Appendix E: Alternative Approaches

The models in this appendix were contributed by Michael Butler, John Fitzgerald, Martin Gogolla, Peter Gorm Larsen, and Jim Woodcock, and are included here with their permission.

Alloy is only one of several approaches to the modeling and analysis of software abstractions. This appendix briefly describes four of these alternatives: B, OCL, VDM, and Z. Its purpose is both to help those in search of an approach that matches their needs, and—for readers already familiar with other approaches—to highlight the respects in which they differ from Alloy.

I chose these four approaches because of their ability to capture complex structure succinctly and abstractly. They are all well known, and each has an active and enthusiastic community of users and researchers. Other modeling and analysis approaches can be used effectively for software design in specialized domains: there are many model checkers, for example, that can check protocols and concurrent algorithms, but they are not considered here since (with the exception of FDR) they tend to have only rudimentary support for structuring of data.

Some features are common to all the approaches, including Alloy. They all offer a notation that can capture software abstractions more succinctly and directly than a programming language can; all of them, despite differences in syntax and semantics, view the state in terms of classical mathematical structures, such as sets and relations, and describe behaviors declaratively, using constraints. Lightweight tools are available for all of them, in which constraints are evaluated against concrete cases, and new tool projects are underway for all these approaches that are likely to extend their power and applicability greatly.

At the same time, there are important differences. B is more operational in flavor; its notation is more like an abstract programming language than a specification language. OCL has a very different syntax from the others, reminiscent of Smalltalk. In B and VDM, and to some extent Z and OCL, a particular notion of state machine is hardwired, in contrast to Alloy, which is designed to support a variety of idioms, each as easy

(or difficult!) to express as the others. B, VDM and Z were designed more with proof in mind than lightweight analysis, and so, unlike Alloy and OCL, are supported by specialized theorem provers.

All these languages predate Alloy, which has benefited greatly from their experience. Alloy was designed for similar applications, but with more emphasis on automatic analysis. In pursuit of this goal, the language was stripped down to the bare essentials of structural modeling, and was developed hand-in-hand with its analysis. Any feature that would thwart analysis was excluded. Consequently, Alloy's analysis is more powerful than the lightweight analyses offered by the other approaches, which (with the exception of ProB) are mostly "animators" that execute a model on given test cases. Unlike an animator, the Alloy Analyzer does not require the user to provide initial conditions and inputs; it does not restrict the language to an executable subset; and, because it covers the entire space within the scope, it is more effective at uncovering subtle bugs. The idea of analysis is built in to the language itself: assertions can be recorded as part of a specification, and the scopes (which bound the analysis) are confined to commands. The other approaches use tool-specific extensions instead.

Another goal in the design of Alloy was to be unusually small and simple; it has fewer concepts than the other languages, and is in some respects more flexible. For example, Alloy unifies all data structuring within the notion of a relation; it uses the same relational join for indexing, dereferencing structures and applying functions; its signatures can simulate the schemas of Z and the classes of OCL; and its assertions can express invariant preservations, refinements and temporal properties over traces.

These benefits are not, of course, without some cost. Alloy is less expressive than the other languages. Whereas Alloy's structures are strictly first order, B, VDM and Z all support higher-order structures and quantifications. Carroll Morgan's well known telephone switching specification [55] in Z, for example, represents the connections as a set of sets of endpoints. Such a structure is not directly representable in Alloy; you'd need to model the connections as a relation between endpoints, or as a set of connection atoms, each mapped to its endpoints. Morgan's ingenious characterization of the behavior of the switch, with a higher-order formula constraining the connection structure to be maximal, would not be expressible at all in Alloy. A more significant (but less fundamental) deficiency of Alloy in this regard is its relatively poor support for sequences and integers.

Aside from occupying a different point in the spectrum of expressiveness versus analyzability, the other languages naturally have their own particular merits. B offers a more direct path to implementation; OCL is integrated with UML, the modeling language of choice for many companies; VDM supports both explicit and implicit forms of modeling; and Z has higher-order features that have been found very useful in the structuring of large specifications.

A single problem is used to illustrate all the approaches. For each alternative approach, a model was constructed by an expert. Michael Butler developed the B version, using the ProB tool; Martin Gogolla developed the OCL version using the USE tool; Peter Gorm Larsen and John Fitzgerald developed the VDM version using VDMTools; and Jim Woodcock developed the Z version using the Z/Eves theorem prover and the Jaza animator. Unfortunately, there was not sufficient space to include all their work in full. In particular, Martin Gogolla wrote a second model showing that OCL could accommodate the "time-instant" idiom used in the Alloy specification as easily as the standard pre/post idiom, and constructed an ASSL procedure for generating test cases automatically; and Jim Woodcock proved precondition theorems for all operations, and the *NoIntruder* assertion with the help of Z/Eves.

E.1 An Example

To illustrate the different approaches, we'll use an example of a scheme for recodable hotel-door locks, similar to (but simpler than) the one that appears in chapter 6. The purpose of the modeling and analysis is to determine whether the scheme is effective in preventing unauthorized access. An Alloy model is shown in figs. E.1 and E.2.

Fig. E.1 shows the declarations of the components of the state space, and the initialization. Each key card is marked with two keys (line 5); these markings are fixed, and do not change over time. The remaining components are time-varying, as can be seen by the presence of the *Time* column in their declarations: the current key for each room (8); the front desk record of keys issued so far (11), and of which keys were issued for which rooms in the immediately previous checkin (12); and the set of cards held by each guest (15).

At initialization, the record at the front desk associating keys with rooms matches the current keys of the room locks themselves (18), no keys have been issued, and no guests hold cards (19).

```
1  module hotel
2  open util/ordering [Time]

3  sig Key, Time {}
4  sig Card {
5      fst, snd: Key
6      }
7  sig Room {
8      key: Key one -> Time
9      }
10 one sig Desk {
11     issued: Key -> Time,
12     prev: (Room -> lone Key) -> Time
13     }
14 sig Guest {
15     cards: Card -> Time
16     }

17 pred init (t: Time) {
18     Desk.prev.t = key.t
19     no issued.t and no cards.t
20     }
```

FIG. E.1 Hotel locking example, in Alloy: part 1.

Fig. E.2 shows the operations corresponding to checking in and entering a room, the definition of execution traces, and an assertion expressing the intended effect of the scheme in terms of denied access.

When a guest *g* checks in at the front desk to a room *r*, the guest is given a card (5) whose first key is the last key that was issued for that room (3), and whose second key is fresh (4). The desk's records are updated accordingly (6, 7). There is no change to the keys in the locks (9).

A guest can enter a room so long as he or she is holding a card (12) whose first or second key matches the current key of the room's lock. If the second key matches, the lock's key remains the same (14); if the first key matches, the lock is recoded with the second key (15). No changes are made to the front desk's records (17) or to the sets of keys that guests hold (18).

To shorten the example, no operation is given for checking out. The use of a key by a new guest should invalidate previously issued keys, so

```
pred checkin (t, t': Time, r: Room, g: Guest) {
    some c: Card {
        c.fst = r.(Desk.prev.t)
        c.snd not in Desk.issued.t
        cards.t' = cards.t + g->c
        Desk.issued.t' = Desk.issued.t + c.snd
        Desk.prev.t' = Desk.prev.t ++ r->c.snd
        }
    key.t = key.t'
    }
pred enter (t, t': Time, r: Room, g: Guest) {
    some c: g.cards.t |
        let k = r.key.t {
            c.snd = k and key.t' = key.t
            or c.fst = k and key.t' = key.t ++ r->c.snd
            }
    issued.t = issued.t' and (Desk<:prev).t = prev.t'
    cards.t = cards.t'
    }
fact Traces {
    init [first]
    all t: Time - last |
        some g: Guest, r: Room |
            checkin [t, next [t], r, g] or enter [t, next [t], r, g]
    }
assert NoIntruder {
    no t1: Time, disj g, g': Guest, r: Room |
        let t2 = next [t1], t3 = next [t2], t4 = next [t3] {
            enter [t1, t2, r, g]
            enter [t2, t3, r, g']
            enter [t3, t4, r, g]
            }
    }
check NoIntruder for 3 but 6 Time, 1 Room, 2 Guest
```

FIG. E.2 Hotel locking example, in Alloy: part 2.

whenever a guest checks in, the previous occupant is implicitly checked out.

If you're reading this appendix before you've read the rest of the book, a few comments about Alloy might be helpful:

- A signature introduces a set, and some relations that have that set as their first column. For example, the declaration for *sig Card* introduces the set *Card* of key cards, and two relations, *fst* and *snd*, from *Card* to *Key*.
- Multiplicities of relations are sometimes implicit, as in the declaration of *fst* and *snd*, each of which maps a *Card* to one *Key*, and sometimes explicit using keywords, as in the declaration of *key* in *Room*, which for a given room, maps one element of *Key* to each element of *Time*. The keyword *lone* means at most one (and can be read "less than or equal to one"), so the declaration of *prev* says that, for a given *Desk*, and at a given *Time*, each *Room* is associated with at most one *Key*.
- The dot operator is relational join. Scalars are treated semantically as singleton sets, and sets are treated as unary relations. Thus *cards.t* is the relation that associates elements of *Guest* with elements of *Card* at time *t*, *c.fst* is the first key of card *c*, and *r.key.t* is the current key of room *r* at time *t*.
- The arrow operator -> is a cartesian product, and is used in the operations to form tuples; + is union; - is difference; and ++ is relational override.

Importing the built-in ordering module (fig. E.1, line 2) introduces a total ordering on time steps, the elements of the signature *Time*. The *Traces* fact (fig. E.2, line 20) constrains the ordering so that the initialization condition holds in the first state, and so that any state (except the last) and its successor are related by either the *checkin* or the *enter* operation.

The assertion (26) claims that three *enter* events cannot occur in sequence for the same room, with the intervening one performed by one guest, and the first and third by another. In other words, two guests can't use the same room at the same time.

The check command for this assertion instructs the analyzer to consider all traces involving 3 cards, 3 keys, 6 time instants, one room and two guests. Executing it produces a counterexample trace in 2 seconds (on a PowerMac G5), consisting of the following steps (shown graphically in the visualizer's output of fig. E.3):

- Initially, the room *Room0* holds key *Key0* in its lock, and the desk associates the room with the key, but holds no record of previously issued keys. Note that the room has been marked with the label *NoIntruder_r*: it will be the witness to the violation of the assertion *NoIntruder*, corresponding to the quantified variable *r*.
- In the second state, following a *checkin*, *Guest0* has acquired a card whose first and second keys are *Key0* and *Key1* respectively, and the desk has recorded *Key0* as issued. Note that the guest has been labeled *NoIntruder_g*, indicating that this guest will be the witness playing the role of the variable *g* in the assertion.
- In the third state, following another *checkin*, a second guest, *Guest1*, has acquired a card whose first and second keys are *Key1* and *Key0* respectively—the same keys as *Guest0*, but in a different order—and the desk has recorded *Key1* as issued. This new guest has been marked with the label *NoIntruder_g'*, indicating that it will be the witness playing the role of the variable *g'* in the assertion—the intruder.
- In the fourth state, the first entry has occurred—of *Guest0*—and the room key has been changed to *Key1*.
- In the fifth state, the second, illegal, entry has occurred—of *Guest1*—and the room key has been changed back to *Key0*.
- In the sixth and final state, the third entry has occurred—of *Guest0* again—and the room key has been changed back to *Key1*.

The fault lies in the initial condition. Because *Key0*, the initial key of *Room0*, was not recorded as having been issued, it was possible to issue it twice, thus setting up the cycle. The keys already in the locks should have been recorded as issued initially:

```
pred init (t: Time) {
   Desk.prev.t = key.t
   Desk.issued.t = Room.key.t and no cards.t
   }
```

With this change, the analysis exhausts the entire space without finding a counterexample. For greater confidence, we can increase the scope. Extending the scope to 4 keys and cards, 7 time instants, two guests and one room

```
check NoIntruder for 4 but 7 Time, 2 Guest, 1 Room
```

reveals another counterexample, in which a guest checks in twice, with another guest checking in between the two. These two guests can then

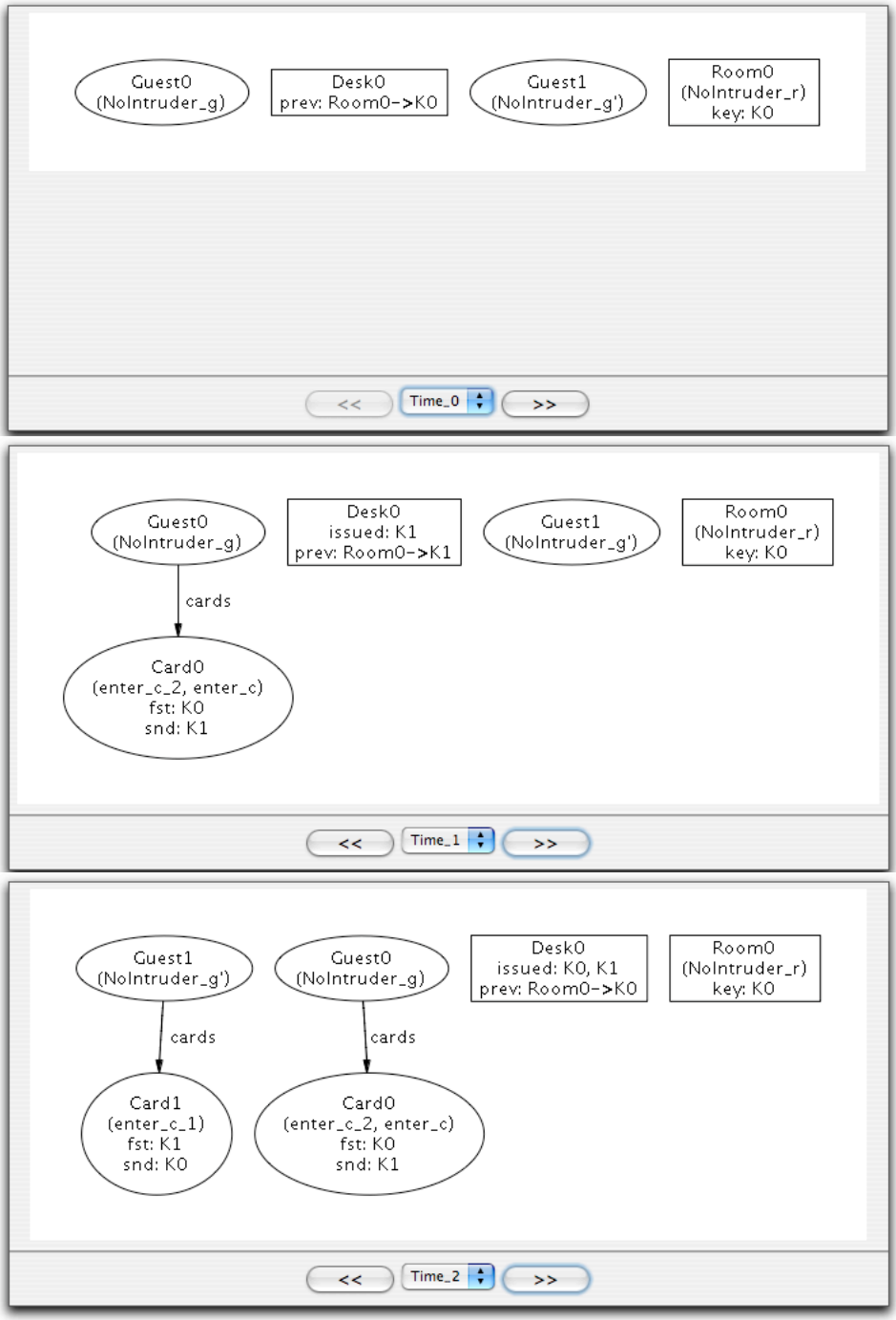
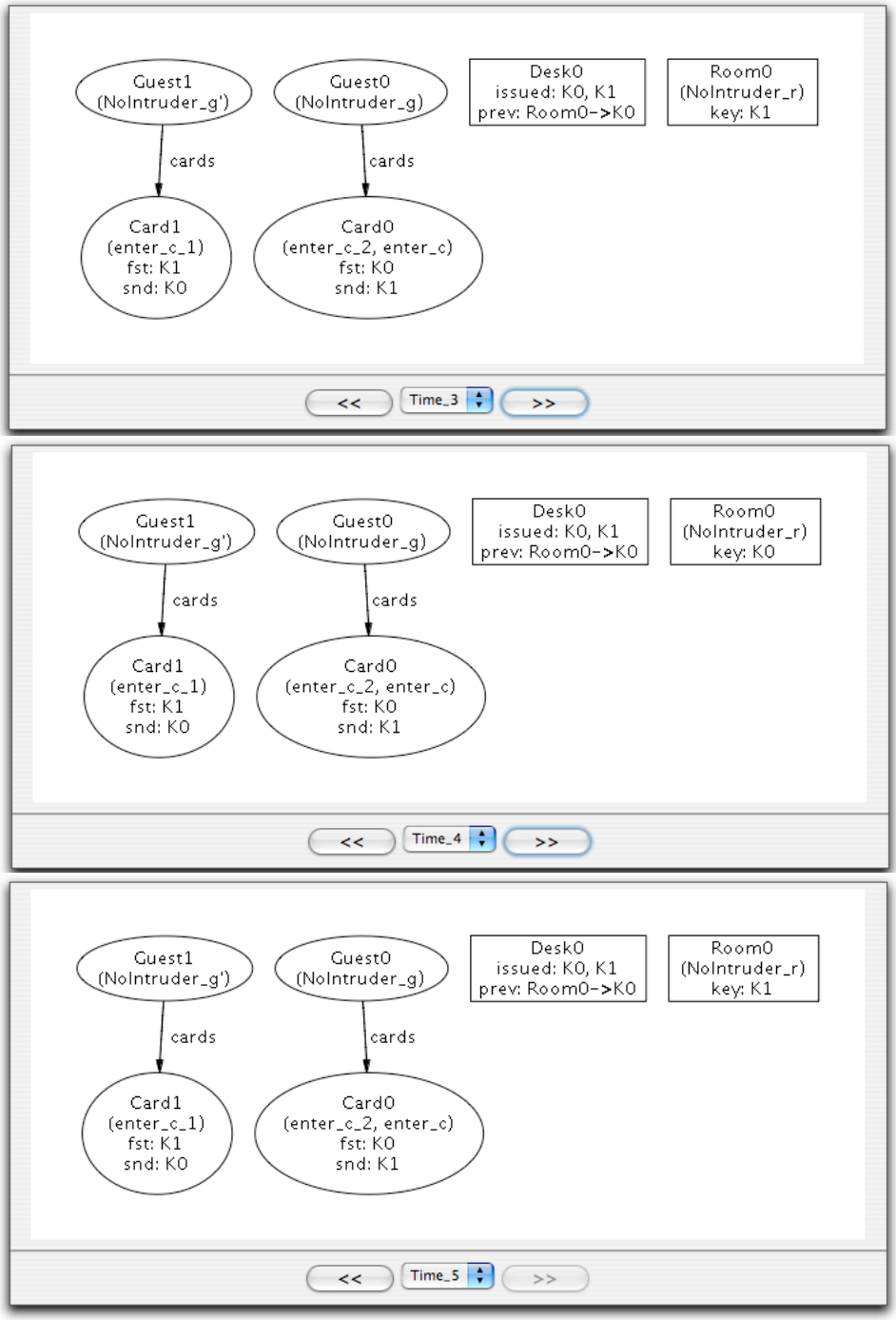


FIG. E.3 Counterexample to assertion of fig. E.2. Each panel corresponds to a state; execution begins in the top left, and continues from the bottom of the left-hand to the top of the right-hand page.


Guest1
(NoIntruder_g')
Guest0
(NoIntruder_g)
Desk0
issued: K0, K1
prev: Room0->K0
Room0
(NoIntruder_r)
key: K1
cards
cards
Card1
(enter_c_1)
fst: K1
snd: K0
Card0
(enter_c_2, enter_c)
fst: K0
snd: K1
<<
Time_3
>>
Guest1
(NoIntruder_g')
Guest0
(NoIntruder_g)
Desk0
issued: K0, K1
prev: Room0->K0
Room0
(NoIntruder_r)
key: K0
cards
cards
Card1
(enter_c_1)
fst: K1
snd: K0
Card0
(enter_c_2, enter_c)
fst: K0
snd: K1
<<
Time_4
>>
Guest1
(NoIntruder_g')
Guest0
(NoIntruder_g)
Desk0
issued: K0, K1
prev: Room0->K0
Room0
(NoIntruder_r)
key: K1
cards
cards
Card1
(enter_c_1)
fst: K1
snd: K0
Card0
(enter_c_2, enter_c)
fst: K0
snd: K1
<<
Time_5
>>

perform the 3 entries in violation of the assertion. We can fix this problem by only allowing guests to check in if they have returned cards they used previously. This can be modeled by changing one line of the *checkin* operation from

```
cards.t' = cards.t + g->c
```

to

```
cards.t' = cards.t ++ g->c
```

where the override operator now causes the guest's set of cards to be replaced, rather than augmented, by the new one. Now no counterexample is found, and we can increase the scope yet further for more confidence. With at most 6 cards and keys, 12 time instants, and 3 guests and rooms

```
check NoIntruder for 6 but 12 Time, 3 Guest, 3 Room
```

the space is exhausted in just under a minute. Of course, we have not *proved* the assertion to hold, and it is possible (though unlikely) that there is a counterexample in a larger scope. In a critical setting, it might make sense to attempt to prove the assertion at this point. Theorem provers can be applied to all of the approaches discussed here, even though our focus is on more lightweight tools. B and Z in particular are supported by readily available proof tools that are tailored to their particular forms.

E.2 B

B was designed by Jean-Raymond Abrial, one of the earliest contributors to Z. It comprises a language (AMN) and a method for obtaining implementations from abstract models by stepwise refinement. Starting with a very abstract machine, details are added one layer at a time, until a machine is obtained that can be translated directly into code. If each refinement step is valid, the resulting code is guaranteed to meet the top-level specification.

B is aimed primarily at the development of critical systems, and has been applied on a number of industrial projects. Its best known application to date was in a braking system for the Paris Metro.

The standard reference is Abrial's book [1]. More introductory texts are available [63, 80, 48], as well as a collection of case studies [64].

E.2.1 Modeling Notions of B

B's specification language, *Abstract Machine Notation* (AMN), reveals its focus in its name: a system is viewed (as in VDM and Z) as a state machine with operations over a global state. A model consists of a series of set declarations (akin to Alloy's signatures or Z's given sets); declarations of state components (called "variables"); an invariant on the state; an initialization condition; and a collection of operations.

State components are structured with sets and relations, as in Z; unlike in Alloy, higher-order structures are permitted. AMN does not separate type constraints from other, more expressive, invariants, so type checking has a heuristic flavor.

As in VDM, the precondition of an operation is explicit. In contrast to all the other approaches, the postcondition is not given as a logical formula, but as a collection of *substitutions*. A substitution is like an assignment statement, and can change the entire value of a state variable or update the value of a relation at a particular point. To partially constrain a state variable, one can assign to it an arbitrary value drawn from a set characterized by a formula.

The rationale for this style of specification is that it makes theorem proving easier: in manipulating operations syntactically, the postcondition can be treated literally as a substitution. The more operational style is also more familiar to programmers, and it makes more explicit the presence of non-determinism. Being programmatic in style, it is also more readily converted into an imperative program. The drawback is less flexibility in comparison to the other languages, and less support for incrementality: you have to give a substitution for every state variable (where, in Alloy for example, you can simulate an operation when constraints have been written for only some of the state components).

E.2.2 Sample Model in B

A version of the hotel locking model in B is shown in 3 figures: the top-level abstract machine in fig. E.4, and a refinement in figs. E.5 and E.6.

The abstract model (fig. E.4) has only a single state component—the room occupancy roster—which is updated by the *Checkin* operation, and guards the *Enter* operation. The special symbol +-> indicates a partial function; B and Z use a collection of special arrows in place of the multiplicity markings of Alloy and OCL.

```
MACHINE hotel1

SETS
   GUEST = {g1,g2} ;
   ROOM = {r1,r2}

VARIABLES  alloc

INVARIANT
   alloc : ROOM +-> GUEST

INITIALISATION  alloc := {}

OPERATIONS

CheckIn(g,r) =
   PRE
      g:GUEST  &  r:ROOM
   THEN
      SELECT
         r /: dom(alloc)
      THEN
         alloc(r) := g
      END
   END ;

Enter(g,r) =
   PRE g:GUEST & r:ROOM THEN
      SELECT
         r |->g : alloc
      THEN
         skip
      END
  END ;
```

FIG. E.4 B model: most abstract machine.

B makes a distinction between the precondition of an operation and its *guard*. When invoked in a state in which the guard is false, an operation blocks; in contrast, an operation should never be invoked unless the precondition holds (and if invoked, any outcome may result). For the *Enter* operation, for example, the precondition says that the arguments should be a guest and a room; the guard says that the operation cannot proceed unless the guest is in the occupancy roster for that room.

The basic sets are given particular values in this specification to set bounds for analysis with the ProB tool. This is just like an Alloy scope specification, but is set globally rather than on a command-by-command basis.

The refined model in figs. E.5 and E.6 has exactly the same structure. The claim that this model, *hotel2*, refines the more abstract one, *hotel1*, is an assertion to be checked by a tool.

In this model, the state is more complex, since it includes the mechanism with cards and locks. The state is described as a collection of sets and relations, as in Alloy, OCL and Z. The expression *POW(e)* denotes the powerset of *e*—the set of sets of elements drawn from *e*—and the colon in each declaration denotes set membership. A declaration such as

```
key: POW(KEY)
```

is thus equivalent to the Alloy declaration

```
key: set KEY
```

even the right-hand expression is higher-order in B but not in Alloy. The arrow symbols >-> and –> denote injective and total functions respectively. The constraint of line 13 says that the first and second keys of a given card must be distinct. Including this as an invariant means that the operations are expected to preserve it. Although semantically this invariant is treated no differently from the declarations of *ckey1* and *ckey2* that precede it, type checking distinguishes them, and will fail if their order is reversed, with the invariant placed before the declarations.

Declaring *ckey1* and *ckey2* as state variables means that an operation can be defined that changes the keys on a card. They might have been declared instead as constants (as in the Alloy, OCL, and VDM models), which would rule this out.

The initialization condition illustrates non-determinism. The *ANY* clause binds an arbitrary set of keys to *ks*, and an arbitrary function from rooms to keys to *f*; the arrow symbol in the declaration of this function makes it injective, ensuring that no key is assigned to more than one room. The body of the clause assigns the set of keys to *key*, and the function to *lock* and *prev*. Note how the assignment of the non-deterministically chosen *f* to these two variables has the same effect as the equality

```
Desk.prev.t = key.t
```

```
REFINEMENT hotel2
REFINES hotel1
SETS
   KEY = {k1,k2,k3,k4} ;
   CARD = {c1,c2,c3}
VARIABLES
   alloc, key, cArd, ckey1, ckey2, lock, prev, guest
INVARIANT
   key : POW(KEY) &
   cArd : POW(CARD) &
   ckey1 : cArd >-> key &
   ckey2 : cArd >-> key &
   !c.(c: cArd => ckey1(c) /= ckey2(c)) &
   guest : cArd -> GUEST &
   lock : ROOM >-> key &
   prev : ROOM >-> key
INITIALISATION
   ANY ks, f  WHERE
      ks : POW(KEY) &
      f : ROOM >-> ks
   THEN
      key  := ks ||
      lock := f ||
      prev := f  ||
      cArd, ckey1, ckey2, guest, alloc := {}, {}, {}, {}, {}
   END
```

FIG. E.5 B model: state and initialization for refined machine.

in the Alloy model, ensuring that the room-key record at the front desk matches the keys of the actual locks, whatever it may be.

In this refined model, the *Entry* operation is split in two: *Enter1* for the normal case, and *Enter2* for the case in which the lock is recoded. In the other approaches, this is expressed with disjunction; in B, a non-deterministic choice operator could be used to the same effect.

```
OPERATIONS

CheckIn(g,r) =
   PRE g:GUEST & r:ROOM THEN
      ANY c, k WHERE
         r : ROOM & r /: dom(alloc) &
         c : CARD  &  c /: cArd &
         k : KEY  &  k /: key
      THEN
         ckey1(c) := prev(r)  ||
         ckey2(c) := k  ||
         guest(c) := g  ||
         prev(r) := k  ||
         key := key \/ {k} ||
         cArd := cArd \/ {c}   ||
         alloc(r) := g
      END
   END ;

Enter1(g,r) =
   PRE g:GUEST & r:ROOM THEN
      ANY c, k WHERE
         c:CARD & k:KEY &
         c |-> g : guest &
         ckey1(c) = lock(r)
      THEN
         lock(r) := ckey2(c)
      END
   END ;

Enter2(g,r) =
   PRE g:GUEST & r:ROOM THEN
      ANY c, k WHERE
         c:CARD & k:KEY &
         c |-> g : guest &
         ckey2(c) = lock(r)
      THEN
         skip
      END
   END ;
```

FIG. E.6 B model: operations for refined machine.

E.2.3 Tools for B

Two commercial tools are available for B: Atelier-B from Steria, and the B-Toolkit from B-Core. Both focus on theorem proving and code generation, but also provide an animator for lightweight analysis.

ProB [51] is a very different tool. It offers very similar functionality to the Alloy Analyzer; of all the tools associated with these alternative approaches, it is the only one that can generate counterexamples to assertions fully automatically. B does not have a facility for defining arbitrary assertions, so ProB focuses on checking the proof obligations that are generated by invariants and refinement claims. Refinement is checked over traces rather than inductively over operations, so the user need not find an inductive invariant. ProB can also check the refinement relationship between a B model and a more abstract description written in the CSP process algebra.

E.3 OCL

OCL, the Object Constraint Language, is the constraint language of UML. It was developed by Jos Warmer and Anneke Klepper, based on Steve Cook and John Daniels's Syntropy language [11] and on modeling work done at IBM. Their book [77] provides an accessible overview. As part of UML, the language is an Object Management Group standard; the most recent specification is available online [57].

The early design of OCL placed less emphasis on precise semantics than the other approaches. Many researchers, in particular those associated with the Precise UML Group, worked to produce a formal semantics for OCL, but since the language was already standardized, it was too late to eliminate its complexities. So although OCL was designed in the hope that it would be simpler than languages such as VDM and Z, it actually ended up more complicated.

Our discussion is based on a variant of OCL designed by Mark Richters and Martin Gogolla [61, 60]. It has a formal semantics; a type system that supports subtyping; and a powerful animator and testing tool called USE.

When OCL was brought into the UML standard, it was viewed as an annotation language for UML class diagrams, so it was not given its own textual notation for declarations. This means that an OCL model,

according to the standard, would have to include a UML diagram for the declarations of classes and relations—an inconvenience, especially for small models. The USE variant of OCL includes a textual notation for declarations, and thus overcomes this problem.

E.3.1 Basic Notions of OCL

An OCL model consists of a description of a state space (given in terms of classes, attributes and associations), some invariants, and a collection of operations. As in VDM, operations separate pre- and postconditions, and include invariants implicitly. In addition, however, OCL allows arbitrary predicates to be packaged, and in this respect, it has more in common with Alloy; the idiom used in the Alloy model, with explicit time instants, for example, can be cast fairly easily into OCL.

Like Alloy, OCL models the state with a collection of sets and relations. Surprisingly, however, a something-to-many relation, mapping an atom to more than one atom, is treated semantically not as a flat relation but rather as a function to sets, resulting in a model whose style is more reminiscent of VDM than of Alloy or Z. This gives a strong directionality to the relations of OCL; they cannot be traversed backwards. An association is thus accessed not as a single relation, but as a pair of relations derived from it called *roles*, one for navigation in each direction.

The multiplicity of a role is part of its type. Navigation is function application, and results in a set or scalar depending on the multiplicity (in contrast to Alloy, in which navigation is relational image, and always yields a set). The advantage of this is that the type checker can detect errors in which a navigation assumes a role to have a multiplicity incompatible with its declaration. The disadvantages are that multiplicities behave differently from explicit constraints that say the same thing; changing a role's multiplicities alters its type, and may require compensating changes where it is used; and conversions are needed between sets and scalars.

OCL has no transitive closure operator. To allow a multi-step navigation through a relation, therefore, it allows predicates and functions to be defined recursively. This brings useful expressiveness, but it has a downside: predicates no longer have a simple logical interpretation, but require a least fixpoint semantics. As a result, an OCL tool can't use a constraint solver in the style of the Alloy Analyzer or ProB, since it will generate spurious cases corresponding to non-minimal solutions.

In two respects, OCL is very different from the other approaches. First, its syntax stacks variable bindings in the style of Smalltalk, and treats the first argument of operators as privileged. Appropriately, it has a notion of *context* within which references to an archetypal member of a class are implicit. Second, like an object-oriented programming language, OCL distinguishes a class from the set of objects associated with it. This makes reflection possible, which is useful for metamodeling, but it also complicates the language.

The underlying datatypes of OCL are defined in library modules, which play a similar role in OCL to the mathematical toolkit in Z. In contrast, the basic types are built into the language in Alloy, B and VDM. This decision has some subtle implications for the type system. For example, unlike Alloy's type checker, a type checker for OCL cannot exploit the meaning of the set and relational operators, but must rely on their declared type alone.

An expression's value must belong to one of the library types. Since relations are not included, this means that, in contrast to the other languages, an expression cannot denote a relation. This does not reduce the expressiveness of the language, since arbitrary quantifications are allowed, but it does make some constraints more verbose.

E.3.2 Sample Model in OCL

An OCL version of the hotel locking model is shown in figs. E.7 and E.8.

The first figure, E.7, shows the declarations of classes and associations. The class *Desk* is included to provide a context for the state component representing the set of issued keys. As in the Alloy model, there is only one instance of *Desk*; this constraint is recorded as the invariant *oneDesk*. Note the use of the expression *Desk.allInstances*, meaning the set of instances of the class *Desk*; it would be illegal to write

```
Desk->size=1
```

instead because *Desk* denotes a class and not a set.

An association is a relation, and it may have any arity, as indicated by the number of roles: *fst* and *snd* have two roles and are binary, for example, whereas *prev* is ternary. This model follows the convention that a role of an association *a* that maps to instances of class *c* is named *c4a*.

For a binary association, the two roles are just binary relations that are transposes of one another. For a ternary relation, however, a role denotes a pair of binary relations, one for each possible source of a navi-

gation (there being two other classes involved); and, in general, for an association with *k* roles, each role denotes *k*-1 distinct binary relations, with the appropriate relation selected according to the context.

Given a desk *d*, for example, the expression *d.key4prev* denotes the set of keys held at desk *d* as previous keys of some room; likewise *d.room4prev* denote the sets of rooms that have previous keys associated with them at desk *d*.

Roles give only a simplified view of higher-arity relations, which is not fully expressive. If there were more than one desk, one could not write an expression like Alloy's *d.prev[r]* for the previous key of room *r* at desk *d*. Fortunately, there is only a single desk, so the problem does not arise. When a truly higher-arity relation is needed in OCL, a different approach must be used, in which the relation is represented explicitly as a set of tuples. Richters explains this in section 4.9.2 of his thesis [60].

The second figure, E.8, shows the declaration of the class *Room*, and the definitions of the operations for checking in and entering a room. The operations are declared within the context of the *Room* class; this gives each an implicit argument that can be referred to by the keyword *self*, and which, unlike Alloy's *this*, can be omitted. The expression

```
self.key4prev
```

on line 5, for example, denoting the previous key associated with the room in context, could be written instead as just *key4prev*—a shorthand not available in Alloy, since it would denote the relation as a whole.

Each operation has preconditions and postconditions that can be broken into separate, named clauses to allow a tool to give feedback about which clause is violated when checking a test case against the model. In a postcondition, roles and attributes that refer to values in the prestate are marked with the suffix *@pre*. The constraint

```
g.card4cards = g.card4cards@pre->including(c) and
```

for example, says that the set of cards associated with the guest *g* in the poststate is the set in the prestate with the card *c* added.

The constraint

```
self.key4prev=Set{c.key4snd}
```

on line 20 in the postcondition of *checkin* says that, in the poststate, the previous key is recorded to be the second key of the card. The *Set* keyword lifts the element *c.key4snd* to a set. You might think it's not nec-

```
model hotel

class Key end
class Card end
class Guest end

class Desk end
constraints
context Desk inv oneDesk: Desk.allInstances->size=1

association fst between
    Card [*] role card4fst
    Key [1] role key4fst
end

association snd between
    Card [*] role card4snd
    Key [1] role key4snd
end

association key between
    Room [*] role room4key
    Key [1] role key4key
end

association prev between
    Desk [*] role desk4prev
    Room [*] role room4prev
    Key [0..1] role key4prev
end

association issued between
    Desk [*] role desk4issued
    Key [*] role key4issued
end

association cards between
    Guest [*] role guest4cards
    Card  [*] role card4cards
end
```

FIG. E.7 OCL model: state declaration.

```
class Room
operations
    init()
    post prev_eq_key:
        self.key4prev = Set{self.key4key}
    post issued_eq_room_key:
        Desk.allInstances.key4issued = Room.allInstances.key4key
    post no_cards:
        Card.allInstances.guest4cards->isEmpty

    checkin(g:Guest)
    pre key_exists:
        Key.allInstances->exists(k| Desk.allInstances.key4issued->excludes(k))
    post fst_snd_ok_cards_issued_prev_updated:
        Card.allInstances->exists(c|
            self.key4prev->includes(c.key4fst) and
            Desk.allInstances.key4issued->excludes(c.key4snd) and
            g.card4cards = g.card4cards@pre->including(c) and
            Desk.allInstances->forAll(d|
                d.key4issued = d.key4issued@pre->including(c.key4snd)) and
            self.key4prev = Set{c.key4snd})
    post key_unchanged:
        self.key4key@pre = self.key4key

    enter(g:Guest)
    pre key_matches:
        g.card4cards->exists(c|
            let k = key4key in c.key4snd = k or c.key4fst = k
    post key_updated:
        g.card4cards->exists(c|
            let k = key4key in
                (c.key4snd = k and self.key4key = self.key4key@pre) or
                (c.key4fst = k and self.key4key = c.key4snd))
    post issued_unchanged:
        Desk.allInstances->forAll(d|d.key4issued@pre = d.key4issued)
    post prev_unchanged:
        Room.allInstances->forAll(r|
            self.desk4prev@pre = self.desk4prev
            and self.key4prev@pre = self.key4prev)
    post cards_unchanged:
        Card.allInstances->forAll(c|c.guest4cards@pre = c.guest4cards)

end
```

FIG. E.8 OCL model: operations.

essary here, since the roles *key4prev* and *key4snd* have multiplicities of *[0..1]* and *[1]* respectively, which are type compatible. For a ternary relation, however, the multiplicity of a role *r* does not indicate the size of the set that *x.r* might represent. Rather, it indicates how many instances of that type are associated with a *combination* of instances of the other types. In this case, if there were multiple desks, *self.key4prev* might contain more than one key, despite the multiplicity, so any value equated to it must be a set and not a scalar.

E.3.3 Tools for OCL

Many tools are available for OCL. Some, such as Octopus (from Klasse Objecten, the company founded by Anneke Kleppe), are standalone tools; others are components in larger tools for model-driven development, such as the OCL component of Borland's Together Designer. Typical features are syntax and type checking, interpretation of OCL constraints over test cases, and generation of code in Java, SQL, etc, from OCL expressions.

Fewer tools support design-time analysis. The most powerful in this class seems to be the USE tool from the University of Bremen [75]. It offers an environment in which a modeler can construct test cases and evaluate OCL expressions and constraints over them. Recently, a facility for enumerating snapshots with user-provided generators has been added [20], which allows an exhaustive search over a finite space of cases in the style of Alloy. Its user interface integrates OCL with the graphical object model of UML, supporting visual editing of declarations and diagrammatic display of snapshots and executions.

With the USE tool, Martin Gogolla was able to simulate scenarios and uncover flaws, including the initialization error in the first variant of the Alloy model.

E.4 VDM

VDM stands for the "Vienna Development Method," so called because it grew out of work at IBM's Vienna Laboratory on programming language definition in the 1970's. The method, developed by Cliff Jones and Dines Bjørner, comprises a specification language and an approach to refining specifications into code. Many of the basic principles and ideas of logic-based specification first appeared in VDM.

Nowadays, the term "VDM" usually refers to the language, for which the classic reference is Jones's book [43]. The latest version of the language, VDM-SL (the VDM Specification Language), was standardized by ISO in the 1990's [49]; it has two syntaxes, one ASCII-based (used by most VDM tools), and one using special mathematical symbols.

VDM has been used in a variety of industrial settings; recent applications have included the development of electronic trading systems, secure smart cards and the Dutch flower auction system.

Two recent books explain the process of modeling in VDM; the first [18] uses the standardized language, VDM-SL, and the second [19] uses VDM^{++}, an extension that includes object-oriented features and concurrency. Both books include case studies, and stress the use of lightweight tool technology for aiding dialog between engineers and domain experts. A paper by Jones discusses the rationale behind the design of VDM [44].

E.4.1 Basic Notions of VDM

A VDM specification describes a state machine comprising a set of states and a collection of operations. The states are given by a top-level declaration and auxiliary declarations to introduce any composite types that it uses. Each declaration can be accompanied by an invariant.

Operations have separate pre- and postconditions. Each operation must be *implementable*, meaning that the postcondition admits at least one poststate for each prestate satisfying the precondition. If an operation is written in an *explicit* style (that is, with a postcondition consisting of assignments to poststate components), it will be implementable by construction. The invariants, as in Alloy, OCL and Z, are implicitly included in the pre- and postconditions. Explicit operations must be preserve invariants.

The pre- and postcondition of one operation can be used in another by *operation quotation*, which treats the operation much like a pair of Alloy predicates. *Validation conjectures* play the role of Alloy's assertions, and are formulated in a tool-specific language extension, rather than in the VDM language itself.

In contrast to Alloy, B, OCL and Z (and in common with languages aimed more at describing code interfaces, such as JML [50] and the Larch interface languages [25]), VDM has *frame conditions* indicating which state variables may be read or written by an operation. A frame condition can make an operation much more succinct (since there is

no need to mention components that don't change), and may make it easier to read at a glance. The downside is that frame conditions assume a more restrictive form of specification than languages such as Alloy and Z permit; you can't, for example, add redundant components to the state that are defined in terms of other components without changing all the operations.

E.4.2 Sample Model in VDM

A VDM version of the hotel locking model is shown in two parts: the type declarations in fig. E.9, and the operations in fig. E.10.

The type declarations begin with the declaration of *Key*, *Room* and *Guest* as "token" types, meaning that they denote sets of uninterpreted atoms. In contrast, *Card* and *Desk* are declared as record types. The special type *Hotel* corresponding to the global state is also a record type. Each type may be followed by an invariant; that of *Desk* (line 11), for example, says that the set of keys associated with rooms currently is a subset of the set of keys previously issued.

Alloy, in contrast, has no composite types (except for relations). The use of record types has benefits and drawbacks. The primary benefit is that a constructor can be used to create a fresh value (as in line 8), where Alloy requires a set comprehension or existential quantifier (as in line 2 of fig. E.2). The drawbacks are extra notation (note VDM's dot in *c.fst* but the brackets in *locks(r)*) and the problems they create for analysis.

Records can often be represented with signatures in Alloy, but the lack of constructors lies at the heart of the unbounded universals problem described in section 5.3. A record has no identity distinct from its value, so the VDM model does not distinguish two cards held by different guests that happen to have the same keys.

The more general, higher-order nature of VDM can be seen in the state invariant on line 16. The formula

```
dunion {{c.fst, c.snd} | c in set dunion rng h.guests}
  subset h.desk.issued
```

says that the first and second keys on cards held by guests must be recorded as issued at the desk. Because the expression *h.guests* is a function from guests to sets of cards, its range, *rng h.guests*, is a set of sets, which must be flattened by taking a distributed union before determining whether card *c* belongs. In Alloy, sets of sets are not expressible, and this constraint would be written instead as

```
all t: Time | Guest.cards.t.(fst + snd) in Desk.issued.t
```

The time variable *t* plays the role of the state variable *h* in the VDM specification. Its placement is an artifact of the idiom chosen, and it would precede rather than follow the field names if the state were modeled as a signature instead, as in the memory or media asset examples of chapter 6.

An operation has a listing of arguments and their types, frame conditions, a precondition and a postcondition. Note how frame conditions shorten their associated postcondition; in *Enter*, for example, because only the *locks* components is writeable, there is no need for equalities on the other components, as in lines 17 and 18 of the Alloy model of fig. E.2.

The explicit precondition makes it easier to see when an operation applies, but it can make the operation more verbose: note how the precondition of *Enter* (line 17) is repeated in the postcondition, since the existential quantifier cannot span both.

VDM's pre- and postconditions are just logical formulas, like the body of an Alloy predicate. Unlike Alloy, and like the other approaches (although to a lesser extent Z), VDM assumes a particular state machine idiom, and provides special syntax to support it. The *state* declaration, unlike the other type declarations, defines a mutable structure, whose components have separate values in the pre- and poststate of an operation. The values of a component *c* in the pre- and poststates are referred to as *c*~ and *c* respectively. The special symbol *&* separates a quantifier's binding from its body.

This is convenient but less flexible than Alloy's approach. All mutations are confined to the top-level components of the state; you could not, for example, make cards mutable in order to model modifications to existing cards by hackers (as you could in Alloy by adding a time column to the relations of *Card*). VDM^{++}, however, allows all structures to be mutable.

VDM distinguishes sets from relations. So where Alloy would use the single operator + for all unions, VDM uses *union* on sets and *munion* on maps. Being higher-order, it requires set former brackets to distinguish maps from tuples and sets from their elements. The initialization condition on line 21, for example, equates the range of the mapping from guests to sets of cards to *{{}}*—the set containing just the empty set—and writing *{}* here instead would mean something different. Simi-

```
types
    Key = token;
    Room = token;
    Guest = token;

    Card ::
        fst : Key
        snd : Key;
    Desk ::
        issued : set of Key
        prev : map Room to Key
    inv d == rng d.prev subset d.issued;

    state Hotel of
        desk : Desk
        locks : map Room to Key
        guests : map Guest to set of Card
    inv h ==
        dom h.desk.prev subset dom h.locks and
        dunion {{c.fst, c.snd} | c in set dunion rng h.guests}
            subset h.desk.issued
    init h == h.desk.issued = rng h.locks and
        h.desk.prev = h.locks and rng h.guests = {{}}
end
```

FIG. E.9 VDM model: type declarations.

larly, the expressions used to extend the maps *guests* and *locks* require set brackets for one (*{g |-> {new_c}}*) but not the other (as in *{r |-> new_k}*).

To apply an animator (such as that of the VDMTools) to an operation, it must be written in an explicit form. An example, for the *Checkin* operation, is shown in fig. E.11. The existential quantifier has been replaced by a *let* statement; the constraints of the postcondition have been replaced by assignments; and the frame condition is no longer necessary. This notation is very similar to B.

```
operations
   CheckIn(g:Guest,r:Room)
      ext wr desk : Desk
         wr guests : map Guest to set of Card
      pre r in set dom desk.prev
      post exists new_k:Key &
         new_k not in set desk~.issued and
         let new_c = mk_Card(desk~.prev(r),new_k) in
            desk.issued = desk~.issued union {new_k} and
            desk.prev = desk~.prev ++ {r |-> new_k} and
            if g in set dom guests
               then guests = guests~ ++ {g |-> guests~(g) union {new_c}}
               else guests = guests~ munion {g |-> {new_c}};
   Enter(r:Room,g:Guest)
      ext wr locks : map Room to Key
         rd guests : map Guest to set of Card
      pre r in set dom locks and g in set dom guests and
         exists c in set guests(g) & c.fst = locks(r) or c.snd = locks(r)
      post exists c in set guests(g) &
         c.fst = locks(r) and locks = locks~ ++ {r |-> c.snd} or
         c.snd = locks(r) and locks = locks~;
```

FIG. E.10 VDM model: Operations expressed implicitly.

```
   CheckInExpl: Guest * Room ==> ()
   CheckInExpl(g,r) ==
      let new_k:Key be st new_k not in set desk.issued in
      let new_c = mk_Card(desk.prev(r),new_k) in (
         desk.issued := desk.issued union {new_k};
         desk.prev := desk.prev ++ {r |-> new_k};
         guests := if g in set dom guests
            then guests ++ {g |-> guests(g) union {new_c}}
            else guests munion {g |-> {new_c}}
         )
      pre r in set dom desk.prev;
```

FIG. E.11 VDM model: Operations expressed explicitly.

E.4.3 Tools for VDM

Under the guidance of Peter Gorm Larsen, IFAD—a Danish company that offered VDM consulting in the 1990's—developed VDMTools, a toolkit for both VDM-SL and VDM^{++}. It included a type checker and theorem prover, and, for an executable subset of VDM, a facility for simulating and testing specifications, and a code generator. The toolkit is now owned by CSK Corporation of Japan. New tool support for VDM^{++} is being developed under the *Overture* open source initiative.

E.5 Z

Z was developed at Oxford University in the 1980's. It has been very influential in education and research, and has been applied successfully on several large projects, notably by Oxford University and IBM on the CICS system, in a series of projects by Praxis Critical Systems, and to the security verification of the Mondex electronic purse developed by NatWest Bank (the first ever product certified to ITSEC Level 6) [71, 70]. Z's clean and simple semantic foundation was an inspiration for the design of Alloy.

Although the language has been standardized by ISO [32], the version described in Mike Spivey's book [67] continues to be the most popular. Many books have been written about Z, including textbooks [42, 58, 78, 79], case study collections [28], and a guide to style [3].

E.5.1 Basic Notions of Z

Z, like Alloy, is at heart just a logic, augmented with some syntactic constructs to make it easy to describe software abstractions. In Alloy, these constructs are the signature, for packaging declarations, and facts/predicates/functions for packaging constraints. In Z, the same construct—the *schema* is used both to package declarations and constraints. The language of schemas, called the *schema calculus*, is rich enough to support a wide variety of idioms.

In practice, though, a particular idiom—called variously the "Oxford style," the "IBM style," and the "established strategy" [3]—has been adopted in almost all Z specifications since the earliest days. A collection of syntactic conventions have grown around it, and have become a de facto part of the language itself. The operator for combining operations by sequential composition, for example, assumes the use of this idiom; without it, the operator will not have the expected meaning.

Z, unlike B, does not have a built-in notion of refinement, and indeed many Z users view it as a system modeling language, and have no intent to prove conformance of their code to the model. There is, however, a well-established theory of refinement for Z, and the language is well-suited to developments by stepwise refinement. Woodcock's book [78] is an accessible introduction to this approach.

The sample model shown here is the first, most abstract, model in a development by refinement. Like the abstract B model of fig. E.4, it determines entry to the room by examining the room roster; in a subsequent refinement (not shown here), entry is determined by keys and locks alone. The abstract Z model does not, however, omit mention of keys and locks entirely: the recoding key on the card is selected on entry (rather than when checking in). The refinement will move this nondeterministic choice backwards in time to the checkin, with the same justification used for the example of section 6.4.6.

A Z specification is built as a series of schema declarations. A declaration has two parts: a series of variable declarations, and a predicate constraining them. When a reference is made to a previously declared schema, both its variable declarations and predicate are incorporated implicitly. A schema representing state typically builds on previous state schemas by adding new components and constraining the state further with additional invariants. A schema representing an operation may incorporate state schemas for the pre- and poststates, or it may extend a previous operation schema, adding constraints to make its behaviour more specific.

Because incorporating a schema brings the variables it declares into scope, there is often no explicit declaration for a variable that appears in a schema's predicate. As a result, Z specifications can be very succinct—sometimes mysteriously so. Exactly the same power, with the same potential for succinctness and obscurity, is found in the inheritance mechanisms of object-oriented programming languages, and in Alloy's signature extension mechanism (which was, incidentally, designed explicitly to support schema-style structuring).

In the Oxford style, state invariants are declared with the state declarations and are thus incorporated implicitly into operations, as in VDM, and in marked contrast to B, where invariants must be shown to be preserved by operations. Preconditions of operations are not separated from postconditions. It is regarded as good style for the precondition to appear explicitly in the operation schema, but it is not necessary. In place of the implementability check of VDM, a Z specifier *derives*

a precondition from an operation schema and compares it to the one expected.

Sets and relations are the predominant datatypes in Z. In this respect, Z is similar to B—which is not surprising, since B's inventor, Jean-Raymond Abrial, was one of the early developers of Z (and had worked before that on database semantics). For both Z and B, sets are seen as fundamental and relations as derived; Z is so named because of its roots in ZF (Zermelo-Fraenkel) set theory. Alloy is also based on sets and relations, but its logic is more influenced by the relational formalisms of Tarski's calculus [72] and Codd's relational database model [10], with sets regarded as a special case of relations.

Like VDM, however, Z does include record types. The same schema construct that is used syntactically for grouping declarations together can be used semantically to declare a 'schema type', whose values are bindings of values to field names. Schema types are more powerful than Alloy's signatures, because they provide constructors. Unlike signatures, however, schemas have no subtyping. One schema can be defined as an extension of another schema, but the types of the two schemas are unrelated. For example, if you declared a schema for a file system object, and extended it into two other schemas corresponding to files and folders, you would not be able to insert an instance of the file or folder schema into a set or relation declared over file system objects.

Z has a distinctive appearance, with boxes drawn around schemas, and its own collection of mathematical symbols. Here we use the "horizontal form," which, although less elegant, can be produced without special layout tools.

E.5.2 Sample Model in Z

A Z specification of the hotel locking problem is shown in figs. E.12, E.13 and E.14.

The first figure (E.12) shows the declaration of the state and initialization. Guests, keys and rooms are declared as *given sets*: uninterpreted sets of atoms that become the basis for type checking. A global variable *initkeys* is declared that represents the function associating room locks with the initial values of their keys. A *Msg* datatype is declared to represent the possible outcomes of operations.

Hotel is our first schema declaration. It introduces 3 variables that will represent the state components of the system: *firsttime*, a set of rooms; *key*, a function from rooms to keys (representing the keys held in their

```
[Guest, Key, Room]

initkeys: Room  Key;

Msg ::= okay
   | room_already_allocated
   | guest_already_registered
   | room_not_allocated
   | wrong_guest
   | key_not_fresh

Hotel  [
   firsttime:  Room
   key: Room  Key
   guest: Room  Guest
   ]

InitHotel  [
   Hotel'
   |
   firsttime' =
   key' = initkeys
   guest' =
   ]
```

FIG. E.12 State and initialization in Z.

locks in a particular state); and *guest*, a function from rooms to guests (representing the occupancy roster). The kind of arrow indicates the multiplicities: that *key* is a partial injection, and *guest* is a partial function.

This model, because it is the first in a development by refinement, will describe exactly when an entry should be permitted, and when a lock should be rekeyed; a later refinement would describe the mechanism by which entry is determined by checking keys. This explains the *firsttime* component, which did not appear in the Alloy model, but is used to mark the set of rooms which, when subsequently entered, should have their locks recoded (since a new guest will be entering for the first time).

```
Checkin0  [
   Δ Hotel
   g?: Guest
   r?: Room
   |
   r?  dom guest
   g?  ran guest
   firsttime′ = firsttime  {r?}
   key′ = key
   guest′ = guest  {r?  g?}
 ]

EnterFst  [
   Δ Hotel
   g?: Guest
   r?: Room
   k?: Key
   |
   r?  firsttime
   r?  dom guest
   guest r? = g?
   k?  ran key
   firsttime′ = firsttime \ {r?}
   key′ = key  {r?  k?}
   guest′ = guest
   ]

EnterSnd  [
   Ξ Hotel
   g?: Guest
   r?: Room
   |
   r?  dom guest
   guest r? = g?
   r?  firsttime
   ]
```

FIG. E.13 Checkin and Enter operations in Z.

InitHotel, the second schema, describes the initialization. Unlike *Hotel* which included only variable declarations, this schema has both declarations and a predicate. The declarations are those of *Hotel*, imported

by mentioning the schema's name. Notice the prime mark appended to the name. This is called *decoration*; its effect is to include not exactly the declarations of *Hotel*, but versions in which each variable is likewise primed. These primed variables are used in Z to refer to the values of state components after execution of an operation (in this case, the initialization).

A schema predicate is just a constraint, formed by conjoining the constraints on each line. Each line's constraint is a simple mathematical formula, with the equals sign denoting equality (and not assignment). So there is no semantic significance to the ordering of the terms in these equations, and we could reverse each equation without changing its meaning. This specification has been written, however, in a form that suggests how it might be executed, with the primed variables on the left. This allows it to be animated using a tool such as Jaza. The same rationale explains why the initialization equates *keys′* to the previously declared function *initkeys*, just as the corresponding component was initialized in the B model (on line 23 of fig. E.5). A more traditional Z style would simply not mention *keys′*, leaving its value unconstrained.

The second figure (E.13) shows the checkin and entry operations for normal cases; the exceptional cases are described in separate operations in the next figure. There are three schemas corresponding to checking in, and two forms of entry—one which recodes the lock, and one which does not.

The first two, *Checkin0* and *EnterFst*, mention Δ *Hotel* in their declarations. This is a schema, defined implicitly by convention, that includes *Hotel* and *Hotel′*, thus introducing standard and primed versions of each state variable, to represent the state components before and after execution. The third schema, *EnterSnd*, mentions Ξ *Hotel*. This refers to a similar schema, also including *Hotel* and *Hotel′*, but additionally a constraint equating each state variable to its primed version. Its use, therefore, indicates an operation that has no effect on the state.

Each operation also declares some variables decorated with question marks. By convention, these represent input arguments; semantically, they are no different from the variables representing the state components. When an operation schema is used elsewhere, these arguments are bound by a syntactic substitution that replaces every occurrence of an argument variable in the schema with a variable name from the new context. In comparison to the explicit parameterization of Alloy, this can be a bit awkward: an expression cannot be substituted for a variable, so if no variable already exists for the actual argument, it must be de-

```
Success   [m!: Msg |   m! = okay]

EnterRoomNotAllocated   [
   Ξ Hotel
   r?: Room
   m!: Msg
   |
   r?   dom guest
   m! = room_not_allocated
   ]

EnterWrongGuest   [
   Ξ Hotel
   g?: Guest
   r?: Room
   m!: Msg
   |
   r?   dom guest
   guest r? ≠ g?
   m! = wrong_guest
   ]

Enter   EnterFst   Success   EnterSnd   Success
   EnterRoomNotAllocated   EnterWrongGuest
```

FIG. E.14 Variant operations in Z.

clared with an existential quantifier. On the other hand, when the actual and formal arguments have the same name, no substitution is necessary and the resulting text is uncluttered by argument lists.

Z does not distinguish pre- and postconditions syntactically, and there is no need to make preconditions explicit at all. It is regarded as good style to list preconditions in full, however, above the constraints of the postcondition. The precondition of *CheckinO*, for example, is that *r?* is not in the domain of *guest*, and *g?* is not in its range—that is, the room requested is not already occupied, and the guest is not already assigned to another room. This stylistic guideline is not generally checkable by simple syntactic means, since the explicit precondition might admit states for which the postcondition cannot be satisfied, so that the actual precondition is stronger. A theorem asserting that the operation has the expected precondition can be formulated. For *CheckinO*, this theorem is:

```
Theorem preCheckin0
    Hotel; g?: Guest; r?: Room |
     r?  dom guest  g?  ran guest · pre Checkin0
```

This kind of theorem is not expressible in Alloy, as explained in section 5.3. Unintentional overconstraint is mitigated instead by simulating the operation.

The operation predicates are unsurprising. *Checkin0*, for example, adds to the *guest* relation a mapping from *r?* to *g?*; *EnterFst* recodes the lock by overriding the *key* relation with a mapping from *r?* to the new key *k?*. Note that, as in Alloy and OCL, a state variable that is unmentioned is unconstrained, so if a component is unchanged, an explicit equality is needed (as in line 24).

The third figure, E.14, shows how the behavior of these operations is augmented to cover exceptional cases. The schema *Success* simply introduces an output argument *m!* and equates it to the message *okay*. The next two schemas specify the conditions under which an entry should be regarded as impermissible, because the room has not been allocated to a guest at all, or because the guest attempting entry is not the legitimate occupant. These conditions are expressed as preconditions, and are accompanied by postconditions that constrain the value of the message accordingly.

Finally, a schema is declared that brings the different cases together: *Enter* is an operation that describes all the scenarios of attempted entry to a room. Note its assembly using just disjunction and conjunction. This simplicity is a consequence of operations being no more than logical formulas. Alloy took this idea from Z, and thus supports the same kind of structuring.

E.5.3 Tools for Z

Most tool support for Z has focused on theorem proving. The most widely used proof tools are ProofPower (from Lemma 1 Ltd), and Z/Eves, a front-end to the Eves theorem prover (from ORA Canada). The sample model was analyzed with Z/Eves. The tool can calculate preconditions and perform "domain checks" (which ensure that partial functions are never applied outside their domains), as well as performing general theorem proving. Although many steps in a proof are executed automatically, complex theorems tend to require guidance from an experienced user.

A number of animators have been built for Z. The sample model was tested using Jaza [76], an animator developed by Mark Utting at the University of Waikato. Jaza can execute operations written in an explicit style, and can do a certain amount of constraint solving over small domains. The entire sample model above can be handled in Jaza. We noted how the initialization, for example, assigns the global function *initkeys* to *keys′* rather than leaving it unconstrained; this allows the initialization to be executed given a value of *initkeys* by the user. Like the USE tool and the animator of the VDMTools, Jaza can evaluate expressions, check given states against invariants and transitions against operations, and can simulate an execution trace with the user selecting operations and providing input arguments.