A Micromodularity Mechanism

Daniel Jackson, Ilya Shlyakhter and Manu Sridharan
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA
dnj@mit.edu

Abstract

A simple mechanism for structuring specifications is described. By modelling structures as atoms, it remains entirely first-order and thus amenable to automatic analysis. And by interpreting fields of structures as relations, it allows the same relational operators used in the formula language to be used for dereferencing. An extension feature allows structures to be developed incrementally, but requires no textual inclusion nor any notion of subtyping. The paper demonstrates the flexibility of the mechanism by application in a variety of common idioms.

Categories and Subject Descriptors

D.2.1 Requirements/Specifications—Languages; D.2.4 Software/Program Verification—Formal methods, Model checking; F.3.1 Specifying and Verifying and Reasoning about Programs—Assertions, Invariants, Specification techniques.

General Terms

Design; Documentation; Languages; Verification.

Keywords

Modeling languages; formal specification; first-order logic; relational calculus; Alloy language; Z specification language; schema calculus.

Introduction

I am neither crazy nor a micromaniac.
(A micromaniac is someone obsessed with
reducing things to their smallest possible form.
This word, by the way, is not in the dictionary.)
–Edouard de Pomiane, French Cooking in Ten Minutes, 1930

Most specification languages provide mechanisms that allow larger specifications to be built from smaller ones. These mechanisms are often the most complicated part of the language, and present obstacles to analysis. This paper presents a simple mechanism that seems to be expressive enough for a wide variety of uses, without compromising analyzability.

To appear, Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9), Vienna, Austria, Sept. 10-14, 2001

This work is part of a larger project investigating the design of a “micro modelling language”. Our premise is that lightweight application of formal methods [6] demands an unusually small and simple language that is amenable to fully automatic semantic analysis. The Alloy language is the result to date of our efforts to design such a language. Based on our experiences with the language [4] and its analyzer [5], we have recently developed a revision of Alloy that overcomes many of its limitations. This paper describes the key feature of the revised language: the *signature*, a new modularity mechanism.

The mechanism allows our existing analysis scheme [3] to be applied to specifications involving structures. This is not achieved by treating the structuring mechanism as a syntactic sugar, which would limit the power of the notation (ruling out, for example, quantification over structures) and would complicate the analysis tool and make output harder for users to interpret. Because of the mechanism’s generality, it has also enabled us to simplify the language as a whole, making it more uniform and eliminating some ad hoc elements.

Our mechanism has a variety of applications. It can express inherent structure in the system being modelled, and can be used to organize a specification in which details are added incrementally. It can be used to construct a library of datatypes, or to describe a system as an instantiation of a more general system. And it can express state invariants, transitions, and sequences, despite the lack of any special syntax for state machines.

In this last respect, the new language differs most markedly from its predecessor [4], which provided built-in notions of state invariants and operations. We now think this was a bad idea, because it made the language cumbersome for problems (such as the analysis of security policies or architectural topology constraints) in which temporal behaviour can be fruitfully ignored, and too inflexible for many problems in which temporal behaviour is important.

Our paper begins by explaining our motivations—the requirements our mechanism is designed to meet. The mechanism is then presented first informally in a series of examples, and then slightly more rigorously feature-by-feature. We discuss related work, especially the schema calculus of Z, and close with a summary of the merits and deficiences of our notation as a whole.

1 Requirements

The goal of this work was to find a single structuring mechanism that would support a variety of common specification idioms:

- *States*: description of complex state as a collection of named components; incremental description both by hierarchy, in which a complex state becomes a component of a larger state, and by extension, in which new components are added; declaration of invariants and definitions of derived components;
- *Datatypes*: separate description of a library of polymorphic datatypes, such as lists, sequences, trees and orders, along with their operators;
- *Transitions*: specification of state transitions as operations described implicitly as formulas relating pre- and post-state; composition of operations from previously defined invariants and operations; sequential composition of operations; description of traces as sequences of states;
- *Abstractions*: description of abstraction relations between state spaces;
- *Assertions*: expression of properties intended to be redundant, to be checked by analysis, including: relationships amongst invariants; wellformedness of definitions (eg, that an implicit definition is functional); establishment and preservation of invariants by operations; properties of states reachable along finite traces; and simulation relationships between abstract and concrete versions of an operation.

We wanted additionally to meet some more general criteria:

- *Simplicity*. The language as a whole should be exceptionally small and simple.
- *Flexibility*. Support for the particular idioms of state-machine specification should not be a straitjacket; the language should not dictate how state machines are expressed, and should not make it hard to describe structures that are not state machines (such as security models and architectural styles).
- *Analyzability*. A fully automatic semantic analysis should be possible. In the present work, this has been achieved by requiring that the modularity mechanism be first order, and expressible in the kernel of the existing language.

Finally, our language design decisions have been influenced by some principles that we believe contribute to these goals, make the language easier to use, and analysis tools easier to build:

- *Explicitness*. The language should be fully explicit, with as few implicit constraints, coercions, etc, as possible.
- *Minimal mathematics*. The basic theory of sets and relations should suffice; it should not be necessary to introduce domains, fixed points, infinities or special logical values.
- *Minimal syntax*. There should be very few keywords or special symbols, and no need for special typography or layout.
- *Uniformity*. A small and general set of constructs should be applied uniformly, independent of context.
- *Lack of novelty*. Whenever possible, notions and syntax should follow standard usage of conventional mathematics and programming.

2 Informal Description

As a running example, we will specify a simple memory system involving a cache and a main memory. The memory has a fixed set of addresses and associates a data value with each address. The cache, in contrast, associates data values with some subset of addresses that varies over time. The cache is updated by a "write-back scheme", which means that updates need not be reflected to main memory immediately. The cache may therefore hold a more current value for an address than the main memory; the two are brought into alignment when the address is flushed from the cache and its value is written to main memory.

2.1 States

We start by declaring the existence of addresses and data values:

```
sig Addr {}
sig Data {}
```

Each line declares a *signature*, and introduces a set of atoms: *Addr* for the set of addresses, and *Data* for the set of data values. Like 'given types' in Z, these sets are disjoint from one another, and their atoms are unstructured and uninterpreted. Signature names can be used as expressions denoting sets, but they are also treated as types, so the expression *Addr+Data*, for example, is ill-typed, since the union operator (+) requires the types of its operands to match.

The signature declaration

```
sig Memory {
    addrs: set Addr,
    map: addrs ->! Data
    }
```

likewise declares a set of atoms, *Memory*, corresponding to the set of all possible memories. In addition, it declares two fields: *addrs* and *map* which associate with a memory a set of addresses and a mapping from addresses to data values respectively. Thus, given a memory *m*, the expression *m.addrs* will be a set of addresses, *m.map* will be a relation from addresses to data values. The memory, addresses and data values should be viewed as distinct atoms in their own right; fields don't decompose an atom, but rather relate one atom to others. The exclamation mark in the declaration of the field *map* is a 'multiplicity marking': it says that *m.map* associates exactly one data value with each address in the set *m.addrs*. The use of *addrs* rather than *Addr* on the left side of the arrow indicates that *m.map* does not associate a data value with an address that is not in the set *m.addrs*.

In these expressions, the dot is simply relational image. More precisely, when we say that *m* is a memory, we mean that the expression *m* denotes a set consisting of a single atom. The field *addrs* is a relation from *Memory* to *Addr*, and *m.addrs* denotes the image of the singleton set under this relation. So for a set of memories *ms*, the expression *ms.addrs* will denote the union of the sets of addresses that belong to the individual memories. Given an address *a*, the expression *a.(m.map)* denotes the set of data values associated with address *a* in memory *m*, which will either be empty (when the address is not mapped) or a singleton. For convenience, we allow the relational image *s.r* to be written equivalently as *r*[*s*], where [] binds more loosely than dot, so this expression may be written as *m.map*[*a*] instead.

Like objects of an object-oriented language, two distinct atoms can have fields of the same value. Unlike objects, however, atoms are immutable. Each field is fixed, and cannot map an atom to one value at one time and another value at another time. To describe an operation that changes the state of a memory, therefore, we will use two distinct atoms in the set *Memory* to represent the memory's state before and after.

2.2 Extension

A signature declaration can introduce a set as a subset of one previously declared, in which case we call it a *subsignature*. In this case, the set does not correspond to a type, but rather its atoms take on the type of the superset. For example, the declaration

```
sig MainMemory extends Memory {}
```

introduces a set of atoms *MainMemory* representing main memories, which is constrained to be a subset of the set *Memory*. Likewise

```
sig Cache extends Memory {
    dirty: set addrs
    }
```

introduces a set of atoms *Cache* representing those memories that can be regarded as caches. It also introduces a field *dirty* that associates with a cache the set of addresses that is dirty; later, we will use this to represent those addresses for which a cache and main memory differ. Because *Cache* is a subset of *Memory*, and *m.addrs* (for any memory *m*) is a subset of *Addr*, the field denotes a relation whose type is from *Memory* to *Addr*. Expressions such as *m.dirty* are therefore type-correct for a memory *m*, whether or not *m* is a cache. But since declaration of the field *dirty* within the signature *Cache* constrains *dirty* to be a relation that maps only caches, *m.dirty* will always denote the empty set when *m* is not a cache.

This approach avoids introducing a notion of subtyping. Subtypes complicate the language, and tend to make it more difficult to use. In OCL [17], which models extension with subtypes rather than subsets, an expression such as *m.dirty* would be illegal, and would require a coercion of *m* to the subtype *Cache*. Coercions do not fit smoothly into the relational framework; they interfere with the ability to take the image of a set under a relation, for example. Moreover, subtypes are generally disjoint, whereas our approach allows the sets denoted by subsignatures to overlap. In this case, we'll add a constraint (in Section 2.4 below) to ensure that *MainMemory* and *Cache* are in fact disjoint.

Declaring *Cache* and *MainMemory* as subsignatures of *Memory* serves to factor out their common properties. Extension can be used for a different purpose, in which a single signature is developed by repeated extensions along a chain. In this case, the supersignatures may not correspond to entities in the domain being modelled, but are simply artifacts of specification—fragments developed along the way. Z specifications are typically developed in this style.

2.3 Hierarchy

The signature declaration also supports hierarchical structuring. We can declare a signature for systems each consisting of a cache and a main memory:

```
sig System {
    cache: Cache,
    main: MainMemory
    }
```

Again, *System* introduces a set of atoms, and each field represents a relation. The omission of the keyword *set* indicates that a relation is a total function. So for a system *s*, the expression *s.cache* denotes one cache—that is, a set consisting of a single cache. This is one of very few instances of implicit constraints in our language, which we introduced in order to make declaration syntax conventional.

Since signatures denote sets of atoms, apparently circular references are allowed. Linked lists, for example, may be modelled like this, exactly as they might be implemented in a language like Java:

```
sig List {}
sig NonEmptyList extends List {elt: Elt, rest: List}
```

There is no recursion here; the field *rest* is simply a homogeneous relation of type *List* to *List*, with its domain restricted to the subset *NonEmptyList*.

2.4 State Properties

Properties of signature atoms are recorded as logical formulas. To indicate that such a property always holds, we package it as a *fact*. To say that, for any memory system, the addresses in a cache are always addresses within the main memory, we might write:

```
fact {all s: System | s.cache.addrs in s.main.addrs}
```

or, using a shorthand that allows facts about atoms of a signature to be appended to it:

```
sig System {cache: Cache, main: MainMemory}
    {cache.addrs in main.addrs}
```

The appended fact is implicitly prefixed by

```
all this: System | with this |
```

in which the *with* construct, explained in Section 3.6 below, causes the fields implicitly to be dereferences of the atom *this*.

A fact can constrain atoms of arbitrary signatures; to say that no main memory is a cache we might write:

```
fact {no (MainMemory & Cache)}
```

where *no e* means that the expression *e* has no elements, and & is intersection.

Most descriptions have more interesting facts. We can express the fact that linked lists are acyclic, for example:

```
fact {no p: List | p in p.^ rest}
```

The expression ^ *rest* denotes the transitive closure of the relation *rest*, so that *p.^rest* denotes the set of lists reachable from *p* by following the field *rest* once or more. This illustrates a benefit of treating a field as a relation—that we can apply standard relational operators to it—and is also an example of an expression hard to write in a language that treats extension as subtyping (since each application of *rest* would require its own coercion).

Often we want to define a property without imposing it as a permanent constraint. In that case, we declare it as a *function*. Here, for example, is the invariant that the cache lines not marked as dirty are consistent with main memory:

```
fun DirtyInv (s: System) {
    all a !: s.cache.dirty | s.cache.map[a] = s.main.map[a]
    }
```

(The exclamation mark negates an operator, so the quantification is over all addresses that are *not* dirty.) Packaging this as a function that can be applied to a particular system, rather than as a fact for all systems, will allow us to express assertions about preservation of the invariant (Section 2.8).

By default, a function returns a boolean value—the value of the formula in its body. The value of *DirtyInv(s)* for a system *s* is therefore true or false. A function may return non-boolean values. We might, for example, define the set of bad addresses to be those for which the cache and main memory differ:

```
fun BadAddrs (s: System): set Addr {
    result = {a: Addr | s.cache.map[a] != s.main.map[a]}
    }
```

and then write our invariant like this:

```
fun DirtyInv (s: System) {BadAddrs(s) in s.cache.dirty}
```

In this case, *BadAddrs(s)* denotes a set of addresses, and is short for the expression on the right-hand side of the equality in the definition of the function *BadAddrs*. The use of the function application as an expression does not in fact depend on the function being defined explicitly. Had we written

```
fun BadAddrs (s: System): set Addr {
    all a: Addr | a in result iff s.cache.map[a] != s.main.map[a]
    }
```

the application would still be legal; details are explained in Section 3.7.

2.5 Operations

Following Z, we can specify operations as formulas that constrain pre- and post-states. An operation may be packaged as a single function (or as two functions if we want to separate pre- and post-conditions in the style of VDM or Larch).

The action of writing a data value to an address in memory might be specified like this:

```
fun Write (m,m': Memory, d: Data, a: Addr) {
    m'.map = m.map ++ (a->d)
    }
```

The formula in the body of the function relates *m*, the value of the memory before, to *m'*, the value after. These identifers are just formal arguments, so the choice of names is not significant. Moreover, the prime mark plays no special role akin to decoration in Z—it's a character like any other. The operator ++ is relational override, and the arrow forms a cross product. As mentioned above, scalars are represented as singleton sets, so there is no distinction between a tuple and a relation. The arrows in the expressions *a->d* here and *addrs->Data* in the declaration of the *map* field of *Memory* are one and the same.

The action of reading a data value can likewise be specified as a function, although since it has no side-effect we omit the *m'* parameter:

```
fun Read (m: Memory, d: Data, a: Addr) {
    d = m.map[a]
    }
```

Actions on the system as a whole can be specified using these primitive operations; in Z, this idiom is called 'promotion'. A read on the system is equivalent to reading the cache:

```
fun SystemRead (s: System, d: Data, a: Addr) {
    Read (s.cache, d, a)
    }
```

The *Read* operation has an implicit precondition. Since the data parameter *d* is constrained (implicitly by its declaration) to be scalar—that is, a singleton set—the relation *m.map* must include a mapping for the address parameter *a*, since otherwise the expression *m.map[a]* will evaluate to the empty set, and the formula will not be satisfiable. This precondition is inherited by *SystemRead*. If the address *a* is not in the cache, the operation cannot proceed, and it will be necessary first to load the data from main memory. It is convenient to specify this action as a distinct operation:

```
fun Load (s,s': System, a: Addr) {
    a !in s.cache.addrs
    s'.cache.map = s.cache.map + (a->s.main.map[a])
    s'.main = s.main
    }
```

The + operator is just set union (in this case, of two binary relations, the second consisting of a single tuple). A write on the system involves a write to the cache, and setting the dirty bit. Again, this can be specified using a primitive memory operation:

```
fun SystemWrite (s,s': System, d: Data, a: Addr) {
    Write (s.cache, s'.cache, d, a)
    s'.cache.dirty = s.cache.dirty + a
    s'.main = s.main
    }
```

A cache has much smaller capacity than main memory, so it will occasionally be necessary (prior to loading or writing) to flush lines from the cache back to main memory. We specify flushing as a non-deterministic operation that picks some subset of the cache addrs and writes them back to main memory:

```
fun Flush (s,s': System) {
    some x: set s.cache.addrs {
        s'.cache.map = s'.cache.map - (x->Data)
        s'.cache.dirty = s.cache.dirty - x
        s'.main.map = s.main.map ++
            {a: x, d: Data | d = s.cache.map[a]}
        }
    }
```

The - operator is set difference; note that it is applied to sets of addresses (in the third line) and to binary relations (in the second). The comprehension expression creates a relation of pairs *a->d* satisfying the condition.

Finally, it is often useful to specify the initial conditions of a system. To say that the cache initially has no addresses, we might write a function imposing this condition on a memory system:

```
fun Init (s: System) {no s.cache.addrs}
```

2.6 Traces

To support analyses of behaviours consisting of sequences of states, we declare two signatures, for ticks of a clock and traces of states:

```
sig Tick {}
sig SystemTrace {
    ticks: set Tick,
    first, last: ticks,
    next: (ticks - last) !->! (ticks - first)
    state: ticks ->! System}
    {
    first.*next = ticks
    Init (first.state)
    all t: ticks - last |
        some s = t.state, s' = t.next.state |
            Flush (s,s')
            || (some a: Addr | Load (s,s',a))
            || (some d: Data, a: Addr | SystemWrite (s,s',d,a))
    }
```

Each trace consists of a set of *ticks*, a *first* and *last* tick, an ordering relation *next* (whose declaration makes it a bijection from all ticks except the last to all ticks except the first), and a relation *state* that maps each tick to a system state.

The fact appended to the signature states first a generic property of traces: that the ticks of a trace are those reachable from the first tick. It then imposes the constraints of the operations on the states in the trace. The initial condition is required to hold in the first state. Any subsequent pair of states is constrained to be related by one of the three side-effecting operations. The existential quantifier plays the role of a *let* binding, allowing *s* and *s'* in place of *t.state* and *t.next.state*, representing the state for tick *t* and the state for its successor *t.next*. Note that this formulation precludes stuttering; we could admit it simply by adding the disjunct *s=s'* allowing a transition that corresponds to no operation occurring.

Bear in mind that this fact is a constraint on all atoms in the set *SystemTrace*. As a free standing fact, the second line of the fact—the initial condition— would have been written:

```
fact {all x: SystemTrace | Init ((x.first).(x.state))}
```

2.7 Abstraction

Abstraction relationships are easily expressed using our function syntax. To show that our memory system refines a simple memory without a cache, we define an abstraction function *Alpha* saying that a system corresponds to a memory that is like the system's memory, overwritten by the entries of the system's cache:

```
fun Alpha (s: System, m: Memory) {
    m.map = s.main.map ++ s.cache.map
    }
```

As another example, if our linked list were to represent a set, we might define the set corresponding to a given list as that containing the elements reachable from the start:

```
fun ListAlpha (p: List, s: set Elt) {
    s = p.*rest.elt
    }
```

2.8 Assertions

Theorems about a specification are packaged as *assertions*. An assertion is simply a formula that is intended to hold. A tool can check an assertion by searching for a counterexample—that is, a model of the formula's negation.

The simplest kinds of assertion record consequences of state properties. For example,

```
assert {
    all s: System | DirtyInv (s) && no s.cache.dirty
        => s.cache.map in s.main.map
    }
```

asserts that if the dirtiness invariant holds,and there are no dirty addresses, then the mapping of addresses to data in the cache is a subset of the mapping in the main memory.

An assertion can express consequences of operations. For example,

```
assert {
    all s: System, d: Data, a: Addr |
        SystemRead (s,d,a) => a in s.cache.addrs
    }
```

embodies the claim made above that *SystemRead* has an implicit precondition; it asserts that whenever *SystemRead* occurs for an address, that address must be in the cache beforehand. An assertion can likewise identify a consequence in the post-state; this assertion

```
assert {
    all s,s': System, d: Data, a: Addr |
        SystemWrite (s,s',d,a) => s'.cache.map[a] = d
    }
```

says that after a *SystemWrite*, the data value appears in the cache at the given address.

Preservation of an invariant by an operation is easily recorded as an assertion. To check that our dirtiness invariant is preserved when writes occur, we would assert

```
assert {
    all s,s': System, d: Data, a: Addr |
        SystemWrite (s,s',d,a) && DirtyInv (s) => DirtyInv (s')
    }
```

Invariant preservation is not the only consequence of an operation that we would like to check that relates pre- and post-states. We might, for example, want to check that operations on the memory system do not change the set of addresses of the main memory. For the *Flush* operation, for example, the assertion would be

```
assert {
    all s,s': System | Flush(s,s') => s.main.addrs = s'.main.addrs
    }
```

which holds only because the cache addresses are guaranteed to be a subset of the main memory addresses (by the fact associated with the *System* signature).

The effect of a sequence of operations can be expressed by quantifying appropriately over states. For example,

```
assert {
    all s, s': System, a: Addr, d,d': Data |
        SystemWrite (s,s',d,a) && SystemRead (s',d',a) => d = d'
    }
```

says that when a write is followed by a read of the same address, the read returns the data value just written.

To check that a property holds for all reachable states, we can assert that the property is an invariant of every operation, and is established by the initial condition. This strategy can be shown (by induction) to be sound, but it is not complete. A property may hold for all reachable states, but may not be preserved because an operation breaks the property when executed in a state that happens not to be reachable.

Traces overcome this incompleteness. Suppose, for example, that we want to check the (rather contrived) property that, in every reachable state, if the cache contains an address that isn't dirty, then it agrees with the main memory on at least one address:

```
fun DirtyProp (s: System) {
    some (s.cache.addrs - s.cache.dirty)
        => some a: Addr | s.cache.map[a] = s.main.map[a]
    }
```

We can assert that this property holds in the last state of every trace:

```
assert {
    all t: SystemTrace | with t | DirtyProp (last.state)
    }
```

This assertion is valid, even though *DirtyProp* is not an invariant. A write invoked in a state in which all clean entries but one had non-matching values can result in a state in which there are still clean entries but none has a matching value.

Finally, refinements are checked by assertions involving abstraction relations. We can assert that a *SystemWrite* refines a basic *Write* operation on a simple memory:

```
assert {
    all s,s': System, m,m': Memory, a: Addr, d: Data |
        Alpha (s,m) && Alpha (s',m') && SystemWrite (s,s',a,d)
        => Write (m,m',a,d)
    }
```

or that the *Flush* operation is a no-op when viewed abstractly:

```
assert {
    all s,s': System, m,m': Memory |
        Alpha (s,m) && Alpha (s',m') && Flush (s,s')
        => m.map = m'.map
    }
```

Note the form of the equality; *m = m'* would be wrong, since two distinct memories may have the same mapping, and the abstraction *Alpha* constrains only the mapping and not the memory atom itself.

Many of the assertions shown here can be made more succinct by the function shorthand explained in Section 3.7 below. For example, the assertion that a read following a write returns the value just written becomes:

```
assert {
    all s: System, a: Addr, d: Data |
        SystemRead (SystemWrite (s,d,a),a) = d
    }
```

and the assertion that *Flush* is a no-op becomes:

```
assert {
    all s: System | Alpha (s).map = Alpha (Flush (s)).map
    }
```

2.9 Polymorphism

Signatures can be parameterized by signature types. Rather than declaring a linked list whose elements belong to a particular type *Elt*, as above, we would prefer to declare a generic list:

```
sig List [T] {}
sig NonEmptyList [T] extends List [T] {elt: T, rest: List [T]}
```

Functions and facts may be parameterized in the same way, so we can define generic operators, such as:

```
fun first [T] (p: List [T]): T {result = p.elt}
fun last [T] (p: List [T]): T {some q: p.*rest | result = q.elt && no
q.rest}
fun elements [T] (p: List [T]): set T {result = p.*rest.elt}
```

In addition, let's define a generic function that determines whether two elements follow one another in a list:

```
fun follows [T] (p: List[T], a,b: T) {
    some x: p.*rest | x.elt = a && x.next.elt = b
    }
```

To see how a generic signature and operators are used, consider replacing the traces of Section 2.6 with lists of system states. Define a function that determines whether a list is a trace:

```
fun isTrace (t: List [System]) {
    Init (first(t))
    all s, s': System | follows (t,s,s') => {
        Flush (s,s')
        || (some a: Addr | Load (s,s',a))
        || (some d: Data, a: Addr | SystemWrite (s,s',d,a))
        }
    }
```

Now our assertion that every reachable system state satisfies *DirtyProp* can now be written:

```
assert {
    all t: List[System] | isTrace(t) => DirtyProp (last(t))
    }
```

2.10 Variants

To illustrate the flexibility of our notation, we sketch a different formulation of state machines oriented around transitions rather than states.

Let's introduce a signature representing state transitions of our memory system:

```
sig SystemTrans {pre,post: System}
    {pre.main.addrs = post.main.addrs}
```

Declaring the transitions as a signature gives us the opportunity to record properties of all transitions—in this case requiring that the set of addresses of the main memory is fixed.

Now we introduce a subsignature for the transitions of each operation. For example, the transitions that correspond to load actions are given by:

```
sig LoadTrans extends SystemTrans {a: Addr}
    {Load (pre, post, a)}
```

For each invariant, we define a set of states. For the states satisfying the dirty invariant, we might declare

```
sig DirtyInvStates extends System {}
```

along with the fact

```
fact {DirtyInvStates = {s: System | DirtyInv(s)}}
```

To express invariant preservation, it will be handy to declare a function that gives the image of a set of states under a set of transitions:

```
fun postimage (ss: set System, tt: set SystemTrans): set System {
    result = {s: System | some t: tt | t.pre in ss && s = t.post}
    }
```

so that we can write the assertion like this:

```
assert {postimage (DirtyInvStates, LoadTrans) in DirtyInvStates}
```

For an even more direct formulation of state machine properties, we might have defined a transition relation instead:

```
fun Trans (r: System -> System) {
    all s, s' : System |
        s->s' in r => Flush (s,s') || …
        }
```

Then, using transitive closure, we can express the set of states reachable from an initial state, and assert that this set belongs to the set characterized by some property:

```
assert {all r: System -> System, s: System |
    Init (s) && Trans(r) => s.*r in DirtyPropStates
    }
```

where *DirtyPropStates* is defined analogously to *DirtyInvStates*.

2.11 Definitions

Instead of declaring the addresses of a memory along with its mapping, as we did before:

```
sig Memory {
    addrs: set Addr,
    map: addrs ->! Data
    }
```

we could instead have declared the mapping alone:

```
sig Memory {
    map: Addr ->? Data
    }
```

and then *defined* the addresses using a subsignature:

```
sig MemoryWithAddrs extends Memory {
    addrs: set Addr}
    {addrs = {a: Addr | some a.map}}
```

Now by making the subsignature subsume all memories:

```
fact {Memory in MemoryWithAddrs}
```

we have essentially 'retrofitted' the field. Any formula involving memory atoms now implicitly constrains the *addrs* field. For example, we can assert that *Read* has an implicit precondition requiring that the argument be a valid address:

```
assert {all m: Memory, a: Addr, d: Data | Read (m,d,a) => a in
m.addrs}
```

even though the specification of *Read* was written when the field *addrs* did not even exist.

3 Semantics

For completeness, we give an overview of the semantics of the language. The novelties with respect to the original version of Alloy [4] are (1) the idea of organizing relations around basic types as signatures, (2) the treatment of extension as subsetting, and (3) the packaging of formulas in a more explicit (and conventional) style. The semantic basis has been made cleaner, by generalizing relations to arbitrary arity, eliminating 'indexed relations' and the need for a special treatment of sets.

3.1 Types

We assume a universe of atoms. The standard notion of a mathematical relation gives us our only composite datatype. The value of an expression will always be a relation—that is, a collection of tuples of atoms. Relations are first order: the elements of a tuple are themselves atoms and never relations.

The language is strongly typed. We partition the universe into subsets each associated with a *basic* type, and write $(T_1, T_2, \ldots, T_n)$ for the type of a relation whose tuples each consist of n atoms, with types T_1, T_2, etc.

A set is represented semantically as a unary relation, namely a relation whose tuples each contain one atom. A tuple is represented as a singleton relation, namely a relation containing exactly one tuple. A scalar is represented as a unary, singleton relation. We use the terms 'set', 'tuple' and 'scalar' to describe relations with the appropriate properties. Basic types are used only to construct relation types, and every expression that appears in a specification has a relational type. Often we will say informally that an expression has a type T where T is the name of a basic type when more precisely we mean that the expression has the type (T).

So, in contrast to traditional mathematical style, we do not make distinctions amongst the atom a, the tuple (a), the set $\{a\}$ containing just the atom, or the set $\{(a)\}$ containing the tuple, and represent all of these as the last. This simplifies the semantics and gives a more succinct and uniform syntax.

3.2 Expression Operators

Expressions can be formed using the standard set operators written as ASCII characters: union (+), intersection (&) and difference (-). Some standard relational operators, such as transpose (~) and transitive closure (^), can be applied to expressions that denote binary relations. Relational override (++) has its standard meaning for binary relations but can applied more broadly.

There are two special relational operators, dot and arrow. The dot operator is a generalized relational composition. Given expressions *p* and *q*, the expression *p.q* contains the tuple $\langle p_1, \ldots p_{m-1}, q_2, \ldots, q_n\rangle$ when *p* contains $\langle p_1, \ldots, p_m\rangle$, *q* contains $\langle q_1, \ldots q_n\rangle$, and $p_m = q_1$. The last type of *p* and the first type of *q* must match, and $m + n$, the sum of the arities of *p* and *q*, must be three or more so that the result is not degenerate. When *p* is a set and *q* is a binary relation, the composition *p.q* is the standard relational image of *p* under *q*; when *p* and *q* are both binary relations, *p.q* is standard relational composition. In all of the examples above, the dot operator is used only for relational image.

The arrow operator is cross product: *p -> q* is the relation containing the tuple $\langle p_1, \ldots, p_m, q_1, \ldots q_n\rangle$ when *p* contains $\langle p_1, \ldots, p_m\rangle$, and *q* contains $\langle q_1, \ldots q_n\rangle$. In all the examples in this paper, *p* and *q* are sets, and *p -> q* is their standard cross product.

3.3 Formula Operators

Elementary formulas are formed from the subset operator, written *in*. Thus *p in q* is true when every tuple in *p* is in *q*. The formula *p : q* has the same meaning, but when *q* is a set, adds an implicit constraint that *p* be scalar (ie, a singleton). This constraint is overridden by writing *p: option q* (which lets *p* to be empty or a scalar) or *p: set q* (which eliminates the constraint entirely). Equality is just standard set equality, and is short for a subset constraint in each direction.

An arrow that appears as the outermost expression operator on the right-hand side of a subset formula can be annotated with *multiplicity markings*: + (one or more), ? (zero or one) and ! (exactly one). The formula

```
r: S m -> n T
```

where *m* and *n* are multiplicity markings constrains the relation *r* to map each atom of *S* to *n* atoms of *T*, and to map *m* atoms of *S* to each atom of *T*. *S* and *T* may themselves be product expressions, but are usually variables denoting sets. For example,

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

make *r* respectively a total function on *S* and an injection.

Larger formulas are obtained using the standard logical connectives: && (and), || (or), ! (not), => (implies), *iff* (bi-implication). The formula *if b then f else g* is short for *b => f && !b => g*. Within curly braces, consecutive formulas are implicitly conjoined.

Quantifications take their usual form:

```
all x: e | F
```

is true when the formula *F* holds under every binding of the variable *x* to a member of the set *e*. In addition to the standard quantifiers, *all* (universal) and *some* (existential), we have *no*, *sole* and *one* meaning respectively that there are no values, at most one value, and exactly one value satisfying the formula. For a quantifier *Q* and expression *e*, the formula *Q e* is short for *Q x: T | x in e* (where *T* is the type of *e*), so *no e*, for example, says that *e* is empty.

The declaration of a quantified formula is itself a formula—an elementary formula in which the left-hand side is a variable. Thus

```
some x = e | F
```

is permitted, and is a useful way to express a *let* binding. Quantifiers may be higher-order; the formula

```
all f: s ->! t | F
```

is true when *F* holds for every binding of a total function from *s* to *t* to the variable *f*. Our analysis tool cannot currently handle higher-order quantifiers, but many uses of higher-order quantifiers that arise in practice can be eliminated by skolemization.

Finally, we have relational comprehensions; the expression

$\{x_1: e_1, x_2: e_2, \ldots \mid F\}$

constructs a relation of tuples with elements x_1, x_2, etc., drawn from set expressions e_1, e_2, etc., whose values satisfy *F*.

3.4 Signatures

A *signature* declaration introduces a basic type, along with a collection of relations called *fields*. The declaration

```
sig S {f: E}
```

declares a basic type *S*, and a relation *f*. If *E* has the type $(T_1, T_2, \ldots, T_n)$, the relation *f* will have the type $(S, T_1, T_2, \ldots, T_n)$, and if *x* has the type *S*, the expression *x.f* will have the same type as *E*. When there are several fields, field names already declared may appear in expressions on the right-hand side of declarations; in this case, a field *f* is typed as if it were the expression *this.f*, where *this* denotes an atom of the signature type (see Section 3.6).

The meaning of a specification consisting of a collection of signature declarations is an assignment of values to global constants– the signatures and the fields. For example, the specification

```
sig Addr {}
sig Data {}
sig Memory {map: Addr -> Data}
```

has 4 constants—the three signatures and one field—with assignments such as:

```
Addr = {a0, a1}
Data = {d0, d1, d2}
Memory = {m0, m1}
map = {(m0,a0,d0), (m1,a0,d1), (m1,a0,d2)}
```

corresponding to a world in which there are 2 addresses, 3 data values and 2 memories, with the first memory (*m0*) mapping the first address (*a0*) to the first data value (*d0*), and the second memory (*m1*) mapping the first address (*a0*) both to the second (*d1*) and third (*d2*) data values.

A fact is a formula that constrains the constants of the specification, and therefore tends to reduce the set of assignments denoted by the specification. For example,

```
fact {all m: Memory | all a: Addr | sole m.map[a]}
```

rules out the above assignment, since it does not permit a memory (such as *m1*) to map an address (such as *a0*) to more than one data value.

The meaning of a function is a set of assignments, like the meaning of the specification as a whole, but these include bindings to parameters. For example, the function

```
fun Read (m: Memory, d: Data, a: Addr) {
    d = m.map[a]
    }
```

has assignments such as:

```
Addr = {a0, a1}
Data = {d0, d1, d2}
Memory = {m0, m1}
map = {(m0,a0,d1)}
m = {m0}
d = {d1}
a = {a0}
```

The assignments of a function representing a state invariant correspond to states satisfying the invariant; the assignments of a function representing an operation (such as *Read*) correspond to executions of the operation.

An assertion is a formula that is claimed to be *valid*: that is, true for every assignment that satisfies the facts of the specification. To check an assertion, one can search for a *counterexample*: an assignment that makes the formula false. For example, the assertion

```
assert {
    all m,m': Memory, d: Data, a: Addr |
        Read (m,d,a) => Read (m',d,a)}
```

which claims, implausibly, that if a read of memory *m* returns *d* at *a*, then so does a read at memory *m'*, has the counterexample

```
Addr = {a0}
Data = {d0,d1}
Memory = {m0, m1}
map = {(m0,a0,d0), (m1,a0,d1)}
```

To find a counterexample, a tool should negate the formula and then skolemize away the bound variables, treating them like the parameters of a function, with values to be determined as part of the assignment. In this case, the assignment might include:

```
m = {m0}
m' = {m1}
d = {d0}
a = {a0}
```

3.5 Extension

Not every signature declaration introduces a new basic type. A signature declared without an extension clause is a *type signature*, and creates both a basic type and a set constant of the same name. A signature *S* declared as an extension is a *subsignature*, and creates only a set constant, along with a constraint making it a subset of each *supersignature* listed in the extension clause. The subsignature takes on the type of the supersignatures, so if there is more than one, they must therefore have the same type, by being direct or indirect subsignatures of the same type signature.

A field declared in a subsignature is as if declared in the corresponding type signature, with the constraint that the domain of the field is the subsignature. For example,

```
sig List {}
sig NonEmptyList extends List {elt: Elt,rest: List}
```

makes *List* a type signature, and *NonEmptyList* a subset of *List*. The fields *elt* and *rest* map atoms from the type *List*, but are constrained to have domain *NonEmptyList*. Semantically, it would have been equivalent to declare them as fields of *List*, along with facts constraining their domains:

```
sig List {elt: Elt,rest: List}
sig NonEmptyList extends List {}
fact {elt.Elt in NonEmptyList}
fact {rest.List in NonEmptyList}
```

(exploiting our dot notation to write the domain of a relation *r* from *S* to *T* as *r.T*).

3.6 Overloading and Implicit Prefixing

Whenever a variable is declared, its type can be easily obtained from its declaration (from the type of the expression on the right-hand side of the declaration), and every variable appearing in an expression is declared in an enclosing scope. The one complication to this rule is the typing of fields.

For modularity, a signature creates a local namespace. Two fields with the name *f* appearing in different signatures do not denote the same relational constant. Interpreting an expression therefore depends on first resolving any field names that appear in it. In an expression of the form *e.f*, the signature to which *f* belongs is determined according to the type of *e*. To keep the scheme simple, we require that sometimes the specifier resolve the overloading explicitly by writing the field *f* of signature *S* as S*f*. (*At the end of the previous section, for example, the reference in the fact to* rest *should actually be to* List*rest*, since the context does not indicate which signature *rest* belongs to.)

In many formulas, a single expression is dereferenced several times with different fields. A couple of language features are designed to allow these formulas to be written more succinctly, and, if used with care, more comprehensibly. First, we provide two syntactic variants of the dot operator. Both *p*::*q* and *q*[*p*] are equivalent to *p.q*, but have different precedence: the double colon binds more tightly than the dot, and the square brackets bind more loosely than the dot. Second, we provide a *with* construct similar to Pascal's that makes dereferencing implicit.

Consider, for example, the following simplified signature for a trace:

```
sig Trace {
    ticks: set Tick,
    first: ticks,
    next: ticks -> ticks,
    state: ticks -> State
    }
```

Each trace *t* has a set of ticks *t.ticks*, a first tick *t.first*, an ordering *t.next* that maps ticks to ticks, and a relation *t.state* mapping each tick to a state. For a trace *t* and tick *k*, the state is *k*.(*t.state*); the square brackets allow this expression to be written instead as *t.state*[*k*]. To constrain *t.ticks* to be those reachable from *t.first* we might write:

```
fact {all t: Trace | (t.first).*(t.next ) = t.ticks}
```

Relying on the tighter binding of the double colon, we can eliminate the parentheses:

```
fact {all t: Trace | t::first.*t::next = t.ticks}
```

Using *with*, we can make the *t* prefixes implicit:

```
fact {all t: Trace | with t | first.*next = ticks}
```

In general, *with e | F* is like *F*, but with *e* prefixed wherever appropriate to a field name. Appropriateness is determined by type: *e* is matched to any field name with which it can be composed using the dot operator. A fact attached to a signature *S* is implicitly enclosed by *all this: S | with this |*, and the declarations of a signature are interpreted as constraints as if they had been declared within this scope. Consequently, the declaration of *first* above should be interpreted as if it were the formula:

```
all this: Trace | with this | first: ticks
```

which is equivalent to

```
all this: Trace | this.first: this.ticks
```

and should be typed accordingly.

3.7 Function Applications

A function may be applied by binding its parameters to expressions. The resulting application may be either an expression or a formula, but in both cases the function body is treated as a formula. The formula case is simple: the application is simply short for the body with the formal parameters replaced by the actual expressions (and bound variables renamed where necessary to avoid clashes).

The expression case is more interesting. The application is treated as a syntactic sugar. Suppose we have a function application expression, *e* say, of the form

$f(a_1, a_2, \ldots, an)$

that appears in an elementary formula *F*. The declaration of the function *f* must list *n* + 1 formal arguments, of which the *second* will be treated as the result. The entire elementary formula is taken to be short for

all result: D | $f(a_1$, result, $a_2, \ldots, a_n)$ => F [result/e]

where *D* is the right-hand side of the declaration of the missing argument, and *F* [*result*/*e*] is *F* with the fresh variable *result* substituted for the application expression *e*. The application of *f* in this elaborated formula is now a formula, and is treated simply as an inlining of the formula of *f*.

To see how this works, consider the definition of a function *dom* that gives the domain of a relation over signature *X*:

```
fun dom (r: X -> X, d: set X) {d = r.X}
```

(We have defined the function monomorphically for a homogeneous relation. In practice, one would define a polymorphic function, but we want to avoid conflating two unrelated issues.)

Here is a trivial assertion that applies the function as an expression:

```
assert {all p: X -> X | (dom (p)).p in X}
```

Desugaring the formula, we get

```
all p: X -> X | all result: set X | dom (p, result) => result.p in X
```

and then inlining

```
all p: X -> X | all result: set X | result = p.X => result.p in X
```

This formula can be reduced (by applying a universal form of the One Point Rule) to

```
all p: X -> X | (p.X).p in X
```

which is exactly what would have been obtained had we just replaced the application expression by the expression on the right-hand side of the equality in the function's definition!

Now let's consider an implicit definition. Suppose we have a signature *X* with an ordering *lte*, so that *e.lte* is the set of elements that *e* is less than or equal to, and a function *min* that gives the minimum of a set, defined implicitly as the element that is a member of the set, and less than or equal to all members of the set:

```
sig X {lte: set X}
fun min (s: set X, m: option X) {
    m in s && s in m.lte
    }
```

Because the set may be empty, *min* is partial. Depending on the properties of *lte* it may also fail to be deterministic. A formula that applies this function

```
assert {all s: set X | min (s) in s}
```

can as before be desugared

```
all s: set X | all result: option X | min (s, result) => result in s
```

and expanded by inlining

```
all s: set X | all result: option X |
    (result in s) && s in result.lte => result in s
```

but in this case the One Point Rule is not applicable.

As a convenience, our language allows the result argument of a function to be declared anonymously in a special position, and given the name *result*. The domain function, for example, can be defined as:

```
fun dom (r: X -> X): set X {result = r.X}
```

How the function is defined has no bearing on how it is used; this definition is entirely equivalent to the one above, and can also be applied as a formula with two arguments.

3.8 Polymorphism

Polymorphism is treated as a syntactic shorthand. Lack of space does not permit a full discussion here.

4 Related Work

We have shown how a handful of elements can be assembled into a rather simple but flexible notation. The elements themselves are far from novel—indeed, we hope that their familiarity will make the notation easy to learn and use—but their assembly into a coherent whole results in a language rather different from existing specification languages.

4.1 New Aspects

The more novel aspects of our work are:

- *Objectification of state.* Most specification languages represent states as cartesian products of components; in our approach, a state, like a member of any signature, is an individual—a distinct atom with identity. A similar idea is used in the situation calculus [11], whose 'relational fluents' add a situation variable to each time-varying relation. The general idea of objectifying all values is of course the foundation of object-oriented programming languages, and was present in LISP. Interestingly, object-oriented variants of Z (such as [1]) do not objectify schemas. The idea of representing structures in first-order style as atoms is present also in algebraic specifications such as Larch [2], which treat even sets and relations in this manner.
- *Components as relations.* Interpreting fields of a structure as functions goes back to early work on verification, and is widely used (for example, by Leino and Nelson [10]). We are not aware, however, of specification languages that use this idea, or that flatten fields to relations over atoms.
- *Extension by global axioms.* The 'facts' of our notation allow the properties of a signature to be extended monotonically. The idea of writing axioms that constrain the members of a set constant declared globally is hardly remarkable, but it appears not to have been widely exploited in specification languages.
- *Extension by subset.* Treating the extension of a structure as a refinement modelled by subset results in a simple semantics, and melds well with the use of global axioms. Again, this seems to be an unremarkable idea, but one whose power has not been fully recognized.

4.2 Old Aspects

The aspects of our work that are directly taken from existing languages are:

- *Formulas.* The idea of treating invariants, definitions, operations, etc, uniformly as logical formulas is due to Z [14].
- *Assertions.* Larch [2] provides a variety of constructs for adding intentional redundancy to a specification in order to provide error-detection opportunities.
- *Parameterized formulas.* The 'functional' style we have adopted, in which all formulas are explicitly parameterized, in contrast to the style of most specification languages, is used also by languages for theorem provers, such as PVS [13]. VDM [8] offers a mechanism called 'operation quotation' in which pre- and post conditions are reused by interpreting them as functions similar to ours.
- *Parametric Polymorphism.* The idea of parameterizing descriptions by types was developed in the programming languages community, most notably in the context of ML [12].
- *Implicit Prefixing.* Our 'with' operator is taken from Pascal [9].
- *Relational operators.* The dot operator, and the treament of scalars as singletons, comes from the earlier version of Alloy [4].

4.3 Z's Schema Calculus

Z has been a strong influence on our work; indeed, this paper may be viewed as an attempt to achieve some of the power and flexibility of Z's schema calculus in a first-order setting. Readers unfamiliar with Z can find an excellent presentation of the schema calculus in [16]. The current definitive reference is [15], although Spivey's manual [14] is more accessible for practioners.

A *schema* consists of a collection of variable declarations and a formula constraining the variables. Schemas can be anonymous. When a name has been bound to a schema, it can be used in three different ways, distinguished according to context. First, it can be used as a *declaration*, in which case it introduces its variables into the local scope, constraining them with its formula. Second, where the variables are already in scope, it can be used as a *predicate*, in which case the formula applies and no new declarations are added. Both of these uses are syntactic; the schema can be viewed as a macro.

In the third use, the schema is semantic. Its name represents a set of *bindings*, each binding being a finite function from variables names to values. The bindings denoted by the schema name are the models of the schema's formula: those bindings of variable names to values that make the formula true.

How a schema is being applied is not always obvious; in the set comprehension $\{S\}$, for example, S represents a declaration, so that the expression as a whole denotes the same set of bindings as S itself. Given a binding b for a schema with component variable x, the expression $b.x$ denotes the value assigned to x in b. Unlike Alloy's dot, this dot is a function application, so for a set of bindings B, the expression $B.x$ is not well formed.

Operations in Z are expressed using the convention that primed variables denote components of the post-state. A mechanism known as *decoration* allows one to write S' for the schema that is like S, but whose variable names have been primed. Many idioms, such as promotion, rely on being able to manipulate the values of a schema's variables in aggregate. To support this, Z provides the theta operator: θS is an expression that denotes a binding in which each variable x that belongs to S is bound to a variable of the same name x declared in the local scope. Theta and decoration interact subtly: $\theta S'$ is not a binding of S', but rather binds each variable x of S to a variable x' declared locally. So where we would write $s=s'$ to say that pre- and post-states s and s' are the same, a Z specifier would write $\theta S = \theta S'$. This formula equates each component x of S to its matching component x' of S', because x and x' are the respective values bound to x by θS and $\theta S'$ respectively.

Our 'fact' construct allows the meaning of a signature name to be constrained subsequent to its declaration. A schema, in contrast, is 'closed': a new schema name must be introduced for each additional constraint. This can produce an undesirable proliferation of names for a system's state, but it does make it easier to track down those formulas that affect a schema's meaning.

The variables of a schema can be renamed, but cannot be replaced by arbitrary expressions (since this would make nonsense of declarations).This requires the introduction of existential quantifiers where in our notation an expression is passed as an actual. On the other hand, when no renaming is needed, it is more succinct.

Z's sequential composition operator is defined by a rather complicated transformation, and relies on adherence to particular conventions. The schema P ; Q is obtained by collecting primed variables in P that match unprimed variables in Q; renaming these in both P and Q with a new set of variable names; and then existentially quantifying the new names away. For example, to say that a read following a write to the same address yields the value written, we would write:

```
all m: Memory, a: Addr, d, d': Data | Read (Write(m,a,d),d') => d = d'
```

which is short for

```
all m: Memory, a: Addr, d, d': Data |
    all m': Memory | Write (m,m',a,d) => Read (m,a,d') => d = d'
```

In Z, assuming appropriate declarations of a schema *Memory* and a given type *Data*, the formula would be:

```
∀ Memory; Memory'; x!: Data • Write ; Read [x!/d!] ⇒ x! = d!
```

which is short for

```
∀ Memory; Memory'; x!: Data •
    ∃ Memory'' •
        ∃ Memory' • Write ∧ θMemory' = θMemory''
        ∃ Memory'; d!: Data •
            Read ∧ θMemory = θMemory'' ∧ d! = x!
    ⇒ x! = d!
```

The key semantic difference between signatures and schemas is this. A signature is a set of atoms; its fields are relational constants declared in global scope. A schema, on the other hand, denotes a higher-order object: a set of functions from field names to values. Our approach was motivated by the desire to remain first order, so that the analysis we have developed [3] can be applied. Not surprisingly, there is a cost in expressiveness. We cannot express higher-order formulas, most notably those involving preconditions. Suppose we want to assert that our write operation has no implicit precondition. In Z, such an assertion is easily written:

```
∀ Memory; a?: Addr •  ∃ Memory'; d!: Data • Write
```

We might attempt to formulate such an assertion in our notation as follows:

```
assert {
    all m: Memory, a: Addr, d: Data | some m': Memory | Write
(m,m',d,a) }
```

Unfortunately, this has counterexamples such as

```
Addr = {a0}
Data = {d0}
Memory = {m0, m1}
map = {}
```

in which the *map* relation lacks an appropriate tuple. Intuitively, the assertion claims that there is no context in which a write cannot proceed; a legitimate counterexample—but one we certainly did not intend—simply gives a context in which a memory with the appropriate address-value mapping is not available.

We have focused in this discussion on schemas. It is worth noting that Z is expressive enough to allow a style of structuring almost identical to ours, simply by declaring signatures as given types, fields and functions as global variables, and by writing facts, and the bodies of functions, as axioms. Field names would have to be globally unique, and the resulting specification would likely be less succinct than if expressed in our notation.

4.4 Phenomenology

Pamela Zave and Michael Jackson have developed an approach to composing descriptions [18] that objectifies states, events and time intervals, and constrains their properties with global axioms. Objectification allows descriptions to be reduced to a common phenomenology, so that descriptions in different languages, and even in different paradigms can be combined. Michael Jackson has argued separately for the importance of objectification as a means of making a more direct connection between a formal description and the informal world: as he puts it, "domain phenomena are facts about individuals" [7]. It is reassuring that the concerns of language design and tractability of analysis that motivated our notation are not in conflict with sound method, and it seems that our notation would be a good choice for expressing descriptions in the form that Zave and Jackson have proposed.

5 Evaluation

5.1 Merits

The key motivations of the design of our mechanism have been minimality and flexibility. It is worth noting how this has been achived by the *omission* of certain features:

- There is only one form of semantic structuring; our opinion is that adding extra mechanisms, for example to group operations into classes, does not bring enough benefit to merit the additional complexity, and tends to be inflexible. (Our language does provide some namespace control for signature and paragraph names in the style of Java packages, but this is trivial and does not interact with the basic mechanism).
- There is no subtyping; subsignatures are just subsets of their supersignatures, and have the same type. There are only two types: basic types (for signatures), and relational types (for expressions). Types are not nested.
- There is only one way that formulas are packaged for reuse. The same function syntax is used for observers, operations, refinement relations, etc. The function shorthand syntax unifies the syntax of both declaration and use for explicit and implicit function definitions.
- The values of a signature with fields are just like the values of any basic type; there is nothing like Z's notion of a schema binding.

Our interpretation of a subsignature as a subset of the supersignature appears to be novel as a mechanism for structuring in a specification language. It has three nice consequences:

- *Elimination of type coercions*. If x belongs to a signature S whose extension S' defines a field f, the expression $x.f$ will just denote an empty set if x does not belong to S'. Contrast this with the treatment of subclasses in the Object Constraint

Language [17], for example, which results in pervasive coercions and often prevents the use of set and relation operators (since elements must be coerced one at a time).

- *Ease of extension*. Constraints can be added to the subsignature simply by writing a constraint that is universally quantified over elements of that subset.
- *Definitional extension*. We can declare an extension *S'* of a signature *S* with additional fields, relate these fields to the fields declared explicitly for *S*, and then record the fact that *S*=*S'* (as illustrated in Section 2.11). The effect is that every atom of *S* has been extended with appropriately defined fields, which can be accessed whenever an expression denoting such an atom is in scope! We expect to find this idiom especially useful for defining additional fields for visualization purposes.

5.2 Deficiencies

One might wonder whether, having encoded structures using atoms, and having provided quantifiers over those atoms, one can express arbitrary properties of higher-order structures. Unfortunately, but not surprisingly, this is not possible. The catch is that fields are treated in any formulas as global variables that are existentially quantified. To simulate higher-order logic, it would be necessary to allow quantifications over these variables, and since they have relational type, that would imply higher-order quantification. The practical consequence is that properties requiring higher-order logic cannot be expressed. One cannot assert that the precondition of an operation is no stronger than some predicate; one cannot in general specify operations by minimization; and one cannot express certain forms of refinement check. An example of this problem is given in Section 4.3 above. Whether the problem is fundamental or can be partially overcome remains to be seen.

The treatment of subsignatures as subsets has a nasty consequence. Since a field declared in a subsignature becomes implicitly a field of the supersignature, two subsignatures cannot declare fields of the same name. The extension mechanism is therefore not properly modular, and a specification should use hierarchical structure instead where this matters.

Modelling a set of states as atoms entails a certain loss of abstraction. In this specification

```
sig A {}
sig S {a: A}
fun op (s,s': S) {s.a = s'.a}
```

the operation *op* has executions in which the pre- and post-states are equal (that is, the same atom in *S*), and executions in which only their *a* components are equal. One might object that this distinction is not observable. Moreover, replacing the formula by *s*=*s'* would arguably be an overspecification—a 'bias' in VDM terminology [8]. The situation calculus [11] solves this problem by requiring every operation to produce a state change: *s* and *s'* are thus regarded as distinct situations by virtue of occurring at different points in the execution. The dual of this solution is to add an axiom requiring that no two distinct atoms of *S* may have equal *a* fields. Either of these solutions is easily imposed in our notation.

Our treatment of scalars and sets uniformly as relations has raised the concern that the resulting succinctness comes with a loss of clarity and redundancy. Extensive use of the previous version of our language, mostly by inexperienced specifiers, suggests that this is not a problem. The loss of some static checking is more than compensated by the semantic analysis that our tool performs.

6 Conclusion

Two simple ideas form the basis of our modularity mechanism: (1) that a structure is just a set of atoms, and its fields are global relations that map those atoms to structure components; and (2) that extensions of a structure are just subsets. Our relational semantics, in which all variables and fields are represented as relations, makes the use of structures simple and succinct, and it ensures that the language as a whole remains first order. For a variety of modelling tasks, we believe that our approach provides a useful balance of expressiveness and tractability.

Acknowledgments

The language described here was refined by experience writing specifications, long before an analyzer existed, and by the development of the analyzer tool itself. Mandana Vaziri and Sarfraz Khurshid were our early adopters, and Brian Lin and Joe Cohen helped implement the tool. The paper itself was improved greatly by comments from Mandana and Sarfraz, from Michael Jackson, from Tomi Mannisto, and especially from Pamela Zave, whose suggestions prompted a major rewrite. Jim Woodcock helped us understand Z, and the clarity and simplicity of his own work has been a source of inspiration to us. Our ideas have also been improved by the comments of the members of IFIP working groups 2.3 and 2.9, especially Tony Hoare, Greg Nelson and Rustan Leino. This work was funded in part by ITR grant #0086154 from the National Science Foundation, by a grant from NASA, and by an endowment from Doug and Pat Ross.

References

[1] R. Duke, G. Rose and G. Smith. Object-Z: A Specification Language Advocated for the Description of Standards. SVRC Technical Report 94-45. The Software Verification Research Centre, University of Queensland, Australia.

[2] 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.

[3] Daniel Jackson. Automating first-order relational logic. Proc. ACM SIGSOFT Conf. Foundations of Software Engineering. San Diego, November 2000.

[4] Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. To appear, ACM Transactions on Software Engineering and Methodology, October 2001.

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

[6] Daniel Jackson and Jeannette Wing. Lightweight Formal Methods. In: H. Saiedian (ed.), An Invitation to Formal Methods. IEEE Computer, 29(4):16-30, April 1996.

[7] Michael Jackson. Software Requirements and Specifications: A

Lexicon of Practice, Principles and Prejudices. Addison-Wesley, 1995.

[8] Cliff Jones. Systematic Software Development Using VDM. Second edition, Prentice Hall, 1990.

[9] Kathleen Jensen and Nicklaus Wirth. Pascal: User Manual and Report. Springer-# Verlag, 1974.

[10] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding . Research Report 160, Compaq Systems Research Center, November 2000.

[11] Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for the Situation Calculus. Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 3(1998), Nr. 018.

[12] Robin Milner, Mads Tofte and Robert Harper. The Definition of Standard ML. MIT Press, 1990.

[13] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS Language Reference. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

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

[15] Ian Toyn et al. Formal Specification—Z Notation—Syntax, Type and Semantics. Consensus Working Draft 2.6 of the Z Standards Panel BSI Panel IST/5/-/19/2 (Z Notation). August 24, 2000.

[16] Jim Woodcock and Jim Davies. Using Z: Specification, Refinement and Proof. Prentice Hall, 1996.

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

[18] Pamela Zave and Michael Jackson. Conjunction as Composition. ACM Transactions on Software Engineering and Methodology II(4): 379–411, October 1993.