Alloy: A Lightweight Object Modelling Notation

Daniel Jackson
MIT Laboratory for Computer Science
dnj@lcs.mit.edu

Abstract

Alloy is a lightweight, precise and tractable notation for object modelling. It attempts to combine the practicality of UML's static structure notation with the rigour of Z, and to be expressive enough for most object modelling problems while remaining amenable to automatic analysis.

Alloy is a textual notation, of which a subset is also expressible graphically. It has a simple set-based semantics, and a type system that, by treating scalars as singleton sets, sidesteps the problem of undefined expressions, and allows relations and functions to be treated uniformly. It also has a novel mutability construct.

This paper explains what object modelling is, and shows how Alloy can be applied to three problems of rather different flavours. It defines the entire language, by translation to a small kernel with a simple semantics and type system. It explains the rationale underlying Alloy's design, and its divergences from existing languages.

Keywords

Object modelling, formal specification, static structure diagram, Z specification language, Unified Modeling Language, Object Constraint Language, relational calculus.

1 Introduction

Alloy is a new object modelling notation that was designed to meet three criteria: first, to be lightweight—small and easy to use, and capable of expressing common properties tersely and naturally; second, to be precise—having a simple and uniform mathematical semantics; and, third, to be tractable—amenable to efficient and fully automatic semantic analysis.

We designed Alloy because no existing language satisfied all three criteria. UML [Rat97, RJB99], whose object modelling facility consists of a graphical notation (the static structure diagram) and OCL [WK99], a textual constraint language, is not precise, and arguably not lightweight either, at least by comparison to its much simpler predecessor OMT [R+91]. Z [Spi92] is precise but intractable, and not a natural fit for object modelling. NP [JD96a], the language of my Nitpick checker [JD96b], had similar design goals to Alloy, but is also not well suited to object modelling, with its Z-style declarations and a lack of quantifiers.

Alloy takes features from each of these. From UML, it takes diagrams and the navigation syntax for relational image, whose origins are in OMT and Syntropy [CD94] respectively. From Z, it takes set-based semantics, in which classes of objects are modelled as sets of elements drawn from primitive types. From NP, it takes the idea of selecting language features with tractability in mind, and structuring the model to expose opportunities for checking.

The challenge of Alloy's design has been to combine these features smoothly. The key idea is a simple type system that associates primitive types implicitly with *domains*—sets that are not declared as subsets of other sets—and that treats scalars as singleton sets. This allows a diagram to be given a direct textual counterpart (with the additional benefit that a model can always be expressed in text alone), and makes possible a uniform syntax for navigation expressions. It also sidesteps the partial function problem.

Lexical and syntactic concerns have played an important part of Alloy's design. The way in which relation arcs are annotated in the graphical syntax, for example, was chosen to enable a matching textual form, while at the same time being easy to draw by hand and by machine. UML's graphical notation, in contrast, has no textual counterpart in OCL.

Alloy has a novel mutability construct. Consider a self-relation *mother* defined on the type *Person*. We might want to assert that a person can't choose a new mother, or more precisely, that the choice of person associated with a person *p* via the *mother* relation is fixed over the lifetime of *p*. In code, this condition is trivial: we just say that the *mother* field of *Person* is constant. It is not clear, however, how to express the condition at the level of abstraction of the object model, in which objects do not have local state, and the *mother* relation is as much a property of the mother as of the child.

Such a condition is not easy to express in Z. It is not correct to say that the mother relation is fixed, because that would rule out the creation of new persons. And, surprisingly, Z does not accommodate such conditions anyway. A state component cannot be defined as fixed; instead, one must define an operation template in which that component is defined not to change, and then import the template explicitly into all operations. UML, on the other hand, offers a large variety of annotations for relation arcs that includes one (*frozen*) that seems to capture the desired property (although it is hard to tell, since the semantics of UML has not been settled). Alloy's mutability construct is more general, since it can be applied to relations between arbitrary sets (where *frozen* is restricted to relations between classes), and has a simple and precise semantics.

An Alloy specification is divided into paragraphs. Like the schemas of Z, these paragraphs allow a specification to be structured into smaller parts, but more importantly, they provide opportunities for analysis. For example, a definition paragraph defines one state component in terms of others, and implies the property that the component is uniquely determined, which can be mechanically checked.

Alloy is a successor to NP [JD96a]. Like NP, Alloy's tractability was ensured by designing it hand-in-hand with its checker. Alloy's checker, Alcoa (Alloy Constraint Analyzer), to be described in a forthcoming paper, exploits the ideas underlying the most recent version of Nitpick, NP's checker [Jac98], but is more efficient and scales to larger models.

Alloy was motivated by two perceived deficiences of NP:

First, being essentially a subset of Z, NP does not offer a direct encoding of graphical object models. For each domain of the model, a type must be defined; and since declared sets cannot be used in subsequent declarations, simple multiplicity constraints must be written as additional textual formulas. The resulting specification is cluttered and unnatural.

Second, NP is essentially a relational calculus, and has no quantifiers. Although quantifiers can be eliminated from almost every specification in favour of relational operators, the results can be mystifying. For example, to say "everybody likes a winner" in Alloy we might write

all p: Person | all w: Winner | w in p.likes

where *likes* is a relation on *Person* and *p.likes* is the set of persons liked by *p*. In NP, we would have to write something like

Winner.~(Un \ likes)= {}

In this formula, the relational expression following the dot (the transpose of the complement of the *likes* relation) maps persons to persons who don't like them; the expression denotes persons who don't like some winner; and the formula as a whole says that the set of such persons is empty. Needless to say, this style of specification did not win many admirers, despite its terseness. We experimented with an algorithm of Tarski's [TG87] for performing the elimination automatically, but it generated huge relational expressions. It was not until we devised a method for analyzing quantified formulas directly that we were able to consider their inclusion in a language.

Alloy is easy to learn, and is the only object modelling language that can be analyzed automatically at a semantic level. A checking tool, Alcoa, is freely available from the author, along with a language reference manual and a collection of pedagogical examples.

Because of the advantages Alloy offers over existing languages such as Z and UML, it is hoped that it will be adopted in their place for the description and analysis of object models. Alloy is not intended to supersede them, however. Formal specification languages such as Z are more appropriate for detailed component specifications, and for describing properties that cannot be conveniently cast in the first-order framework of an object model. UML provides a collection of notations for many aspects of software development, and it will continue to be popular for visual descriptions of code.

The ideas behind Alloy might also contribute to the development of these languages. For Z, Alloy might suggest a useful subset that is less expressive but simpler and more tractable. For UML, Alloy might suggest an approach to giving a semantics to class diagrams and to the Object Constraint Language, and to integrating its graphical and textual notations (which are currently largely unrelated).

The paper is structured as follows. Section 2 explains what an object model is, in an attempt to dispel some widely held prejudices: that object models are necessarily vague and informal; that they are simply visual representations of program structure; and that they offer no features that are not already available in formal specification languages.

Section 3 presents three small examples of object models to give a flavour of Alloy and how it can be applied in three different areas: in modelling abstract properties of systems and domains; in describing architectural styles; and in recording invariants in code.

Section 4 defines the essence of Alloy: a kernel language with its type system and semantics. Section 5 defines Alloy by translation into this kernel.

Section 6 describes the graphical syntax of Alloy informally; it corresponds exactly to the syntax of the *domain* and *state* paragraphs of the textual language.

Section 7 explains the rationale for the major design decisions in Alloy, making the case in detail for a new language, with explicit comparisons to Z and UML. Section 8 discusses some related work: similar attempts to bring rigour and analyzability to object models. Section 9 comments on some deficiencies of Alloy to be addressed in the future.

Appendices give the syntax of the full language and the translation rules into the kernel.

2 What is an Object Model?

An object model describes a state machine whose states have complex structure, and whose transitions involve complex transformations. A file system, for example, is such a machine: each state is an elaborate graph of files, directories and links, and the transitions are actions such as creating and deleting files and directories. Typically, even when restricted to a handful of objects, a huge number of states can be formed: 5 file system objects, for example, can be configured in 2^{25} ways in the absence of any constraints.

By this definition, a Z specification is an object model. Two factors distinguish object modelling notations such as UML and OMT from formal specification languages such as Z. First, an object model is not usually concerned with primitive types, such as integers. Its essence is the sets of objects and the relations between them. Although UML allows instance variables of classes to be shown, in my experience these are best omitted.

Second, an object model is first-order: sets contain only atoms, and relations map atoms to atoms. There are graphical notations that incorporate higher-order features. The IFO data model [AH87], for example, has a powerset construct, and Fusion [C+94], a notation influenced by Z, allows the values of a set in one object model to be defined as the states of another object model. Such features are rarely used, however, and are not available in OMT or UML.

Object models are often constructed to mirror the problem domain, in order to leave room for further development. This results in more concrete models that eliminate higher-order aspects. For example, a Z specification might view the state of a telephone switch as a partition of the subscribers, with two subscribers in the same equivalence class when some call connects them (see [Mor93], for example). An object model, in contrast, would represent the state as a set of calls, and a partial function mapping subscribers to calls. The notion of a call matches our intuition about the problem domain, provides a hook on which to attach properties of calls (such as their duration), and eliminates the 'set of sets'.

Three misconceptions about object models are widespread. First, an object model is *not* object-oriented. Its value is precisely in postponing the allocation of state to individual objects. It makes no sense at specification time to ask whether the *mother* relation is a property of the parent or the child, or to whom the action of being born belongs. Object models are about objects in the sense that, unlike relational database schemas, they rely on the notion of object identity. But they eschew programming language notions such as subclassing and polymorphism, which are techniques for organizing code, and have nothing to do with behavioural properties of a system. OMT's contribution to object-oriented development was to emphasize the value of an "analysis model" that avoided implementation notions [R+91], an approach that has been bolstered recently by Fusion and Catalysis [DW98], but is unfortunately being eroded by UML.

A second misconception is due to unfortunate terminology. Object models are often described as 'static', in comparison to state transition diagrams, message sequence charts, interaction diagrams, etc., which are 'dynamic'. It is true that, during execution, state changes at the level of the object model are usually less frequent than state changes in these other models. Indeed, the behaviour of a program may be viewed as the superposition of a low-frequency component and high-frequency component, with the object model describing the former. And of course, in an object-oriented development, the object model has a major impact on the syntactic structure of the program. But an object model represents a state machine, and is well suited to describing what researchers in software architecture call 'dynamic architectures'.

Third, the prejudice still lingers that graphical notations are by their very nature informal. Many who concede this point still regard them as poor cousins of textual notations, suitable only for those with limited intellectual ability. Despite powerful arguments in favour of visual representations (in software development [Har88], in reasoning [AW96], and in engineering generally [Fer92]), and despite their widespread use in industry, they have yet to penetrate computer science education. The issue is perhaps not technical at all, but rather one of cognitive disposition: some people like diagrams, and some don't. And oddly, amongst those who don't, formalists and hackers seem to share each others' company.

3 Three Examples

This section presents three examples of object models in Alloy. Their purpose is to illustrate the kinds of problem that are addressed by object modelling, and to impart some of the flavour of Alloy.

The examples are drawn from three different areas. The first example shows how an object model can capture abstract properties of a system or problem domain. This is the area in which semantic data models and formal specifications have most commonly been applied. The second uses an object model to describe an architectural style, namely a collection of rules that characterize, for a family of systems, how components may be assembled. Finding appropriate ways to express such rules is currently a popular area of research [Met96, MK96, Mon98, DR99]. The third example is more low-level: its object model expresses invariants amongst heap-allocated objects in the execution of a program. This is the area most suited to notations such as UML.

An Alloy model is broken into paragraphs. The *domain paragraph* introduces basic sets called domains. The *state paragraph* declares additional sets whose elements are drawn from the domains, and relations between sets. Declarations may include multiplicity constraints (how many members a set may have, how many atoms a relation maps a given atom to, etc), mutability constraints (how sets and relations may change), and may specify that the subsets of a set are disjoint or form a partition. The graphical language can express no more and no less than can be expressed in these two paragraphs; they may be replaced or illustrated by a diagram.

Invariant paragraphs specify additional constraints on the state components. By breaking the constraints into several named paragraphs, the specifier may give some more structure to the model, and enable more fine-grained checking. *Definition paragraphs* give constraints that define redundant state components. *Condition paragraphs* specify constraints that are not required always to hold; they are used to factor out common constraints that would otherwise be repeated in other paragraphs. *Assertion paragraphs* specify constraints that are intended to be valid; these add checkable redundancy to the model. Finally, *opera-*

tion paragraphs specify transitions of the model with constraints that relate pre-states and post-states, the latter being referred to by priming the names of state components.

A few hints about the syntax of Alloy constraints may be helpful. A plain arrow declares a relation (not a function, as in Z). The multiplicity symbols !, ?, and + denote *exactly one*, *zero or one* and *one or more* respectively. They are used to indicate the number of elements in a set, and to constrain relations. A declaration

```
S:T!
```

for example, declares *S* to be a set of one atom (that is, a scalar) drawn from the set *T*, and

```
r:S->T?
```

declares *r* to be a relation from *S* to *T* that associates zero or one *T* with each *S* (that is, a partial function). The expression *e.r* denotes the image of the set denoted by *e* under the relation *r* ; *s & t* is the intersection of *s* and *t* ; *s + t* is their union. The formula *some e* is true when the set denoted by *e* is not empty; *one e* is true when *e* contains one element; *sole e* is true when *e* contains one element or no elements.

The first model, *Sectors*, shown in Figure 1, came out of work on the redesign of an air-traffic control system [JC99]. It describes some basic properties of the organization of airspace, and the assignment of responsibility to controllers. Airspace is divided into *areas*, each of which consists of sectors. There is a controller for each sector, and a supervisor for each area.

Because traffic varies throughout the day, the division into sectors is adjusted dynamically. At night, when traffic drops dramatically, there are fewer controllers on duty, and several sectors are amalgamated into a single sector. To model this, we distinguish *physical sectors*, whose relationship to an area is unchanging, from *logical sectors*, each of which contains some physical sectors, and which may expand and contract. The sets of areas and physical sectors are declared as *fixed*, since the physical topology does not change.

The state declarations include basic constraints, for example that supervisors are mapped one-to-one to areas, and that each physical sector belongs to one logical sector (both *multiplicity* constraints), and that the area to which a physical sector belongs may not change (a *mutability* constraint, expressed by the keyword *static* on the right-hand side of the declaration of the relation *area*).

More elaborate constraints are expressed in the following paragraphs. The definition of *supervises* says the obvious: a supervisor *s* supervises a controller *c* if *c* controls a logical sector containing a physical sector in the area that *s* oversees.

The first invariant paragraph, *Abutment*, gives some properties of the *abuts* relation that models how physical sectors are arranged: it says that no physical sector is regarded as abutting itself, and that if one sector abuts a second, the second abuts the first. The second invariant paragraph, *Division*, specifies how physical sectors may be divided into logical sectors: physical sectors in the same logical sector must be clustered together (and not separated from one another by physical sectors in another logical sector), and must belong to a single area.

Lastly, the operation *Coalesce* describes how logical sectors are merged: one logical sector acquires the physical sectors of a second, which subsequently disappears. The operation includes an explicit precondition that the two logical sectors abut, as well as implicit frame conditions—implied by the static declarations—that the unmentioned components, such as the *abuts* relation, are unchanged.

We have constructed many models in this style with Alloy, such as: a dependence model for self-updating software that describes relationships between components, their specifications and their needs; a model of a graph-drawing program that describes how subgraphs may be elided into abstract nodes, and how visibility of nodes and edges is related to abstraction; and a model of Haystack, an information retrieval system that tracks meta-relationships between records and the agents that created them.

The second model, *Click*, shown in Figure 2, describes the architectural style that underlies a scheme for assembling network routers from predefined modules [M+99]. Each router is made of some number of boxes. Boxes are classified into interfaces, which connect the router to the network, and 'factions', which pass packets within the router. Interfaces in turn are classified into receiving and transmitting interfaces. Boxes have ports, which may be inputs or inputs. There are connections from port to port between boxes (modelled by the *connect* relation) and within boxes (modelled by the *forward* relation, and its transpose *backward*).

At execution time, packets are passed by procedure call. A packet may be passed from an output to an input by a 'push', in which the outputting box calls a procedure of the inputting box to accept the packet, or, vice versa, by a 'pull'. Some ports are only capable of push or pull behaviour. To ensure that boxes are put together in a consistent way, the constraints require matching of the 'personalities' of ports along a flow (namely whether they are pull or push). Some ports may support either pull or push; these are given 'agnostic' personality, and the model requires that these be instantiated to push or pull.

This model makes use of two kinds of paragraph not shown in the first example. Condition paragraphs are used to declare constraints that are either used in other paragraphs or (as here) are analyzed independently to experiment with special cases. We added the condition paragraphs here in order to have the tool generate some instances of the state that are not pathological. Assertion paragraphs add checkable redundancy. The assertion given here turns out to be invalid, and causes a counterexample to be generated.

Examples of models we have constructed of this sort include descriptions of: the legal configurations of a package router [JJ94]; registration and propagation of events in implicit-invocation systems [Jac97], translated from Garlan and Notkin's Z specification [GN91]; composition issues in Microsoft's COM, translated from Sullivan's Z specification [SSM97]; the runtime configurations of DFC, a new architecture for telephone switching [JZ98].

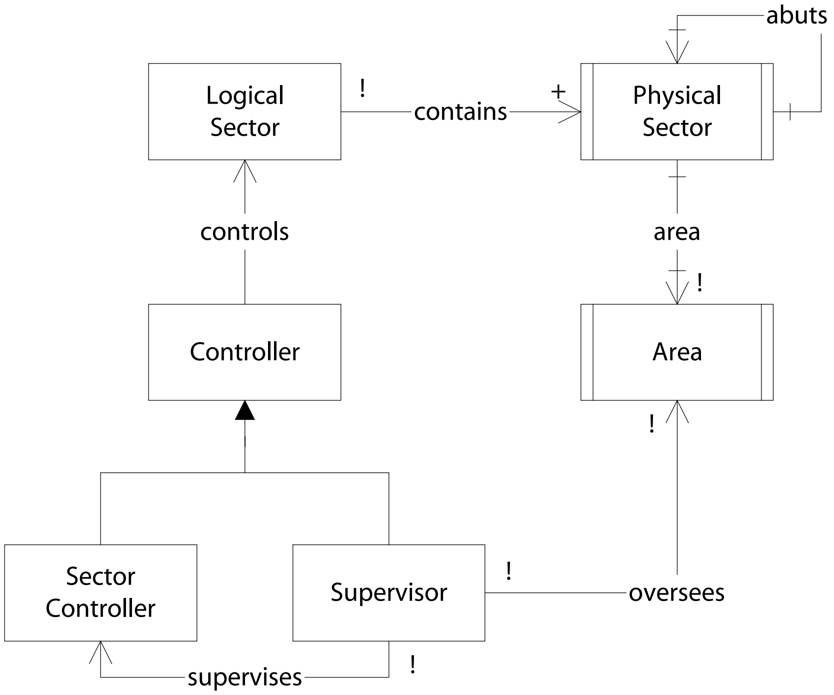
The third model, *Formulas*, shown in Figure 3, uses Alloy to record some invariants amongst the objects of a program. During the construction of Alcoa, we used Alloy to record and reason about various partial models underlying its design. This model concerns the interning of variables within formulas. In the actual program, formulas have a more complex structure, but the model captures the essence of the problem.

Formulas come in several varieties. An elementary formula contains no smaller formulas, and mentions some variables. A binary formula consists of a left formula and a right formula. A quantified formula has a variable and a body, which is another formula. Variables have names.

It is convenient to define an additional relation, *contains*, that maps a formula to the formulas it contains directly. In this model, the relations and sets are to be interpreted more concretely: the set *Var*, for example, denotes a set of objects that belong to the class *Var*, and the relation *name* is an instance variable of the class whose value is a reference to a Java string. The graphical part of the model can be extracted automatically from the code using our Womble tool [JW99].

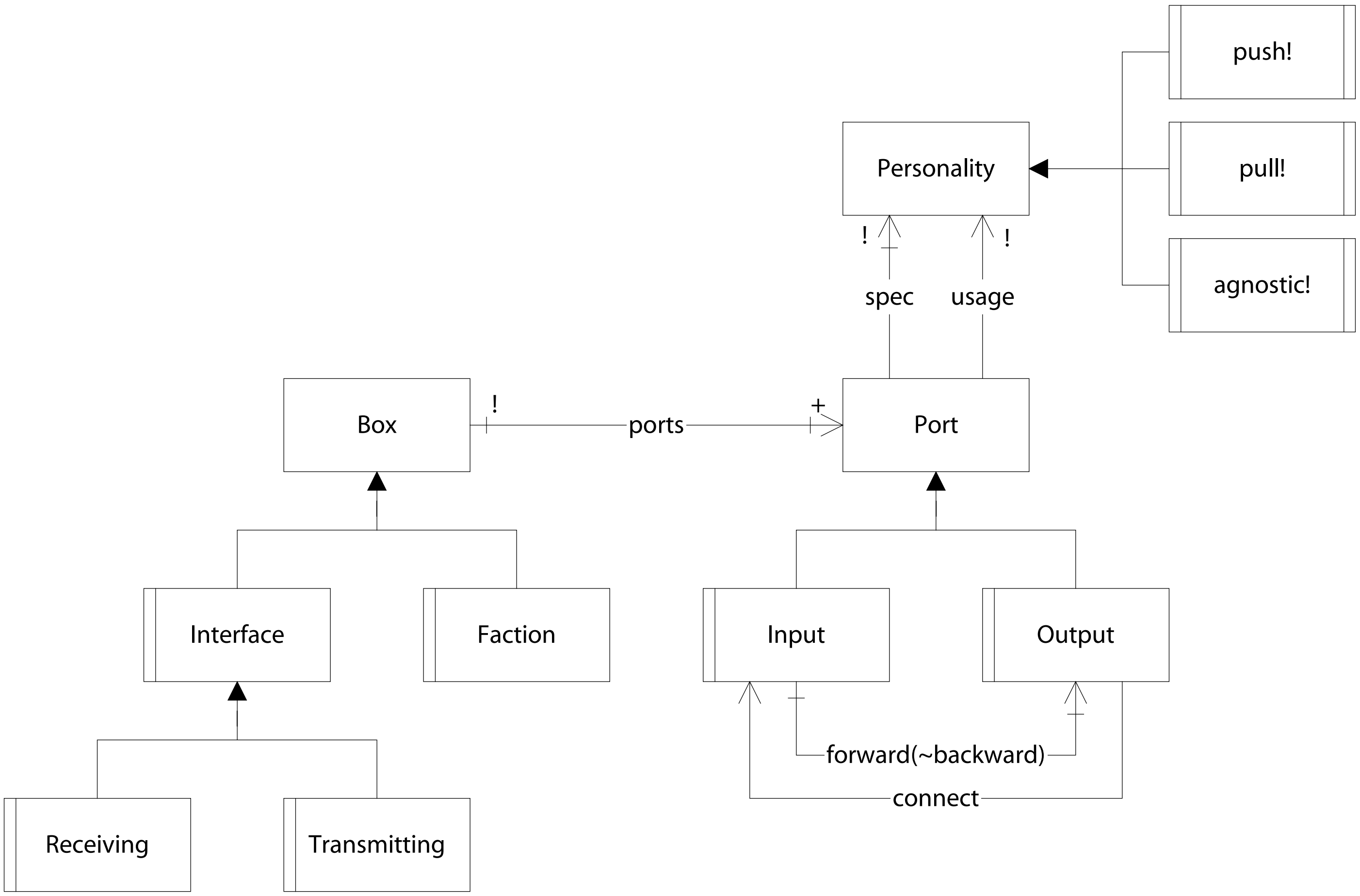
NoShadowing says that variables in quantified formulas may not be shadowed. That is, a quantified formula may not contain another quantified formula with the same variable. *Interned* says that if two variables whose names are equal appear within a formula, they must be represented by the same object. *NoShadowing* is an invariant; it holds for every abstract syntax tree, since a source file in which a variable is shadowed causes an error message to be generated, and no syntax tree is created. *Interned* is a condition: it does not necessarily hold just after the syntax tree has been formed, but is established by execution of a visitor that walks the tree and performs the interning, and is relied upon by later phases, such as type checking.

We constructed a small model in this style for our implementation of a new air-traffic control system component; by checking hypothesized constraints by hand against the code, we found several serious bugs [Lee99]. Alloy has also been used to design the object structures for a financial analysis tool [Iri99]. Beyond this, however, we have not had much experience in applying Alloy at this level, but are investigating its suitability, and the possibility of checking the constraints by static analysis.



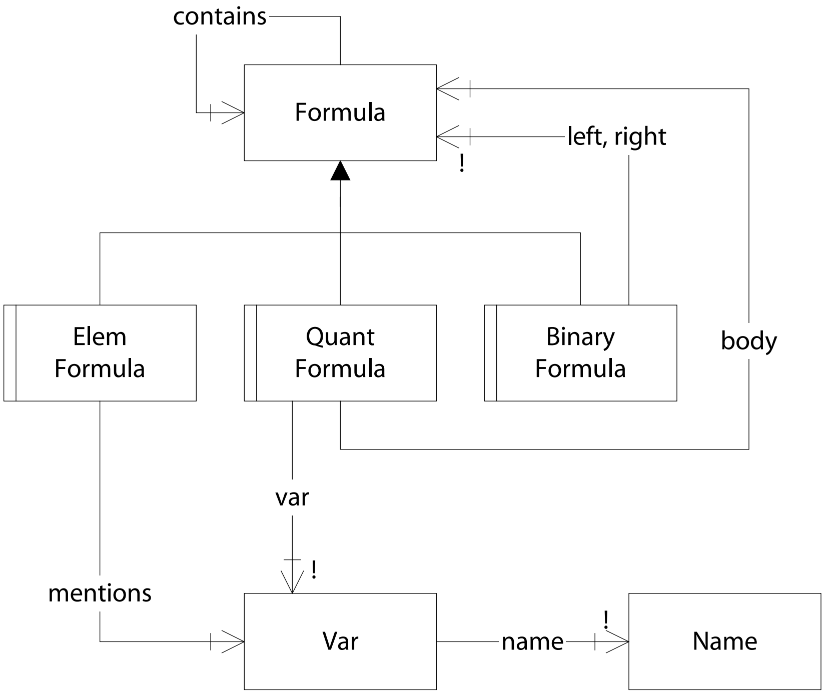
```
model Sectors {
  domain {LogicalSector, fixed PhysicalSector, fixed Area, Controller}
  state {
    partition SectorController, Supervisor : Controller
    supervises : Supervisor! -> SectorController
    oversees : Supervisor! -> Area!
    area : static PhysicalSector -> static Area!
    controls : Controller! -> LogicalSector!
    contains : LogicalSector! -> PhysicalSector+
    abuts : static PhysicalSector -> static PhysicalSector
    }

  def supervises {
    all s | s.supervises = {c | c.controls.contains.area = s.oversees}
    }

  inv Abutment {
    // abuts is irreflexive and symmetric
    no ps | ps in ps.abuts
    all ps1, ps2 | ps1 in ps2.abuts <-> ps2 in ps1.abuts
    }

  inv Division {
    // all physical sectors in a logical sector are reachable from one another
    all ls | all p1, p2 : ls.contains | p1 in p2.*abuts
    // a logical sector may not contain physical sectors from different areas
    all s | sole s.contains.area
   }

  op Coalesce (s1, s2: LogicalSector!) {
    // coalescing s1 and s2 eliminates s2 and causes s1 to contain s2's physical sectors
   // precondition: the sectors are distinct and contain abutting physical sectors
   s1 != s2  && some (s1.contains.abuts & s2.contains)
    LogicalSector' = LogicalSector - s2
    s1.contains' = s1.contains + s2.contains
    // no other sectors change
    all s: LogicalSector - (s1 + s2) | s.contains' = s.contains
    }
}
```

Figure 1: Model describing organization of airspace in a center and assignment of responsibility to controllers


push!
Personality
pull!
!
!
spec
usage
agnostic!
Box
!
ports
+
Port
Interface
Faction
Input
Output
forward(~backward)
connect
Receiving
Transmitting

```
 model Click {
  domain {Box, Port, Personality}
  state {
   partition Interface, Faction : static Box
   partition Receiving, Transmitting : static Interface
   partition Input, Output : static Port
   spec : Port -> static Personality!                          // p.spec is specified usage of p
   usage : Port -> Personality!                                // p.usage is actual usage of p
   ports : static Box! -> static Port+
   partition push, pull, agnostic : fixed Personality!
   connect : Output -> Input                                   // models connections between boxes
   forward (~backward) : static Input  -> static Output        // models connections within boxes
   }

inv {
  all p: Output | p.usage = push -> one p.connect              // pull inputs connected exactly once
   all p: Output | p.usage = p.connect.usage                   // usages match along connections
   all p: Input | some p.forward.usage -> p.usage = p.forward.usage       //… and along internal flows
  all b: Receiving | b.ports in Output && b.ports.usage = push  // receivers have only push output ports
  all b: Transmitting | b.ports in Input && b.ports.usage = pull   // transmitters have only pull input pts
  all p | p.usage in pull + push                               // actual usage must be push or pull
  all p | p.personality = agnostic || p.usage = p.personality  // actual usage must match spec
  }

 // asserts that if a box has one output port, then it cannot have input ports of both usages
 // not valid: there may be no internal flows at all to constrain port personalities
 assert NoMixed {
  all b |
     one b.ports & Output ->
     not (some push.~usage & b.ports & Input
     && some pull.~usage & b.ports & Input)
  }

 // various conditions used to generate sample instance routers
 cond A {some Box}
 cond B {some Receiving some Transmitting  some Faction}
}
```

Figure 2: Model describing the architectural style of Click, a scheme for building modular routers


contains
Formula
left, right
!
Elem
Formula
Quant
Formula
Binary
Formula
body
var
!
mentions
Var
name
!
Name

```
model Interning {
 domain {Formula, Var, Name}
 state {
  partition BinaryFormula, QuantFormula, ElemFormula : static Formula
  left, right : BinaryFormula -> static Formula!
  body : QuantFormula -> static Formula!
  var : QuantFormula -> static Var!
  name : Var -> static Name!
  mentions : ElemFormula -> static Var
  contains : Formula -> static Formula
  }

 // a binary formula contains its left and right formulas;  a quantified formula contains its body
 def contains {
  all f | f.contains = f.left + f.right + f.body
  }

 // a condition stating that variables have been interned:
 // within any quantified formula, variables of the same name are represented by the same object
 cond Interned {
  all qf | all v: qf.+contains.occurs | v.name = qf.var.name -> v = qf.var
  }

 // an invariant stating that variables may not be shadowed
 inv NoShadowing {
  // formula f does not contain a subformula fs whose variable is the same as f's
  all f | no fs: f.contains | f.var /= fs.var
  }
}
```

Figure 3: Model describing interning of variables in formulas in the implementation of Alloy

4 Kernel Syntax and Semantics

Alloy is itself a small language, but is based on an even smaller kernel language. The kernel differs from the full language in two respects. It is a language of formulas with no structuring mechanisms: there is no notion of a definition distinct from an invariant, for example. And as a formula language, although as expressive as the full language in a technical sense, it lacks many convenient syntactic features. We shall give the semantics in terms of the kernel, and then explain how these additional features can be regarded as syntactic sugar.

The top-level syntactic category in the kernel is a *frame*. A frame is a formula with accompanying type declarations, and will be the result of translating a paragraph in the full language. For example,

```
connect: PORT -> PORT
Port: PORT
(all p: PORT | p.connect in Port && p.~connect in Port)
  && (all p: PORT | not (p in Port) || not (p in p.connect))
```

is a frame. It declares two components: *connect*, a relation with left and right atoms of type *PORT*, and *Port*, a set of atoms drawn from the type *PORT*. Its formula is a conjunction. The first conjunct says that the left and right atoms of *connect* must be drawn from the set *Port*. The second conjunct says that *connect* is irreflexive—no port is connected to itself. This is the frame that would result from translating the invariant paragraph of this model:

```
model Example {
domain {Port}
state {connect: Port -> Port}
inv {all p | p !in p.connect}
}
```

This example illustrates most of the differences between the kernel language and the full language: the implicit introduction of a type (*PORT*) associated with the domain *Port*; the constraints implicit in the declaration of a relation (*connect*); the inference of the type of the bound variable (*p*) in a quantified formula, along with a bounding constraint (*p in Port*); simple formula shorthands (*!in*); and the structuring mechanism, which in this case identifies the formula as an invariant to be implicitly included in other paragraphs (not shown).

4.1 Syntax

The syntax of the kernel is shown in Figure 4. Two non-terminals are named to avoid confusion with meta-symbols: *BAR* for a vertical bar, and *PRIME* for a prime mark.

Three kinds of component may be declared. The declaration

```
s : d
```

declares *s* to be a set component, whose atoms are drawn from a set associated with the domain type *d*. The declaration

```
r : d1 -> d2
```

declares *r* to be a relation component, whose left atoms are drawn from a set associated with the domain type *d1*, and whose right atoms are drawn from a set associated with the domain type *d2*. The declaration

r [dx] : d1 -> d2

declares *r* to be an indexed relation component. For each atom *a* in the set associated with the domain type *dx*, there is a relation *r[a]* with left atoms from *d1* and right atoms from *d2*.

Formulas are uninteresting. We have the usual logical operators, and a quantifier. There is only one kind of elementary formula,

e1 in e2

which is true when the expression *e1* denotes a set that is a subset of that denoted by *e2*.

All expressions denote sets. A quantified variable denotes a singleton set; a set component denotes an arbitrary set. The set operators +, - and & denote union, difference and intersection respectively. Set comprehension takes its standard form: {*v: d | F*} denotes the set containing all atoms *a* drawn from the set associated with domain type *d* such that the formula *F* evaluates to true when *a* is substituted for *v*.

The expression *e.s*, which denotes the image of the set denoted by *e* under the relation denoted by *s*, is often called a *navigation expression*, since the image is obtained by 'navigating' along the relation from all starting points in *e*. The qualifiers of navigation expressions are an impoverished form of relational expression: one can take only the transpose (*~s*) or transitive closure (*+s*) of a relation. Relations are specified from indexed relation components always by indexing with a variable, since the index must evaluate to a single atom of the indexing domain type, and an arbitrary expression may denote no value, or several values.

Components are named by identifiers that may be primed (to denote values in the post-state of a transition). Variables may not be primed.

frame ::= decl* formula

decl ::= set **:** domtype | rel : domtype **->** domtype | indexrel [domtype] **:** domtype **->** domtype

formula ::= **not** formula | formula **&&** formula | **all** var **:** domtype BAR formula | expr **in** expr

expr ::= var | set | expr **+** expr | expr **–** expr | expr **&** expr | expr **.** qualifier | **{** var : domtype BAR formula **}**

qualifier ::= rel | indexrel **[** var **]** | **~** qualifier | **+** qualifier

set ::= identifier | identifier PRIME

rel ::= identifier | identifier PRIME

indexrel ::= identifier | identifier PRIME

var ::= identifier

domtype ::= identifier

Figure 4: Syntax of Alloy Kernel

$$\frac{E\,[\,s:\ S\,] \vdash \mathit{decls}\ f}{E \vdash s : S\ \mathit{decls}\ f}$$

$$\frac{E\,[\,r : S \rightarrow T\,] \vdash \mathit{decls}\ f}{E \vdash r : S \rightarrow T\ \mathit{decls}\ f}$$

$$\frac{E\,[\,r:\ X \rightarrow S \rightarrow T\,] \vdash \mathit{decls}\ f}{E \vdash r\,[X] : S \rightarrow T\ \mathit{decls}\ f}$$

$$\frac{E \vdash f}{E \vdash \textbf{not}\ f}$$

$$\frac{E \vdash f, E \vdash g}{E \vdash f\ \textbf{\&\&}\ g}$$

$$\frac{E[v : T] \vdash\ f}{E \vdash \textbf{all}\ v : T \mid f}$$

$$\frac{E \vdash e1 : T,\ E \vdash e2 : T}{E \vdash e1\ \textbf{in}\ e2}$$

$$\frac{c \in \mathit{var} \cup \mathit{set}}{E\,[c : T\,] \vdash v : T}$$

$$\frac{E \vdash e1 : T,\ E \vdash e2 : T, op \in \{+, -, \&\}}{E \vdash e1\ op\ e2 : T}$$

$$\frac{E \vdash e : S,\ E \vdash s : S \rightarrow T}{E \vdash e.s : T}$$

$$\frac{E\,[v : T]\ \vdash f}{E \vdash \{\,v : T \mid f\,\} : T}$$

$$\frac{r \in \mathit{rel}}{E[\,r : S \rightarrow T\,] \vdash r : S \rightarrow T}$$

$$\frac{E \vdash r : S \rightarrow T \rightarrow V, E \vdash v : S}{E \vdash r\,[\,v\,] : T \rightarrow V}$$

$$\frac{E \vdash q : S \rightarrow T}{E \vdash {\sim}q : T \rightarrow S}$$

$$\frac{E \vdash q : T \rightarrow T}{E \vdash +q : T \rightarrow T}$$

Figure 5: Type rules for Alloy Kernel

4.2 Typing

A frame is typed in the context of a type environment that maps variables and state components to types. Types take three forms. Expressions are given set types; the judgment

$E \vdash e : S$

says that, in environment E, expression e denotes a set of elements of domain type S. Qualifiers are given relation types; the judgment

$E \vdash q : S \rightarrow T$

says that the qualifier q denotes a relation from domain type S to domain type T. An indexed relation component r declared as

$r\,[S] : T \rightarrow V$

results in the judgment

$E \vdash r : S \rightarrow T \rightarrow V$

saying that r denotes a function that maps atoms from the domain type S to relations from T to V. In the type $S \rightarrow T$, we shall refer to S as the *left-type* and T as the *right-type*. In the indexed type $S \rightarrow T \rightarrow V$, S is the *index-type*, T the *left-type* and V the *right-type.* Note that there are no scalar types; a scalar is just a singleton set. Also, types may not be nested: S and T may not themselves be types.

Formulas do not have types, but must be type checked. To represent the successful typing of the formula f in the context of declarations $decls$, we use the judgment

$E \vdash decls\ f$

Declarations and quantifiers extend the environment; we write $E\,[\,x1 : t1, x2 : t2, \ldots]$ to denote the environment that is like E but which additionally assigns the type $t1$ to the variable or component $x1$, $t2$ to $x2$, etc.

The type rules are shown in Figure 5 in the conventional form, as an inference system. Each rule has a hypothesis (above the line) and a consequent (below); when the hypothesis holds, the consequent may be inferred. For example, the rule

$$\frac{E \vdash e1 : T,\ E \vdash e2 : T,\ op \in \{+, -, \&\}}{E \vdash e1\ op\ e2 : T}$$

says that if expressions $e1$ and $e2$ can be shown both to have type T, then the compound expression $e1 + e2$ has type T also. Note that, throughout, S and T denote domain types and not arbitrary type expressions.

4.3 Semantics

The meaning of a frame is the set of assignments for which its formula is true. An assignment A is a mapping that associates with a component c a value $A[\![c]\!]$. For set components, the value is a set of atoms; for relation components, a relation; and for indexed relations, a function from atoms to relations. Each subformula and expression is evaluated within the context of an assignment; these assignments, unlike the top-level assignment, may also bind variables. For a variable v, $A[\![v]\!]$ is also a set of values, but always a singleton set.

Values are constructed from a universe $\mathcal{E}$ of atoms. For a given frame, an assignment A is well formed if it maps

1 each domain type appearing in a declaration to a set of atoms:

$$A\llbracket T \rrbracket : \mathbb{P}\, \mathcal{E}$$

2 each declared set component to a set of atoms:

$$A\llbracket s \rrbracket : \mathbb{P}\, \mathcal{E}$$

3 each declared relation component to a set of pairs of atoms:

$$A\llbracket r \rrbracket : \mathbb{P}\, \mathcal{E} \times \mathcal{E}$$

4 each declared indexed relation component to a function from atoms to sets of pairs of atoms:

$$A\llbracket r \rrbracket : \mathcal{E} \rightarrow \mathbb{P}\, \mathcal{E} \times \mathcal{E}$$

(Here $X \rightarrow Y$ denotes the partial function from X to Y.)

All assignments, including those used to give meaning to subformulas and expressions, must be well typed. That is, the value associated with a component must be consistent with the value associated with the constituents of its type:

1 the denotations of distinct domain types are disjoint;

$$A\llbracket S \rrbracket \cap A\llbracket T \rrbracket \neq \emptyset \Rightarrow S = T$$

2 the denotation of a set component is a subset of the denotation of its domain type;

$$E \vdash s : T \Rightarrow A\llbracket s \rrbracket \subseteq A\llbracket T \rrbracket$$

3 each pair in the denotation of a relation draws its first (second) element from the denotation of the left-type (right-type) of the relation's type;

$$E \vdash p : S \rightarrow T \Rightarrow A\llbracket p \rrbracket \subseteq A\llbracket S \rrbracket \times A\llbracket T \rrbracket$$

4 the denotation of an indexed relation component is a function mapping atoms belonging to the denotation of its index-type to a set of pairs whose first (second) elements are drawn from the denotations of the left (right) types:

$$E \vdash p : S \rightarrow T \rightarrow V \Rightarrow A\llbracket p \rrbracket \in A[S] \rightarrow (\mathbb{P}\, A\llbracket T \rrbracket \times A\llbracket V \rrbracket)$$

5 the denotation of a variable is a singleton set:

$$E \vdash v : T \Rightarrow \exists\, a \,.\, A\llbracket v \rrbracket = \{a\} \wedge a \in A\llbracket T \rrbracket$$

In these rules, E is the type environment that is active in the context of the assignment. For a quantified formula, for example, the type environment will include a binding for the quantified variable, and assignments for which the body formula is evaluated must respect the types of this environment. In a different quantified formula, the type environment may map the same variable name to a different type, and different assignments will be legal.

Semantic functions

$$F : Formula \rightarrow Assignment \rightarrow Bool$$
$$E : Expr \rightarrow Assignment \rightarrow \mathrm{P}\, \mathcal{E}$$
$$Q : Qualifier \rightarrow Assignment \rightarrow \mathrm{P}\, \mathcal{E} \times \mathcal{E}$$

Formulas

$$F \llbracket \text{not } f \rrbracket A = \neg F \llbracket f \rrbracket A$$
$$F \llbracket f \;\&\&\; g \rrbracket A = F \llbracket f \rrbracket A \wedge F \llbracket g \rrbracket A$$
$$F \llbracket \text{all } v : T \mid f \rrbracket A = \wedge \{ F \llbracket f \rrbracket (A \oplus v \mapsto a) \mid a \in A \llbracket T \rrbracket \}$$
$$F \llbracket d \text{ in } e \rrbracket A = E \llbracket d \rrbracket A \subseteq E \llbracket e \rrbracket A$$

Expressions

$$E \llbracket v \rrbracket A = A \llbracket v \rrbracket$$
$$E \llbracket s \rrbracket A = A \llbracket s \rrbracket$$
$$E \llbracket d + e \rrbracket A = E \llbracket d \rrbracket A \cup E \llbracket e \rrbracket A$$
$$E \llbracket d - e \rrbracket A = E \llbracket d \rrbracket A \setminus E \llbracket e \rrbracket A$$
$$E \llbracket d \,\&\, e \rrbracket A = E \llbracket d \rrbracket A \cap E \llbracket e \rrbracket A$$
$$E \llbracket e.s \rrbracket A = \{ y \mid \exists x.\; x \in E \llbracket e \rrbracket A \wedge (x, y) \in Q \llbracket s \rrbracket A \}$$
$$E \llbracket \{ v : T \mid f \} \rrbracket A = \{ a \in A \llbracket T \rrbracket \mid E \llbracket f \rrbracket (A \oplus v \mapsto a) \}$$

Qualifiers

$$Q \llbracket r \rrbracket A = A \llbracket r \rrbracket$$
$$Q \llbracket p[v] \rrbracket A = (Q \llbracket p \rrbracket A)\, (i) \text{ where } \{i\} = E \llbracket v \rrbracket A$$
$$Q \llbracket \sim p \rrbracket A = \{ (y, x) \mid (x, y) \in Q \llbracket p \rrbracket A \}$$
$$Q \llbracket +p \rrbracket A = \{ (x, z) \mid \exists y1, y2, \ldots, yn.\; (x,y1) \in Q \llbracket p \rrbracket A \wedge (y1,y2) \in Q \llbracket p \rrbracket A \wedge \ldots \wedge (yn,z) \in Q \llbracket p \rrbracket A \}$$

Figure 6: Semantics of Kernel

The semantics of the kernel is shown in Figure 6. There is a semantic function for each syntactic category: in the context of an assignment, the meaning of a formula is a boolean value; the meaning of an expression is a set; and the meaning of a qualifier is a relation. In the definitions of the semantic functions, $A \oplus x \mapsto a$ is used to represent the assignment obtained by adding to A a binding from x to a; if a binding for x already exists, it is replaced.

Finally, the meaning of a kernel frame *decls f* is the set of all assignments A that are well typed in the type environment induced by *decls*, for which $F\llbracket f \rrbracket A$ is true. We call such assignments *instances* of the frame, avoiding the overloaded term ‘model’. A frame is *consistent* if it has at least one instance, and *valid* if every well-formed assignment is an instance.

It is useful to consider assignments of a bounded size. A scope is a function that maps domain types to natural numbers. We say that A is within scope σ if for each domain type T,

$$| A\llbracket T \rrbracket | \leq \sigma(T)$$

Since Alloy is not decidable, whether a frame is valid or consistent cannot be determined automatically in general. But it is possible to perform the check within a scope, and often the scope that is required to show that a frame is consistent or invalid is surprisingly small.

A frame might correspond to a collection of state invariants, in which case its meaning is the set of states satisfying the invariants—that is, each instance is a satisfying state. Alternatively, it may correspond to an operation, in which case its meaning is a set of state transitions. And finally, it may represent the negation of an assertion claimed to be true, in which case its meaning is a set of counterexamples: states if the assertion is about states, and pairs of states if about transitions.

4.4 Expressiveness

Since the kernel language represents the essence of the full language, it can act as a surrogate for theoretical purposes. We have yet to conduct a serious investigation into Alloy's expressive power, but a few key properties are clear.

Alloy is a first-order language: there are no relations over relations, for example, or quantifiers over sets or relations. On the other hand, the language is more expressive than first-order logic with at most dyadic predicates, since it includes transitive closure, which has no first-order axiomatizion, and is thus undecidable. Since tuples can be modelled with extra primitive types, and appropriate projection functions, an argument could be made (similar to that made by Tarski for the relational calculus [TG87]) that Alloy subsumes first-order logic with predicates of arbitrary arity. At any rate, Alloy is at least as powerful as its predecessor NP (for which a kernel is given in [Jac98]), and is less powerful than Z.

5 Full Language

The full Alloy language is defined as extensions of the kernel, most of which are simple syntactic sugar. In the sections that follow, we show some fragments of the grammar of the full language, and give translation rules to the kernel; these are collected together in the appendix. Two grammatical conventions are used: angle brackets to indicate optional phrases, and *x,* for a comma-separated list of *x*'s.

5.1 Paragraph Structuring

An Alloy model consists of a sequence of paragraphs, starting with a domain paragraph, followed by a state paragraph, then any number of definition, invariant, condition, assertion and operation paragraphs. Condition paragraphs may be referenced within other paragraphs; declaration must always precede use.

```
model ::= model model-name { domain {domdecl*} state {compdecl*} para*}
para ::= inv ⟨name⟩ { formula* }
        | def name { formula* }
        | cond name ⟨arglist⟩ { formula* }
        | assert ⟨name⟩ { formula*}
        | op name ⟨arglist⟩ { formula*}
```

With the exception of definition paragraphs, which are named by the components they define, and domain and state paragraphs that are not named, a paragraph may be given any name, so long as it is unique. As a convenience for small models, there may be one anonymous invariant and assertion paragraph.

5.2 Domain Paragraph: Domains & Implicit Typing

The domain paragraph comes first. It takes the form

dompara ::= **domain {**domdecl***}**
domdecl ::= ⟨**fixed**⟩ *set*

Each declaration introduces a set component called a *domain*, and, associated with it, a domain type. The full Alloy language, unlike its kernel, is implicitly typed, and the syntax does not allow type names to appear in a model. As a notational convention for discussing the typing scheme, we shall use the domain name in upper case to refer to its domain type. For every set *d* that appears in the domain paragraph, then, it is as if the kernel declaration

d : D

had been made.

Components, including domains, may change over time. In an operation paragraph, the values of the component *c* before and after execution will be referred to as *c* and *c′*. For each set *d*, therefore, an additional declaration

d′ : D

is implicit, except for domains declared as *fixed*, which are unchanging and may not be primed.

Some properties of domains follow from their implicit typing. Since the semantics requires that the sets assigned to types be disjoint, domains must also be disjoint. And since *d* and *d′* have the same type, an operation that moves an atom from one domain to another cannot be expressed.

For example, the domain paragraph

domain {Person, fixed Gender, Name}

produces the kernel declarations

Person: PERSON
Person′: PERSON
Gender: GENDER
Name: NAME
Name′: NAME

These admit operations that create and destroy persons and names, but the set of genders is fixed. No person is a name, and no person may become a name.

5.3 State paragraph: Multiplicity & Mutability

In the state paragraph, all components except domains are declared.

statepara ::= **state {***compdecl****}**
compdecl ::= *setdecl* | *reldecl*
setdecl ::= ⟨**disjoint** | **partition**⟩ *set*, : ⟨**fixed** | **static**⟩ *set mult*
reldecl ::= *relx* ⟨**(~** *relx* **)**⟩ , : ⟨**static**⟩ *set mult* **->** ⟨**static**⟩ *set mult*
relx ::= *rel* | *indexrel* **[** *set* **]**
mult ::= **?** | **!** | **+**

The set declaration

s : t

constrains the new component *s* to be a subset of the previously declared component *t*, and implicitly assigns to *s* the type *T* previously assigned to *t*. The relation declaration

r : s -> t

constrains the new component *r* to be a relation whose left atoms are drawn from the set *s*, and whose right atoms are drawn from the set *t*, and implicitly assigns it the type $S \to T$. The declaration

r [x] : s -> t

introduces an indexed relation component *r*, so that for each atom *a* in the set *s*, *r[a]* is a relation of type $S \to T$ with left (right) atoms are drawn from *s* (*t*).

Several components may be declared at once, and along with a relation component its transpose may be declared. For example,

r , p(~q) : s -> t

introduces three relations. Two of these, *r* and *p*, have the same types and constraints; the third, *q*, is defined to be the transpose of *p*, and thus has the same types and constraints but in mirror form. When several set components are declared together, the collection may be marked as disjoint or as forming a partition; thus

partition a, b, c : s

declares *a*, *b* and *c* to be disjoint, and to have *s* as their union.

A simple class of invariants, called *multiplicity constraints*, may be specified as part of a component's declaration. The multiplicities *?*, *!* and + are read *zero or one*, *exactly one* and *one or more*. Appended to a set declaration, a multiplicity constrains the number of atoms in the set, so, for example

s : t+

declares *s* to be a non-empty set,

s : t !

declares *s* to be a singleton (that is, a *scalar*), and

s : t ?

declares *s* to be an empty set or a singleton (that is, an ‘option’). In relation declarations, multiplicities allow various special kinds of relation to be declared. The declaration

```
r : s m -> t n
```

where *m* and *n* are multiplicities, constrains *r* so that it maps each left atom in *s* to *n* atoms in *t*, and maps *m* left atoms to each right atom. For example,

```
r : s -> t ?
```

declares r to be a partial function: each left atom is mapped to at most one right atom.

The *fixed* and *static* keywords introduce *mutability constraints*. A set declared to be fixed, like a fixed domain, is unchanging. As with domain declarations, each state declaration results in an additional declaration and appropriate constraints for the primed version of the component. For a set component declared as fixed

```
s : fixed t
```

however, no declaration for *s′* is created, but the constraints that would have applied to the primed version apply to the unprimed version also, so in this case the declaration implies not only that *s* is a subset of *t*, but also that *s* is a subset of *t′* (unless *t* itself is fixed).

A *static* set component is not fixed, but its evolution is circumscribed. The rule is subtle but not hard to apply in practice. Intuitively, the classification of atoms induced by a static set must be static and not dynamic: an atom that belongs to a domain both before and after a transition, cannot move in or out of a static set. This constraint can be summarized as a formula that is implicitly included in any operation. For the declaration

```
s : static t
```

the constraint is

```
all x : t & t' | x & s = x & s'
```

It is instructive to compare this to the constraint induced by

```
s : fixed t
```

which might be written

```
all x : D | x & s = x & s'
```

(where *D* is the domain type of *s*) if mention of *s′* were legal. This constraint is equivalent to

```
s = s'
```

since *s* must be a subset of the domain type, but it makes clear the crucial difference between *fixed* and *static.*

For example, for a domain *Person* it would be appropriate to declare *Woman*, the subset of persons that are female, as static

```
Woman : static Person
```

since an individual cannot become a woman or cease to be a woman except by becomoing or ceasing to be a person. But *Parent* would not be static

```
Parent : Person
```

since a *Person* can become a parent by giving birth to a child.

Novices seem to find mutability especially confusing for scalars. A file system model might include the declaration

```
Root : fixed Dir !
```

One common misconception is that this somehow makes the properties of the root directory fixed. But in an object model, there is no notion of local object state; the name of the root directory, the objects it contains, and so on, will be determined by the values of some relations. A more subtle issue arises when the identity of objects may not be readily observable: it is unclear how one would notice a transition in which the root were replaced by a different object with identical properties. Static scalars, incidentally, are rare, since the creation of one atom would have to coincide with the death of another.

Relations may also carry mutability constraints. A relation annotated as *static* on the left-hand side is said to be *left-static*; one annotated on the right-hand side is *right-static*. (We also use the terms *source-* and *target-static* respectively, which are more convenient when looking at a diagram since a relation drawn as an arrow from right to left will have its 'left set' on the right-hand side!) A relation may be both left- and right-static.

A right-static relation from *S* to *T*

```
r : S -> static T
```

may change only in a way that preserves the image of each atom that exists in *S* before and after the transition. This can be summarized in a formula that is implicitly imported into every operation:

```
all s: S & S' | s.r = s.r'
```

Conversely, a left-static relation preserves the image of each right atom under the transpose of the relation; from

```
r : static S -> T
```

we would derive

```
all t : T & T' | t.~r = t.~r'
```

The reader may wonder whether Alloy has fixed, as well as static, relations. It does not; surprisingly we have found little use for such a notion. Relations that represent the state of a system always change as new atoms are created and deleted; what often does not change, as captured by the static constraints, is the set of atoms associated with a given atom. Should a fixed relation be needed, it can easily be specified by declaring left and right sets that are fixed, and by declaring the relation itself as left- and right-static.

Here are some examples of static relations. The relation that maps a president to her vice-president is static, so long as vice-presidents are not allowed to die in office or be fired:

```
President, VicePresident : Person?
vp : President -> static VicePresident !
```

Note that the static annotation constrains only the change of vice president associated with a *president*. It does not say that which vice president is associated with a *person* may not change, thus allowing a person to acquire a vice president on being elected president. In the formal translation of the constraint, this distinction is evident in the bound that causes the relevant atoms to range over the specified set, and not over the domain.

A link in a file system points to some other file system object. To model this, we might declare a static set of links, and a right-static relation that maps a link to the object it points to:

```
Link : static FileSystemObject
link : Link -> static FileSystemObject !
```

The set of links is static, since an existing file system object cannot become a link, and a link cannot be decommissioned without being deleted; the relation is static because a link cannot be redirected.

The relation that maps a mother to her biological children is left-static, since a child cannot acquire a new biological mother:

```
Mother : Person
children : static Mother  -> Person
```

These mutability constraints often embody subtle modelling decisions. In this case, for example, either the death of a mother during the life of a child is ruled out, or the atom modelling the mother is assumed to exist after the mother's death (as would be appropriate, for example, in a specification of a family tree).

The combination of multiplicity and mutability constraints allow enumerations to be declared conveniently without extending the language:

```
partition Red, Blue, Green : fixed Colour !
```

5.4 Formulas in Paragraphs: Quantifier Shorthands

The domain and state paragraphs declare state components, along with some rudimentary constraints. More elaborate constraints are expressed in the *formula paragraphs*. Before discussing the roles of the various kinds of formula paragraph, we explain the notation for formulas that all formula paragraphs share.

The body of a formula paragraph is a sequence of formulas, implicitly conjoined, each satisfying the following grammar:

```
formula ::=
  not formula | formula logic-op formula |
  quantifier var-decl , BAR formula |
  expr comp-op expr |
  quantifier expr |
  ( formula ) |
  paragraph ⟨( expr , )⟩

var-decl ::= var , ⟨: expr⟩

logic-op ::= && | || | -> | <->
comp-op ::= in | = | !in | != | /= | /in
quantifier ::= all | some | no | sole | one

expr ::= var | set | expr expr-op expr | expr . qualifier | { var-decl BAR formula } | ( expr )
expr-op ::= + | - | &

qualifier ::= rel | indexrel [ var ] | ~ qualifier | + qualifier | * qualifier
```

Formulas in the full language differ from formulas in the kernel language in several respects, of which only one is non-trivial: the treatment of quantifier bounds. Expressions are identical, except for set comprehensions (which are much like quantified formulas).

The full language adds the standard logical operators (such as implication), comparison operators (such as equality), and relational operators (reflexive-transitive closure). Two special comparison operators are added to remedy an oddity that arises from Alloy's treatment of scalars and partial functions. The formula

```
x.f in e
```

is true when the scalar *x* is not in the domain of the partial function *f*, because then *x.f* denotes an empty set. Often, a specifier writing this meant to say that *x* is in the domain of *f*, and that the result of the application, a scalar, belongs to the set *e*. This is written

```
x.f /in e
```

which is short for

```
x.f  in e  &&  some x.f
```

with the side-condition *some x.f* (discussed below) stating that the set *x.f* is non-empty. Similarly, for scalars *x* and *y*, and partial functions *f* and *g*, *x.f /= y.g* says that *f* maps *x* to the same value that *g* maps *y* to, where, in contrast, *x.f = y.g* is true additionally when neither is mapped.

For quantified formulas, the full language offers a richer grammar of variable declarations. It includes standard conveniences, such as bounding a variable with an expression and declaring several variables at once. Surprisingly, it *excludes* the declarations of the kernel, so the kernel formulas are not strictly a subset of the formulas of the full language. In the kernel, a variable is declared with its domain type. In the full language, domain types never appear anywhere; quantified variables must be bounded (explicitly or implicitly) by set expressions. The rationale for this is explained below (Section 7.3).

To explain the quantified formula syntax, we shall present a series of transformations that result in a kernel formula.

1 Multiple declarations are expanded. A formula of the form

   ```
   Q decls v | f
   ```

   where *decls* is a sequence of comma separate declarations and *Q* is a quantifier, each of one or more variables, is expanded to

   ```
   Q d0 ; d1 ; d2 ; … | f
   ```

   where *di* is the declaration of the *ith* variable *vi*, and is either of the form *vi: ei*, where *ei* is an explicit bound, or of the form *vi* alone, where the bound is implicit. For example,

   ```
   all a, b : s, c | f
   ```

   becomes

   ```
   all a :s ; b : s ; c | f
   ```

2 Missing bounds are inferred. Consider a formula of the form

   ```
   Q decls1 v  decls2 | f
   ```

where *Q* is any quantifier, and *decls1* and *decls2* are lists of declarations (possibly empty, separated from *v* by semicolons), and *v* is a variable declared without a type or bound. By analysis of the body formula *f* using the inference rules of Figure 5, the type of *v* is inferred. An error is raised either if no such type exists, or if the type is not uniquely determined. Otherwise, if *D* is in the inferred domain type, the formula is expanded to

Q decls1 v : d decls2 | f

where *d* is the domain set associated with *D*. For example, in the context of the declarations

domain {Box, Port }
state {box : Port -> Box }

the formula

all p | one p.box

is expanded to

all p : Port | one p.box

since *p* is inferred to have type *PORT* from the expression *p.box*, and *Port* is the domain associated with that type.

3 Additional quantifiers that are not part of the kernel are translated away. There are two standard quantifiers:

some decls | f becomes *not all decls | not f*
no decls | f becomes *not some decls | f*

and two special quantifiers. The formula

sole v : e | f

says that *f* is true for at most one value of *v* in *e*. It is translated to

all x | (some v : e | v = x && f) -> (no v : e – x | f)

The formula

one v : e | f

says that *f* is true for at exactly one value of *v* in *e*, and is short for

(some v : e | f) && (sole v : e | f)

4 Multiple declarations are unwound into nested quantifications; the formula

all v0 : e0 ; v1 : e1 ; … | f

becomes

all v0 : e0 | all v1 : e1 | …| f

5 Bounds are eliminated:

all v : e | f

becomes

all v : D | v in e -> f

where *D* is the domain type of the expression *e*.

Set comprehensions are treated like quantified formulas: missing bounds are inferred, and then bounds are eliminated:

{ v | f }	becomes	*{ v : d | f }* where *d* is the domain corresponding to the domain type *D* of *v*
{v : e | f }	becomes	*{v : D | v in e && f }*

Finally, the quantifiers are used in place of set constants. The formula *Q e*, where *Q* is any quantifier and *e* is an expression, is short for *Q v | v in e*. So *no e*, *some e*, *sole e* and *one e* state that *e* contains no elements, some elements, at most one element and exactly one element. Because of the implicit bound of the quantifier, *all e* means that *e* contains every element of the relevant domain (that whose type is the same as the type of *e*). This leads to the initially surprising (but actually rather intuitive) consequence that

all e && no e

can be true, when the domain itself is empty.

5.5 Paragraph References

The formulas of one paragraph may be implicitly or explicitly imported into another. The rules are as follows:

1 The constraints implicit in the declarations of the state paragraph are included in every other paragraph; for operation paragraphs, and for assertion paragraphs that mention primed variables, primed versions of the constraints are included too. The primed version of a formula is obtained by priming every appearance of every component that is not declared to be fixed.

2 An invariant paragraph or condition paragraph that mentions a defined component implicitly imports the corresponding definition, and transitively includes definitions of components mentioned in the definition.

3 An assertion paragraph implicitly imports all invariants, and if its formula (after expansion of references – see below) contains primed variables, it includes primed versions of the invariants too.

4 An operation paragraph implicitly imports all invariants and their primed versions.

5 Conditions and operations may be referred to within other paragraphs as formulas. For a condition or operation paragraph with name *N* and declared arguments *a0, a1, …* of domain type *D0*, *D1*, …, a reference of the form *N(e0, e1, …)* is allowed, where the numbers of formals and actuals match, and expression *ei* has the type *Di*. For a condition, the reference *N' (e0, e1, …)* is also allowed; the priming of the paragraph name causes a primed version of the condition to be included instead.

When paragraphs are imported, arguments replace formals in the obvious way: the formula of the imported paragraph is 'inlined' with the formals replaced by the actuals.

Arguments are declared like set components:

condition ::= **cond** *name* ⟨*arglist*⟩ { *formula* }
operation ::= **op** *name* ⟨*arglist*⟩ { *formula* }
arglist ::= **(** *argdecl,* **)**
argdecl ::= *arg,* **:** *set mult*

The same constraints are inferred from the declarations, and are implicitly conjoined to the paragraph formula.

Conditions may not mention primed components. To apply a condition to the post-state, the reference to the condition is itself primed; a condition that relates pre- and post-states is not a condition, but an operation. Invariants and definitions also never mention primed variables.

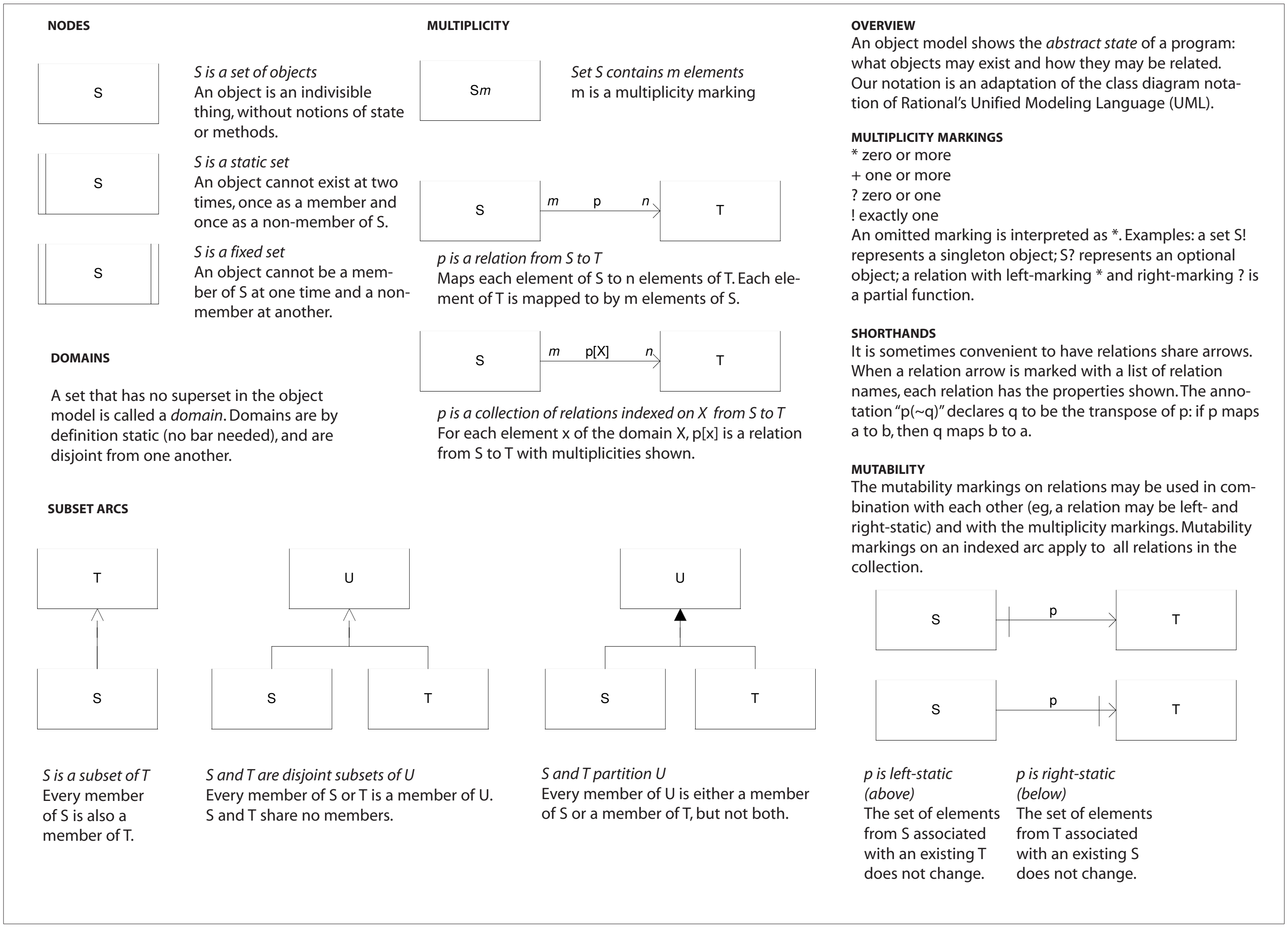
The grammar of Alloy permits a set declaration to use a primed set on the right-hand side. This feature may only be used in the declaration of operation arguments. The operation

op Add (n: Name'!, d: Date'!) {n.birthday' = d}

for example, specifies that, after execution, the birthday associated with name *n* is *d*. The argument declarations imply the constraints

n in Name'
one n
d in Date'
one d

Had we omitted the prime marks in the declarations, the implied constraints would have required *n* and *d* to exist in the pre-state.



NODES

S is a set of objects
An object is an indivisible thing, without notions of state or methods.

S is a static set
An object cannot exist at two times, once as a member and once as a non-member of S.

S is a fixed set
An object cannot be a member of S at one time and a non-member at another.

DOMAINS

A set that has no superset in the object model is called a *domain*. Domains are by definition static (no bar needed), and are disjoint from one another.

MULTIPLICITY

Set S contains m elements
m is a multiplicity marking

p is a relation from S to T
Maps each element of S to n elements of T. Each element of T is mapped to by m elements of S.

p is a collection of relations indexed on X from S to T
For each element x of the domain X, p[x] is a relation from S to T with multiplicities shown.

SUBSET ARCS

S is a subset of T
Every member of S is also a member of T.

S and T are disjoint subsets of U
Every member of S or T is a member of U. S and T share no members.

S and T partition U
Every member of U is either a member of S or a member of T, but not both.

OVERVIEW
An object model shows the *abstract state* of a program: what objects may exist and how they may be related. Our notation is an adaptation of the class diagram notation of Rational's Unified Modeling Language (UML).

MULTIPLICITY MARKINGS
* zero or more
+ one or more
? zero or one
! exactly one
An omitted marking is interpreted as *. Examples: a set S! represents a singleton object; S? represents an optional object; a relation with left-marking * and right-marking ? is a partial function.

SHORTHANDS
It is sometimes convenient to have relations share arrows. When a relation arrow is marked with a list of relation names, each relation has the properties shown. The annotation "p(~q)" declares q to be the transpose of p: if p maps a to b, then q maps b to a.

MUTABILITY
The mutability markings on relations may be used in combination with each other (eg, a relation may be left- and right-static) and with the multiplicity markings. Mutability markings on an indexed arc apply to all relations in the collection.

p is left-static (above)
The set of elements from S associated with an existing T does not change.

p is right-static (below)
The set of elements from T associated with an existing S does not change.

Figure 7: Graphical Sublanguage

6 Graphical Syntax

Alloy has a graphical syntax (Figure 7). A diagram can only express state, and not operations, and only those constraints that are expressible in the domain and state paragraphs. The graphical elements correspond directly to the elements of the domain and state paragraphs.

Each set component is represented as a box. The set declaration *S : T m* , where *S* and *T* are sets and *m* is a multiplicity symbol, is represented as an arrow with a triangular head from a box marked *S m* to a box marked *T*. Domains are not specially marked; they are simply the sets with no outgoing triangular arrows. If a set is static, it has a vertical stripe down the left-hand side of the box; if it is fixed, it has a stripe on the right too. Several sets may be connected to a single set with arrows that meet in a single arrowhead; this declares the sets to be disjoint. If the arrowhead is filled, the collection additionally forms a partition.

A relation *r* from *S* to *T* is represented as an arrow, with a smaller, open head, and labelled *r*, from the box labelled *S* to the box labelled *T*. Multiplicity symbols are written at the ends of the arrow. If the relation is source-static, the tail of the arrow is marked with a small hatch; if it is target-static, the head is marked.

Relations with identical declarations may share a single arrow, labelled with a comma-separated list of relation names. An indexed relation declared textually as

```
r [X] : S -> T
```

is represented by a relation arrow from *S* to *T* labelled *r*[*X*]. Similarly,

```
p (~q) : S -> T
```

is represented by a relation arrow from *S* to *T* labelled *p(~q)*.

7 Rationale

Existing languages have the great advantage of familiarity; a new language that offers only small advantages over its predecessors stands no hope of adoption. For this reason, we now explain and justify, in summary, the respects in which Alloy differs from existing languages. Two essential qualities mark the language as different:

- Alloy is tractable; its checker (to be described in a forthcoming paper) can usually handle with ease a model with a dozen state components. Of course many object models are much larger than this, but their constraints and operations can usually be analyzed within submodels of this size. No existing language offers comparable checking: users must choose between shallow syntactic checks on the one hand, and labour-intensive theorem proving on the other. Z is too expressive to be tractable; UML is not sufficiently well defined for tool support to be viable.

- Alloy integrates its textual and graphical notations. Data modelling languages, such as IFO [AH87] and SDM [HM81], do this too, but they are far less expressive. UML forces the combination of graphical and textual notations, since the class diagram cannot be expressed textually, but it provides no common semantic base. Making sense of UML is an ongoing research project.

7.1 Lexical & Syntactic Issues

Alloy is written in plain old ASCII, and requires no special typefaces. To my mind, Z's dependence on Latex is a serious mistake: although it results in prettier specifications, it makes it awkward to pass specifications around by email, to embed them in documents, to refer to them in code, and so on.

Alloy's graphical notation is designed primarily to be easy to draw by hand, and to require only standard symbols so that drawing by machine is not too hard. The multiplicity symbols were chosen to be appropriate also in textual form; UML's symbols (eg, 0..1 for ?) do not sit comfortably within a textual declaration.

The graphical syntax is monotonic: adding an element to a diagram cannot cause it to say less. OMT's multiplicities, which are otherwise very appealing, are not monotonic: to turn a relation into a total function, the solid blob on the target end must be *erased*. Similarly, Syntropy's state types correspond to non-static sets, but the frame condition associated with a static set is obtained also by deleting a marking from the diagram.

7.2 Paragraphs

In Alloy, parts of a specification that play different roles are syntactically distinguished. This brings three benefits. First, specifications are often shorter: there is no need, as in Z, to explicitly import invariant paragraphs or to specify mutability constraints in an operation template.

Second, the specification expresses important methodological distinctions: the difference between invariants and definitions, for example, and between constraints of the specification proper and redundant constraints that are intended to follow. These distinctions are not new of course; Michael Jackson has discussed the former (and its omission from languages such as Z) at length [MJa95], and the Larch Shared Language [GHM90] has an *implies* clause for the latter.

Third, a tool can exploit the distinctions. Our tool, for example, has a toggle that allows invariants to be 'turned off': it can thus generate executions of an operation from pre-states that violate the invariant, or check that operations preserve invariants. Distinguishing definitions from invariants suggests more elaborate forms of checking (not currently done by our tool): for example, one might check that for each legal state (that is, assignment of values to all state components except for the one being defined), there is exactly one value of the defined component.

These gains come with an inevitable loss of generality. Alloy is a language for describing state machines; Z, in contrast, is a "system of notation for building structured mathematical theories", and has "no necessary connection with computer programming" [Spi92, p.128]. This explains both the strength and the weakness of Z. It is, on the one hand, remarkably powerful and flexible. On the other hand, its very generality prevents it from fitting any problem perfectly, and is a liability for automatic checking.

In UML, defined components are distinguished in the graphical but not in the textual notation. There is no notion of assertions.

7.3 Implicit Typing

Following Z, Alloy is set based; UML is instead class based. Each box in a UML diagram is a class with its own type. Consequently, expressions that are well-typed in Alloy cannot be written in UML, and type coercions are needed. For example, given these declarations

> *Married: Person*
> *partition Man, Woman: static Person*
> *spouse: Married! -> Married!*

we can say that the spouse of a married man is a woman by writing

> *all p: Married & Man | p.spouse in Woman*

In UML, however, the expression (*Married & Man*), denoting the set of married men, would be illegal, since *Married* and *Man* are distinct subclasses of *Person*, and the constraint would have to be written:

> *Married*
> *Married.allInstances->select (oclIsTypeOf (Man)).spouse->forall(oclIsTypeOf (Woman)*

The primary motivation for Alloy's implicit typing is to match the intuitive semantics of graphical object models, in which boxes without parents in the diagram represent sets that are assumed to be disjoint, but which, unlike the sets denoted by the given types of Z, are not constant. It dispenses with the need to associate given types explicitly with these sets; allows declarations to include simple constraints; and makes type checking only marginally harder and no less effective. In Z, a set component cannot be used on the right-hand side of a declaration, so the declaration of *spouse* above would have to be split into a declaration (over some given type *PERSON*), and a separate constraint on the domain and range.

Inferring bounds in quantified formulas is convenient for the specifier, but has a deeper purpose: it ensures that Alloy specifications are *scope-monotonic*. As explained above (Section 4.3), an assignment is said to be within a scope of k if the denotation of each domain type is a set containing no more than k elements. A formula is scope-monotonic if, whenever it has a satisfying assignment in a scope k, it also has a satisfying assignment in scope $k+1$. This is guaranteed in Alloy by the omission of domain types from the language: the specification may only mention set components whose values are subsets of the domain types, so adding elements to a domain type can never invalidate an assignment. Scope monotonicity is crucial for a tool like ours that searches for instances within fixed scopes; without it, the bizarre situation can arise in which increasing the scope and thus considering more assignments results in fewer instances found.

7.4 Scalars as Sets

Alloy treats scalars as sets for two reasons: it allows a uniform notation for navigation expressions, and sidesteps the partial function problem. Suppose, given these declarations

> *CurrentPresident : President?*
> *predecessor : President? -> President?*
> *advisors : President -> Person*

we want to say that a president never shares advisors with her predecessor. In Alloy, we might write

```
no CurrentPresident.advisors & CurrentPresident.predecessor.advisors
```

In conventional Z style, the second navigation expression would be written

$$advisors\ (\{predecessor\ (CurrentPresident)\})$$

in which the function application is written differently from the relational image, and the result of the function application must be lifted to a set. The formula as a whole would require a guard in Z, since the subexpression

$$predecessor\ (CurrentPresident)$$

is undefined for the first president (who has no predecessor), so we would have to write

$$\begin{array}{l} CurrentPresident \in \mathrm{dom}\ predecessor \Rightarrow \\ \quad advisors\ (\{CurrentPresident\}) \cap advisors\ (\{predecessor\ (CurrentPresident)\}) = \emptyset \end{array}$$

Of course, one could adopt the style of Alloy in Z, and simply wrap all scalars in set brackets, but this is not how Z specifications are usually written.

The partial function problem is inescapable. In Alloy, the catch is that the formula

```
not (CurrentPresident.predecessor  in CurrentPresident.advisors)
```

does not have the meaning the reader intuitively expects: namely that presidents do not take advice from predecessors. It requires that every president have a predecessor, since otherwise, for the first president, the first expression will evaluate to the empty set, and the subformula will be vacuously true. Instead, we probably meant

```
not (CurrentPresident.predecessor  /in CurrentPresident.advisors)
```

which is short for

```
not (some CurrentPresident.predecessor && CurrentPresident.predecessor in CurrentPresident.advisors)
```

and equivalent to

```
no CurrentPresident.predecessor  || CurrentPresident.predecessor ! in CurrentPresident.advisors
```

For object modelling, this seems to be a smaller price than other approaches would exact. Here, in brief, is a review of the problem and some of the other solutions that have been proposed.

In all specification languages in which functions can be represented, a fundamental question arises: how to give meaning to the formula $f(x) = y$ in which the function f is applied to a value x that is outside its domain. This question has been the subject of debate for almost a century; it is connected with the old problem in logic of whether “The King of France is bald” is true if France has no king.

Specification languages have taken several approaches, each with its merits. One approach, taken by Larch [GHM90], and advocated by Gries and Schneider [GS95] and others, is to model all functions as if they were total. This approach is simple and clear, and preserves Leibniz’s law—that from $x = y$ one can infer $f(x) = f(y)$. But it is unsuitable for an object modelling language, since it makes an unnatural distinction between functions and relations, functions being required to carry with them the set representing the domain.

Another common approach is to introduce undefined values, for the result of an out-of-domain application, for the meaning of a formula containing such applications, or both. VDM [Jon90], for example, uses a three-valued logic; one unfortunate consequence is the loss of the law of the excluded middle. A subtle variant of this approach, adopted by Z, is to regard the meaning of some expressions and formulas to be undetermined. This also has problems: Leibniz's law goes, along with the ability to say for a given assignment whether a formula is true or false. Moreover, in practice it seems to confuse specifiers; Saaltink has analyzed a large body of published Z specifications and found almost all (including one of mine!) to be faulty in their use of partial functions [Saa97]. Even the innocuous-looking assertion $\#s = 3$, that the set s has three elements, is undefined in Z unless the type declaration of s says explicitly that s is finite.

A simpler approach is advocated by Parnas [Par95] and was adopted in Alloy's predecessor, NP. An elementary formula in which a function is applied outside its domain is deemed to evaluate to false. In this approach, we can infer from $f(x) = y$ that x is in the domain of f, so the formula

$$\exists\, x, y.\, f(x) = y$$

which is undefined in Z, has the expected meaning, namely that some element is mapped by f. Unfortunately, however, $f(x)=f(x)$ is sometimes false, and the specifier must know which formulas are regarded as elementary: if x is outside the domain of f, $f(x) \neq y$ will be true if $\neq$ is an elementary operator and false otherwise.

In Alloy, the classical laws hold; all expressions and formulas have a value for a given assignment; elementary formulas need no special treatment; there is no undefinedness and guards can generally be omitted. The one price paid, it seems, is the sometimes unexpected meaning of membership tests and the need for the strong equality and membership operators. (Incidentally, Martin Abadi has pointed out to me that confusing singleton sets and scalars in this way has a fine pedigree [Kan97].)

7.5 Mutability

Alloy takes the idea of mutability annotations from data modelling, but treats them as constraints that can be incorporated into operations. Formal specification languages such as Z do not typically regard such annotations as part of the state declaration, and only provide a notion of mutability for sets and relations as a whole (rather than with respect to individual atoms). UML has a variety of annotations that can be attached to relation ends. The *frozen* annotation appears to be similar to Alloy's *static* annotation, but the meaning of the construct and its relationship to OCL constraints is unclear.

Although we have been careful to give an abstract semantics to Alloy so that it can be used for specification, it can be used also to model code, in which case a more concrete intepretation is appropriate. Set components are interpreted as classes and relations as heap references. Static annotations can then be used to express some familiar notions. Consider a relation f from A to B, representing a field named f if type B in each A object, with multiplicity of zero-or-one on both sides. If f is right-static, the field is constant: once established on construction of an object it cannot be changed. If f is left-static, the B object cannot escape the A to which it is initially assigned: in other words, the B object is contained in the A object.

8 Related Work

The fundamental idea of object modelling—namely constructing models of state in which object identity plays a central role—originates in the semantic data models developed in the context of databases. One of the earliest and best known is Chen's entity-relationship model [Che76]; it was followed by more elaborate and sophisticated models, such as SDM [HM81] and DAPLEX [Shi81]. Abiteboul and Hull formalized many of the notions present in these models in their IFO model [AH87]. The key insight that a semantic data model is a better starting point for object-oriented development than a model in which objects have methods and fields is due to OMT [R+91].

Several recent object-oriented methods have used ideas from formal specification, most often Z, to bring rigour to and extend the expressiveness of object models. Like Alloy, Fusion [C+94] views the object model as an invariant on the state, and allows transitions to be expressed as global operations. It does not have a formal constraint language, and its object models have higher-order features (and are thus likely to be intractable). Syntropy [CD94] introduced the notion of navigation expressions, adopted later by UML, and although not formal, addresses carefully many issues treated superficially in UML (such as the meaning of aggregation). Catalysis [DW98] embodies a refined version of UML, with a cleaner syntax and a semantics based on sets. It demonstrates how operations and invariants can be used in a sophisticated setting in which specifications are composed from multiple views, and refined as progress towards implementation is made.

Several schemes have been devised to translate object models to formal specifications, both as an exercise to expose semantic issues, and as a means of obtaining an analyzable model. Larch is a popular target because its Shared Language provides a mathematical foundation that is not biased towards any style of specification structuring [BC95, BG98, HHK98]. Bickford and Guaspari's translation [BG98] addresses many of the complexities of UML, including subtyping and contexts. Their work also exposes a number of serious flaws in the definition of UML—for example, that its type system admits Russell's paradox.

Z is another popular target, because its built-in datatypes—sets and relations—are well-suited to object modelling. Larch's algebraic operators, in contrast, are not a natural fit, and translation is rather indirect. UML classes are usually represented in Z with schema types [FBL97, FBLS97], following the object-oriented style of [Hall90]. More recently, work has begun on a formalization of UML in which semantic functions are defined in Z [AC98].

Alloy's syntax has some similarities to an Ascii notation for mathematics developed by Richard Botting [Bot92].

9 Future Work

Alloy has several deficiencies that we plan to correct:

- Operations are often tedious to write because frame conditions must be made explicit. We are considering providing a switch in our tool that will add, implicitly, constraints that components not mentioned are unchanged; what mentioned or unchanged should mean is tricky. We are also investigating the addition of an imperative subset so that an operation can be written as a sequence of assignments but interpreted semantically as a constraint. This would also allow operations previously specified to be composed sequentially.

- Sequences are not available. We hope to build a notion of sequences on top of Alloy's indexed relations, so that one can specify a function that maps an atom to a sequence of atoms, and make use of standard operators such as concatenation.
- Alloy's structuring mechanisms are limited. It lacks the generic schemas of Z, so the modeller cannot define polymorphic operators or constraints (eg, that a relation forms a tree). Also, it has no features for composing models (see for example, the notion of views in [Jac95] or frameworks in [DW98]).

Acknowledgments

Many thanks to Anthony Hall, Michael Jackson, and Barbara Liskov for their insightful comments and suggestions during the development of Alloy; to Albert Lee, Ian Schechter, Ilya Shlyakhter and Benjamin Self, Alloy's first users; to Allison Waingold, who participated in the design of the graphical notation.

This research was funded in part by the National Science Foundation (under grant CCR-9523972), and by the MIT Center for Innovation in Product Development (under NSF Co-operative Agreement Number EEC-9529140).

Visio Corporation helped us develop a Visio template for Alloy (available from the author), which was used to draw the diagrams in this paper, and donated their software to the author's software engineering class in which Alloy was used.

References

[AC98] A. S. Evans and A.N.Clark. Foundations of the unified modeling language. In *2nd Northern Formal Methods Workshop*, Ilkley, Electronic Workshops in Computing. Springer-Verlag, 1998.

[AH87] Serge Abiteboul and Richard Hull. IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems*,Vol. 12, No. 4, December 1987, pp. 525-565.

[AW96] Gerald Allwein and Jon Barwise, eds. *Logical Reasoning with Diagrams*. Oxford University Press, 1996.

[BC95] Robert H. Bourdeau and Betty H.C. Cheng. A Formal Semantics for Object Model Diagrams. *IEEE Transactions on Software Engineering*, October 1995.

[BG98] Mark Bickford and David Guaspari. *Lightweight Analysis of UML*. TM-98-0036, Odyssey Research Associates, Ithaca, NY, November 1998.

[Bot92] Richard Botting. *Maths in Ascii*. Poster. Joint Meeting of Southern California Chapter of Mathematical Association of America and Society for Industrial and Applied Mathematics (SIAM). Spring 1992.
Text available at: http://www.csci.csusb.edu/dick/papers/rjb92b.discrete.

[C+94] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes and Paul Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[CD94] Steve Cook and John Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice Hall, 1994.

[Che76] Peter P. Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, Vol. 1, No. 1, (1976), pp. 9–36.

[DR99] Elisabetta Di Nitto and David Rosenblum. Exploiting Architecture Description Languages to Specify Architectural Styles Induced by Middleware Infrastructures. Proc. 21st International Conference on Software Engineering. Los Angeles, CA, May 1999, pp. 13-22.

[DW98] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks With Uml : The Catalysis Approach*. Addison-Wesley, 1998.

[FBL97] Robert B. France, Jean-Michel Bruel and Maria M. Larrondo-Petrie. An Integrated Object-Oriented and Formal Modeling Environment. *Journal of Object Oriented Programming* (JOOP), 10(7), November/December 1997.

[FBLS97] Robert B. France, Jean-Michel Bruel, Maria M. Larrondo-Petrie, and Malcolm Shroff. Exploring the Semantics of UML Type Structures with Z. *Proceedings of the Formal Methods for Open Object-based Distributed Systems* (FMOODS'97), 1997.

[Fer92] Eugene S Ferguson. *Engineering and the Mind's Eye*. MIT Press, 1992.

[GHM90] John V. Guttag, James J. Horning, and Andres Modet. *Report on the Larch Shared Language: Version 2.3.* Technical Report 58, Compaq Systems Research Center, Palo Alto , CA, 1990.

[GN91] David Garlan and David Notkin. Formalizing Design Spaces: Implicit Invocation Mechanisms. *VDM '91: Formal Software Development Methods*, pp. 31–44 (October 1991). Appears as Springer-Verlag *Lecture Notes in Computer Science 551*.

[GS95] David Gries and Fred B. Schneider. *Avoiding the Undefined by Underspecification.* In Jan van Leeuwen (ed.), *Computer Science Today: Recent Trends and Developments*, pages 366–373. Volume 1000, Lecture Notes in Computer Science, Springer-Verlag, NY, 1995.

[Hall90] Anthony Hall. Using Z as a Specification Calculus for Object-Oriented Systems. In D. Bjorner, C.A.R. Hoare, and H. Langmaack, eds., *VDM and Z: Formal Methods in Software Development*, Lecture Notes in Computer Science, Volume 428, pp. 290–381, Springer-Verlag, New York, 1990.

[Har88] David Harel. On visual formalisms. *Communications of the ACM*, Vol. 31, No. 5, pp. 514-530, 1988.

[Hay93] Ian Hayes. *Specification Case Studies*. Prentice Hall, 1993.

[HHK98] Ali Hamie, John Howse and Stuart Kent. Interpreting the Object Constraint Language. *Proceedings of Asia Pacific Conference in Software Engineering*, IEEE Press, 1998.

[HM81] Michael Hammer and Dennis McLeod. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, Vol. 6, No. 2, June 1981, pp. 351–386.

[Iri99] Joseph B. Irineo. *An Object-Oriented, Maximum-Likelihood Parameter Estimation Program for GARCH(p,q)*. Masters Thesis, Department of Electrical Engineering and Computer Science. Massachusetts Institute of Technology, June 1999.

[Jac95] Daniel Jackson. Structuring Z Specifications with Views. *ACM Transactions on Software Engineering and Methodology*, Vol. 4, No. 4, October 1995, pp. 365–389.

[Jac97] Daniel Jackson. Automatic Analysis of Architectural Style. Unpublished manuscript. August 1997.

[Jac98] Daniel Jackson. An Intermediate Design Language and its Analysis. *Proceedings of ACM SIGSOFT Foundations of Software Engineering*, Orlando, Florida, 1998.

[JC99] Daniel Jackson and John Chapin. *Simplifying Air-traffic Control: An Exercise in Software Design.* Submitted for publication, July 1999.

[JD96a] Daniel Jackson and Craig A. Damon. *Nitpick Reference Manual.* Technical Report CMU-CS-96-109. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1996.

[JD96b] Daniel Jackson and Craig A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering*, Vol. 22, No. 7, July 1996, pp. 484–495.

[JJ94] Daniel Jackson and Michael Jackson. Problem Decomposition for Reuse. *Software Engineering Journal*, Vol. 11, No. 1, January 1996, pp. 19–30.

[Jon90] Cliff Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.

[JW99] Daniel Jackson and Allison Waingold. Lightweight Extraction of Object Models from Bytecode. *Proc. International Conference on Software Engineering*, Los Angeles, CA, May 1999.

[JZ98] Michael Jackson and Pamela Zave. Distributed Feature Composition: A Virtual Architecture for Telecommunications Services. *IEEE Transactions on Software Engineering,* XXIV(10): 831–847, October 1998.

[Kan97] Akihiro Kanamori. The mathematical import of Zermelo's well-ordering theorem. *Bulletin of Symbolic Logic*, Volume 3, Issue 3, September 1997, pages 281–311.

[Lee99] SeungYong Albert Lee. *Object Modeling Applied to CTAS*. Masters Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[M+99] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. *The Click Modular Router*. MIT Laboratory for Computer Science, Cambridge, MA. Submitted for publication, June 1999.

[Met96] Daniel Le Metayer. Software Architecture Styles as Graph Grammars. *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Software Engineering Notes, Vol. 21, No. 6, ACM Press, October 1996, pp. 3–14.

[MJa95] Michael Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*. “Definitions”, pp. 51–54. Addison-Wesley, 1995.

[MK96] Jeff Magee and Jeff Kramer. Dynamic Structure in Software Architectures. *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Software Engineering Notes, Vol. 21, No. 6, ACM Press, October 1996, pp. 3–14.

[Mon98] Robert T. Monroe. *Capturing Software Architecture Design Expertise With Armani.* Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University, October 1998.

[Mor93] Carroll Morgan. Telephone Network. Chapter 3 (pp. 31–42) of [Hay93].

[Par95] David Parnas. A Logic for Describing, not Verifying, Software. *Erkenntnis* (Kluwer), Volume 43, No. 3, November 1995, pp. 321–338.

[R+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Rat97] Rational Inc. *The Unified Modeling Language*. see http://www.rational.com.

[RJB99] James Rumbaugh, Ivar Jacobson and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[Saa97] Mark Saaltink. Domain Checking Z Specifications. *4th NASA LaRC Formal Methods Workshop*, September 1997.

[Shi81] David W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, Vol. 6, No. 1, March 1981, pp. 140–173.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual*. Second edition, Prentice Hall, 1992.

[SSM97] K.J. Sullivan, J. Socha and M. Marchukov. Using Formal Methods to Reason about Architectural Standards. *Proceedings of the International Conference on Software Engineering* (ICSE97), Boston, Massachusetts, May 1997.

[TG87] Alfred Tarski and Steven Givant. *A Formalization of Set Theory Without Variables*. American Mathematical Society Colloquium Publications, Volume 41, 1987.

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

Appendix A: Alloy Syntax

Vertical bar denotes choice; star denotes zero or more repetitions; angle brackets indicate optional phrases; _x,_ indicates a comma-separated lists of _x_'s. _STAR_, _BAR_ and _PRIME_ are terminals representing an asterisk, vertical bar and prime mark respectively.

model ::= **model** _model-name_ **{ domain {**_domdecl, *_**}** _para*_**}**
para ::= **state** ⟨_name_⟩ **{** _compdecls_ **}**
| **inv** ⟨_name_⟩ **{** _formula*_ **}**
| **def** _comp_ **{** _formula*_ **}**
| **cond** _name_ ⟨_arglist_⟩ **{** _formula*_ **}**
| **assert** ⟨_name_⟩ **{** _formula*_**}**
| **op** _name_ ⟨_arglist_⟩ **{** _formula*_**}**

domdecl ::= ⟨**fixed**⟩ _set_
compdecls ::= _setdecl_ | _reldecl_ | _setdecl_ ⟨,⟩ _compdecls_ | _reldecl_ ⟨,⟩ _compdecls_
setdecl ::= ⟨**disjoint** | **partition**⟩ _set,_ **:** ⟨**fixed** | **static**⟩ _set mult_
reldecl ::= _relx_ ⟨**(**~ _relx_**)**⟩ **,** **:** ⟨**static**⟩ _set mult_ **->** ⟨**static**⟩ _set mult_
relx ::= _rel_ | _rel_ **[**_set_**]**
mult ::= **?** | **!** | **+**
comp ::= _set_ | _rel_

arglist ::= **(** _argdecl,_ **)**
argdecl ::= _arg,_ **:** _set mult_

formula ::= _negate formula_
| _formula logic-op formula_
| _quantifier var-decl_ **,** _**BAR**_ _formula_
| _expr comp-op expr_
| _quantifier expr_
| **(** _formula_ **)**
| _name_ ⟨**(** _expr_ **,** **)**⟩

var-decl ::= _var_ **,** ⟨**:** _expr_⟩

logic-op ::= **&&** | **||** | **->** | **<->**
negate **::=** **not** | **!**
comp-op ::= **in** | **=** | _negate_ **in** | _negate_ **=** | **/=** | **/in**
quantifier ::= **all** | **some** | **no** | **sole** | **one**

expr ::= _var_ | _arg_ | _set_ | _expr expr-op expr_ | _expr_ **.** _qualifier_ | **{** _var-decl_ _**BAR**_ _formula_ **}** | **(** _expr_ **)**
expr-op ::= **+** | **-** | **&**
qualifier ::= _rel_ | _rel_ **[** _var_ **]** | **~** _qualifier_ | **+** _qualifier_ | _**STAR**_ _qualifier_

arg ::= _id_
var ::= _id_
name ::= _id_
set ::= _id_ | _id_ _**PRIME**_
rel ::= _id_ | _id_ _**PRIME**_

Appendix B: Shorthands

declaration	*implicit condition*	*comment*
s : t	*s in t*	
disjoint a, b, c : t	*no (a & (b+c))*	
	no (b & (a + c))	
	no (c & (a + b))	
partition a, b, c : t	*a + b + c = t*	*in addition to disjointness*
r : s -> t	*all x : Ty(s) | x.r in t*	*right atoms drawn from t*
	all x : Ty(t) | x.~r in s	*left atoms drawn from s*
domain {fixed d}	*d = d’*	
s : static t	*all x : t & t’ | x in s <-> x in s’*	
s : fixed t	*s = s’*	
r : s -> static t	*all x : s & s’ | x. r = x. r’*	
r : static s -> t	*all x : t & t’ | x.~r = x.~r’*	
s : t !	*one s*	*s is scalar*
s : t ?	*sole s*	*s is optional*
s : t +	*some s*	*s is non-empty*
r : s -> t ?	*all a: s | sole a.r*	*r is a partial function*
r : s -> t !	*all a: s | one a.r*	*r is a total function*
r : s -> t +	*all a: s |some a.r*	*r is total*
r : s ? -> t	*all a: t | sole a.~r*	*r is injective*
r : s ! -> t	*all a: t | one a.~r*	*r is injective and surjective*
r : s+ -> t	*all a: t |some a.~r*	*r is surjective*

syntactic sugar	*kernel translation*	*comment*
f || g	*not (not f && not g)*	*disjunction*
f -> g	*not (f && not g)*	*implication*
some v : T | f	*not all v : T | not f*	*existential quantification*
d = e	*d in e && e in d*	*equality*
d != e	*not d = e*	
d /in e	*some d && d in e*	
d /= e	*some d && d = e*	
*e.*q.s*	*(e.+q) + e).s*	*reflexive transitive closure*
no v : T | f	*all v : T | not f*	
all decls v | f	*all d0 ; d1 ; d2 ; … | f*	
Q decls1 v decls2 | f	*Q decls1 v : d decls2 | f*	*Q is any quantifier*
some decls | f	*not all decls | not f*	
no decls | f	*not some decls | f*	
one decls | f	*some decls | f && sole decls | f*	
sole v0 : e0 ; v1 : e1 ; … | f	*all v01, v02 : e0 ; v11, v12 : e1 ; … |*	
	(some v0, v1, … | f && v0 = v01 && v1 = v11 && …) &&	
	(some v0, v1, … | f && v0 = v02 && v1 = v12 && …)	

	-> v01 = v02 && v11 = v12 && …	
all v0 : e0 ; v1 : e1 ; … | f	*all v0 : e0 | all v1 : e1 | … | f*	
all v : e | f	*all v : D | v in e -> f*	*D is domain type of expression e*
{ v | f}	*{ v : d | f}*	*d is domain corresponding to domain type D of v*
{v : e | f}	*{v : D | v in e && f}*	*D is domain type of expression e*
some e	*some x | x in e*	*e has some elements*
no e	*no x | x in e*	*e has no elements*
all e	*all x | x in e*	*e has every element (in its domain)*
sole e	*sole x | x in e*	*e has at most one element*
one e	*one x | x in e*	*e has one element*

Notes

1 Declaration constraints apply cumulatively. For example, the constraints implicit in

r : static s ? -> static t !

are obtained by conjoining those given by

r : static s -> t
r : s -> static t
r : s? -> t
r : s -> t!

2 The declaration constraints for an indexed relation are obtained by quantifying over each relation value. For example, for the declaration

r [X] : s -> t ?

the constraint is

all x : X | all a: s | sole a.r[x]