (s, s′: State) { ... }
```

the refinement check is

```
assert op′_refines_op {
   all s, s′: State | op′ (s, s′) => op (s, s′)
   }
```

Variants

Comparison of refactored version.

Equivalence of operations: birefinement.

Examples

```
module book
sig Addr, Name {}
sig Book {
    addr: Name -> (Name + Addr)
    } {
    all n: Name | n not in n.^addr
    }

pred del1 (b, b′: Book, n: Name) {
    let others = univ - n, ad = b.addr {
        b′.addr = others <: (b.addr) : others + ad.n-n.ad
        }
    }
pred del (b, b′: Book, n: Name) {
    let ad = ^(b.addr), ad′ = ^(b′.addr) |
        all x: Name |
            x = n =
                no x.ad′ + ad′.x ,
                x.ad′ - n = x.ad - n
    }

assert delOK {
    all b, b′: Book, n: Name |
        (del1 (b, b′, n) => del (b, b′, n))
    }
check delOK for 2 but 3 Name, 3 Addr
```

Discussion

Relation to replacement?

Terminology: simulation, refinement, replacement, etc?

Operation Replacement

Check whether one operation can replace another

Motivation

A related, but slightly different, form of refinement arises when an operation is modelled at two levels, one a more abstract level suitable for reasoning about the system's behaviours, and another a more concrete level, suitable for guiding implementation. In this case, the more concrete operation is expected to refine the more abstract operation in a similar way, and a similar check can be applied.

Basic Form

[unfinished]

For implementation refinement, a slightly different notion is needed. An operation `op′` *implementation refines* an operation `op` if the behaviour of `op′` would be acceptable wherever the behaviour of `op` is expected. In terms of pre- and post-conditions, this amounts to the following rules:

- the post-condition of `op′`, when limited to the pre-condition of `op`, must imply the post-condition of `op`, so that any transition `op′` permits from a state in which `op` can be invoked yields a state acceptable to `op`;
- the pre-condition of `op` implies the pre-condition of `op′`, so that `op′` is applicable in any context in which `op` is.

The crucial difference here is the role of the pre-condition. In the basic notion of refinement, the refining operation cannot add new transitions; in implementation refinement, it can add new transitions to long as they are for pre-states that the refined operation rejects. Also, refinement allows the pre-condition to be *narrowed*; implementation refinement only allows it to be *expanded*.

Given two operations defined over the same state space

```
sig State {…}

pred pre_op (s: State) {…}
pred post_op (s, s′: State) {...}
pred op (s, s′: State) { pre_op (s) and post_op (s, s′) }

pred pre_op′ (s: State) {...}
pred post_op′ (s, s′: State) {...}
pred op′ (s, s′: State) { pre_op′ (s) and post_op′ (s, s′) }
```

the implementation refinement checks are:

```
assert op_pre_refinement {
   all s: State | pre_op (s) => pre_op′ (s)
   }

assert op_post_refinement {
   all s, s′: State |
      (pre_op (s) and post_op′ (s, s′)) => post_op (s, s′)
   }
```

Note the role of the precondition in the post-refinement check: the operation op′ is only expected to work appropriately in those states that satisfy the precondition of op.

Over different state spaces. Abstraction functions.

Variants

Comparison of refactored version.

Equivalence of operations: birefinement.

Examples

checking address book

Discussion

Implicit Invariant

Include invariant implicitly in pre- and post-conditions

Motivation

Succinct, simple modelling. Loss of redundancy, but operations can be much simpler, and less mechanistic.

Basic Form

Consider abstract machine with invariant:

```
sig State {
    ...
    }
pred inv (s: State) { ... }
pred op (s, s′: State) { ... }
```

In standard setup, invariant is separate from operations, and we do preservation checks:

```
assert opPreservesInv {
    all s, s′: State | inv(s) and op (s, s′) => inv (s′)
    }
```

In this pattern, invariant is assumed as a fact:

```
sig State {
    ...
    }
pred inv (s: State) { ... }
pred op (s, s′: State) { ... }
fact { all s: State | inv (s) }
```

The effect is the same as including the invariant in every operation, applied to pre and post-states:

```
pred op (s, s′: State) {
    ...
    inv (s) and inv (s′)
    }
```

In practice, when using this pattern, often just make invariant a fact without the predicate, or use a signature fact:

```
sig State {
    ...
    } {
    -- state invariant
    }
```

Variants

Extreme form is *Views*.

Examples

```
module book
sig Addr, Name {}
sig Book {
    known: set Name,
    addr: known -> one Name + Addr
    }
    {
    no iden & ^addr
    }

pred add (b, b': Book, n: Name, a: Addr) {
    b'.known = b.known + n
    b'.addr[n] = a
    (b.known) <: (b'.addr) = b.addr
    }

run add for 3 but 2 Book

assert addOK {
    all b, b': Book, n: Name, a: Addr |
        add (b, b', n, a) =>
            b'.addr = b.addr + n -> a
    }
check addOK for 3 but 2 Book
```

Discussion

Incremental State

Model state incrementally to separate concerns

Motivation

The structure of a system's state often reflects multiple concerns. The state of a distributed system, for example, may include the communication topology, the data stored in the nodes, and the contents of message queues between them. Sometimes, the best way to model such a system is to build the state up incrementally, adding one concern at a time.

Basic Form

The basic form consists of a series of state signatures, related by signature extension:

```
abstract sig State_0 {
    -- field declarations
    } {
    -- invariants over fields of State_0
    }
abstract sig State_1 extends State_0 {
    -- field declarations
    } {
    -- invariants over fields of State_0 and State_1
    }
...
abstract sig State_N extends State_{N-1} {
    -- field declarations
    } {
    -- invariants over fields of State_0 ... State_N
    }
```

Each state signature introduces new fields and invariants that relate them to each other, and to previously introduced fields. Declaring the signatures as abstract ensures that at any stage in the development, any particular state must belong to the most refined signature. The last signature in the series need not be declared as abstract (and makes no difference anyway, since the keyword only affects the relationship between a signature and its extensions).

Operations can be defined at the end, on the final state signature

```
pred op (s, s': State_N) {...}
```

or they can be defined along with the state signatures

```
pred op_N-1 (s, s': State_N-1) {...}
```

and then lifted, if desired, at the end over the final state

```
pred op_N (s, s': State_N) {
   op_i (s, s')
   s'.f_j = s.f_j
   }
```

with a frame condition for each field f_j introduced between the signature over which the original operation was defined (here, $State_i$ say), and the final signature (here, $State_N$).

Variants

A similar pattern can be applied to the operations. Each operation is objectified as a signature, whose instances correspond to executions of the operation:

```
abstract sig Op {
   pre, post: State,
   arg: T
   } {
   – transition constraints
   }
```

A new class of operations can be obtained by extending such a signature, adding new arguments and constraints as needed:

```
abstract sig Op' extends Op {
   arg': T
   } {
   – transition constraints
   }
```

Example

A ring of communicating nodes may be modelled in two phases. In the first, we declare a signature for the components of the state corresponding to the topology (using the pattern *Implicit Invariant*):

```
module incremental_state/ring

sig Node {}
abstract sig Ring {
   nodes: set Node,
   succ: nodes one -> one nodes
   } {
   all n: nodes | nodes in n.^succ
   }
```

We can define operations on this state, such as the operation that patches in a new node after a given node:

```
pred insert (r, r′: Ring, n, after: Node) {
   n != after
   r′.nodes = r.nodes + n
   r′.succ = r.succ ++ after->n ++ n->r.succ[after]
   }
```

In the second phase, we extend the declaration of the state to model the storage of data:

```
sig Block {}
abstract sig DataRing extends Ring {
   blocks: set Block,
   data: nodes some - blocks
   }
```

Each node stores some set of data blocks; every data block associated with the ring is stored on at least one node. Now we can define operations on this extended state, such as the operation that stores a block at some node, and (non-deterministically) some duplicates in succeeding nodes:

pred store (r, r’: DataRing, b: Block, at: Node) { b not in r.blocks some dups: set r.nodes {

– nodes used for duplicate storage form a contiguous sequence

```
        all n: dups | r.succ.n in (dups + at)
        r'.data = r.data + (at + dups) - b
        }
    r'.succ = r.succ
    }
```

The node insertion operation can be lifted over the new state, by invoking the old operation and adding a frame condition saying that insertion does not affect the data stored:

```
pred insert2 (r, r': DataRing, n, after: Node) {
    insert (r, r', n, after)
    r'.data = r.data
    }
```

Alternatively, we can write operations as signatures, and then lifting can be done by signature extension. First, we declare a signature for the class of ring operations:

```
abstract sig RingOp {
    ring, ring': Ring,
    n: Node
    }
```

Now the insertion operation is obtained by extension:

```
abstract sig InsertOp extends RingOp {
    after: Node
    } {
    n != after
    ring'.nodes = ring.nodes + n
    ring'.succ = ring.succ ++ after - n ++ n - ring.succ[after]
    }
```

and can be lifted like this:

```
abstract sig DataRingInsertOp extends InsertOp {}
    {ring + ring' in DataRing and ring'.data = ring.data}
```

Related Patterns

Incremental State is often used in combination with *Implicit Invariant.*

Hierarchical State offers an alternative approach to separating the components of a state.

Discussion

What are the origins of this pattern?

This style of incremental modelling was invented by the developers of the Z specification language. The Z language supports it particularly smoothly, with a construct known as the *schema*.

How do Z's schemas differ from Alloy's signatures?

A schema denotes a set of objects with labelled components, so has much in common with a signature. But schema extensions are purely syntactic, and there is no relationship between the set of bindings of a schema and the bindings of an extension of that schema; in contrast, the atoms of a signature are also atoms of the signature it extends. As a result, signatures can be used for classification in a way that schemas cannot. On the other hand, schemas can be used to express higher-order logical properties that cannot be expressed in Alloy.

Why are operations lifted?

If a set of operations is defined on a particular version of the state, and you want to analyze their interaction with operations defined later on the extended state, you'll need to lift the first set so that all operations act on the same state space. If it's sufficient to analyze the set of operations associated with a given version of the state in isolation, there's no need to lift them.

Hierarchical State

Model state hierarchically for separation of concerns and simpler frame conditions

Motivation

[Not yet written].

The structure of a system's state often reflects multiple concerns. The state of a distributed system, for example, may include the communication topology, the data stored in the nodes, and the contents of message queues between them...

Analysis advantage of fewer atoms!

Basic Form

Variants

Examples

Related Patterns

Discussion

Split Operation

Split an operation into suboperations to factor out exceptional behaviours

Motivation

Key idea: precondition isn't disclaimer; it's the part of the context you're addressing.

Basic Form

Make precondition and postcondition explicit.

```
op = op1 or op2 or op3
```

Variants

Examples

Address book:

```
module book

sig Addr, Name {}
sig Book {
   addr: Name -> (Name + Addr)
   } {
   no ^addr & iden
   }
```

Suppose we want to specify a deletion operation that removes a name, both as something that maps to other names and addresses, and as a target for a name. We probably don't want to just remove all mappings from and to the name, because ...

```
– simple case: n is not mapped to
pred pre_del0 (b: Book, n: Name) {
   some b.addr[n] and no b.addr.n
   }
pred del0 (b, b': Book, n: Name) {
   pre_del0 (b, n)
```

```
    b'.addr = b.addr - (n->univ)
    }
-- n is mapped to, and can be removed without leaving dangling
names
pred pre_del1 (b: Book, n: Name) {
    some n0: Name, x: (Name + Addr) - n | n + x in b.addr[n0]
--  some b.addr.n and all n0: b.addr.n | some x: (Name + Addr)
- n | n + x in b.addr[n0]
    }
pred del1 (b, b': Book, n: Name) {
    pre_del1 (b, n)
    b'.addr = b.addr - (n->univ) - (univ->n)
    }

-- removing n would leave dangling names
pred pre_del2 (b: Book, n: Name) {
    some n0: Name | b.addr[n0] = n
    }
pred del2 (b, b': Book, n: Name) {
    pre_del2 (b, n)
    b'.addr = b.addr - (n->univ) - (univ->n)
    }
```

Analyses:

```
assert delPreCover {
    all b: Book, n: Name |
        pre_del0 (b, n) or pre_del1 (b, n) or pre_del2 (b, n)
    }
check delPreCover for 3 but 1 Book
-- discover case of n not in book at all

assert delPreDisj {
    no b: Book, n: Name |
        (pre_del0 (b, n) and pre_del1 (b, n))
        or (pre_del0 (b, n) and pre_del2 (b, n))
        or (pre_del1 (b, n) and pre_del2 (b, n))
    }
check delPreDisj for 2 but 1 Book
-- disover case of being overlapping for one and not for another
```

Discussion

Implied Precondition

Check whether an operation has an implied precondition

Motivation

Often convenient not to make all preconditions explicit, especially when using *Implicit Invariant.* Can check that a precondition is implied.

Example

```
module impliedPrecondition

sig Node {}
sig Ring {
    nodes: set Node,
    succ: nodes one -> one nodes
    } { all n: nodes | nodes in n.^succ }

pred insert (r, r': Ring, n, after: Node) {
    n != after
    r'.nodes = r.nodes + n
    r'.succ = r.succ ++ after -> n ++ n -> r.succ[after]
    }
assert pre_insert {
    all r, r': Ring, n, after: Node |
        insert (r, r', n, after) => after in r.nodes
    }
check pre_insert for 3 but 2 Ring
```

Discussion

Can you find a hidden precondition without knowing what it is?

No, that's much harder. Use simulation or unsat core.

Local State

Embed state in individual objects

Motivation

The *Abstract Machine* pattern collects all the components of the state together as fields of a single signature. When objects are arranged in a classification hierarchy, and have local state components depending on which sets in the hierarchy they belong to, this pattern is unattractive, because it forces you to pull the state components out from their natural location.

Local State solves this problem. Instead of moving a modifiable field to a state signature, the field is simply extended with an additional column corresponding to states or time points. It thus preserves the modularity of classification, in much the same way that mutable objects are declared in programming languages like Java.

Basic Form

Suppose you have a static model in which a signature X has a field f mapping each X to a Y:

```
sig X {
   f: set Y
   }
```

To turn this into a dynamic model, in which the value of the field can change over time, the standard *Abstract Machine* pattern would move the field to the state signature:

```
sig X {}
sig State {
   f: X -> Y
   }
```

and the result of applying f to x in state s is written s.f[x] or x.(s.f).

Local State allows the field to remain in its natural place, and instead adds a column to it:

```
sig State {}
sig X {
```

```
f: Y -> State
}
```

Now the result of applying f to x **in** state s is written x.f.s.

Fields that are not time-varying do not receive an additional column.

Variants

Dynamic sets with as.

Examples

[Example with classification?]

```
module patterns/local_state/ring_election
open util/ordering[Process] as po

sig Time {elected: set Process}
sig Process {succ: Process, toSend: Process -> Time}

fact ring {
    all p: Process | Process in p.^succ
    }

pred init (t: Time) {
    all p: Process | p.toSend.t = p
    }

pred step (t, t′: Time, p: Process) {
    let from = p.toSend, to = p.succ.toSend |
        some id: from.t {
            from.t′ = from.t - id
            to.t′ = to.t + (id - po/prevs(p.succ))
            }
    }
```

Related Patterns

Often used with *Trace*.

Discussion

Why is the time column last?

Closure etc, easier to write, because can write f.t for relation at time t.

Trace

Model execution sequences of an abstract machine

Motivation

No need to formulate reachability invariant. Can check all linear temporal logic properties, and more.

Basic Form

```
open util/ordering[State] as so
sig State {}
pred op1 (s, s': State) { ... }
...
pred opN (s, s': State) { ... }
pred init (s: State) { ... }
fact traces {
   init (so/first())
   all s: State - so/last() | let s' = so/next (s) |
      op1 (s, s') or ... or opN (s, s')
   }
```

Now you have the traces. Example of checking a reachability property:

```
pred Safe (s: State) { ... }
assert AllReachableSafe {
   all s: State | Safe (s)
   }
```

Variants

Examples

```
module ringelection
open util/ordering[Time] as to
open util/ordering[Process] as po

sig Time {elected: set Process}
sig Process {succ: Process, toSend: Process -> Time}
```

```
fact ring {
    all p: Process | Process in p.^succ
    }

pred init (t: Time) {
    all p: Process | p.toSend.t = p
    }

pred step (t, t': Time, p: Process) {
    let from = p.toSend, to = p.succ.toSend |
        some id: from.t {
            from.t' = from.t - id
            -- only accept id if it's greater than my id
            to.t' = to.t + (id - po/prevs(p.succ))
            }
    }

pred skip (t, t': Time, p: Process) {
    p.toSend.t = p.toSend.t'
    }

fact traces {
    init (to/first ())
    all t: Time - to/last() |
        let t' = to/next (t) {
            some Process.toSend.t => some p: Process | not skip (t,
t', p)
            all p: Process |
                step (t, t', p) or step (t, t', succ.p) or skip (t, t', p)
            }
    }

-- elected processes is set that received their own id in that step
fact defineElected {
    no to/first().elected
    all t: Time - to/first()|
        t.elected =
            {p: Process | p in p.toSend.t - p.toSend.(to/prev(t))}
    }

pred show () {
    some elected
    }
```

```
run show for 3 but 5 Time

assert AtMostOneElected {
    all t: Time | lone t.elected
    }
check AtMostOneElected for 3 but 7 Time

assert AtLeastOneElected {
    some t: Time | some t.elected
    }
check AtLeastOneElected for 3 but 7 Time
```

Related Patterns

Machine Diameter. Abstract Machine.

Discussion

Machine Diameter

Calculate the diameter of a machine for bounded model checking

Motivation

When using *Trace* pattern, risk that traces aren't long enough. Can sometimes show that a certain trace length is sufficient to reach all states.

Basic Form

Trace pattern, plus:

```
pred similar (s1, s2: State) { ... }
pred non_looping_run () {
   all disj s, s': State | not similar (s, s')
   }
run non_looping_run for m but k State
```

Similarity checks components of states. So it says that essentially the same state doesn't recur. If run is inconsistent for some *k*, then *k - 1* is a limit on the trace length that needs to be considered.

Variants

Symmetry, property specific bounds.

Examples

```
module ringelection
open util/ordering[Time] as to
open util/ordering[Process] as po

sig Time {elected: set Process}
sig Process {succ: Process, toSend: Process -> Time}

fact ring {
   all p: Process | Process in p.^succ
   }

pred init (t: Time) {
   all p: Process | p.toSend.t = p
```

```
    }

pred step (t, t': Time, p: Process) {
    let from = p.toSend, to = p.succ.toSend |
        some id: from.t {
            from.t' = from.t - id
            -- only accept id if it's greater than my id
            to.t' = to.t + (id - po/prevs(p.succ))
            }
    }

pred skip (t, t': Time, p: Process) {
    p.toSend.t = p.toSend.t'
    }

fact traces {
    init (to/first ())
    all t: Time - to/last() |
        let t' = to/next (t) {
            some Process.toSend.t => some p: Process | not skip (t,
t', p)
            all p: Process |
                step (t, t', p) or step (t, t', succ.p) or skip (t, t', p)
            }
    }

-- elected processes is set that received their own id in that step
fact defineElected {
    no to/first().elected
    all t: Time - to/first()|
        t.elected =
            {p: Process | p in p.toSend.t - p.toSend.(to/prev(t))}
    }

-- compute machine diameter
pred non_looping_run () {
    no disj t, t': Time | toSend.t = toSend.t'
    }
run non_looping_run for 13 Time, 3 Process
```

Discussion

How well does this work in practice?

Easy to use, but often diameter is too big. This computes an upper bound on the diameter which is often much larger than necessary.