Out of the Tar Pit

Ben Moseley
ben@moseley.name

Peter Marks
public@indigomail.net

February 6, 2006

Abstract

Complexity is the single major difficulty in the successful development of large-scale software systems. Following Brooks we distinguish *accidental* from *essential* difficulty, but disagree with his premise that most complexity remaining in contemporary systems is essential. We identify common causes of complexity and discuss general approaches which can be taken to eliminate them where they are accidental in nature. To make things more concrete we then give an outline for a potential complexity-minimizing approach based on *functional programming* and *Codd's relational model of data*.

1 Introduction

The "software crisis" was first identified in 1968 [NR69, p70] and in the intervening decades has deepened rather than abated. The biggest problem in the development and maintenance of large-scale software systems is complexity — large systems are hard to understand. We believe that the major contributor to this complexity in many systems is the handling of *state* and the burden that this adds when trying to analyse and reason about the system. Other closely related contributors are *code volume*, and explicit concern with the *flow of control* through the system.

The classical ways to approach the difficulty of state include object-oriented programming which tightly couples state together with related behaviour, and functional programming which — in its pure form — eschews state and side-effects all together. These approaches each suffer from various (and differing) problems when applied to traditional large-scale systems.

We argue that it is possible to take useful ideas from both and that — when combined with some ideas from the relational database world —

this approach offers significant potential for simplifying the construction of large-scale software systems.

The paper is divided into two halves. In the first half we focus on complexity. In section 2 we look at complexity in general and justify our assertion that it is at the root of the crisis, then we look at how we currently attempt to understand systems in section 3. In section 4 we look at the causes of complexity (i.e. things which make understanding difficult) before discussing the classical approaches to managing these complexity causes in section 5. In section 6 we define what we mean by "accidental" and "essential" and then in section 7 we give recommendations for alternative ways of addressing the causes of complexity — with an emphasis on avoidance of the problems rather than coping with them.

In the second half of the paper we consider in more detail a possible approach that follows our recommended strategy. We start with a review of the relational model in section 8 and give an overview of the potential approach in section 9. In section 10 we give a brief example of how the approach might be used.

Finally we contrast our approach with others in section 11 and then give conclusions in section 12.

2   Complexity

In his classic paper — "No Silver Bullet" Brooks[Bro86] identified four properties of software systems which make building software hard: Complexity, Conformity, Changeability and Invisibility. Of these we believe that Complexity is the *only* significant one — the others can either be *classified as* forms of complexity, or be seen as problematic solely *because* of the complexity in the system.

Complexity is *the* root cause of the vast majority of problems with software today. Unreliability, late delivery, lack of security — often even poor performance in large-scale systems can all be seen as deriving ultimately from unmanageable complexity. The primary status of complexity as *the* major cause of these other problems comes simply from the fact that being able to *understand* a system is a prerequisite for avoiding all of them, and of course it is this which complexity destroys.

The relevance of complexity is widely recognised. As Dijkstra said [Dij97, EWD1243]:

> "...we have to keep it crisp, disentangled, and simple if we refuse to be crushed by the complexities of our own making..."

. . . and the Economist devoted a whole article to software complexity [Eco04] — noting that by some estimates software problems cost the American economy $59 billion annually.

Being able to think and reason about our systems (particularly the effects of changes to those systems) is of *crucial* importance. The dangers of complexity and the importance of simplicity in this regard have also been a popular topic in ACM Turing award lectures. In his 1990 lecture Corbato said [Cor91]:

> "The general problem with ambitious systems is complexity.",
> "...it is important to emphasize the value of simplicity and elegance, for complexity has a way of compounding difficulties"

In 1977 Backus [Bac78] talked about the "complexities and weaknesses" of traditional languages and noted:

> "there is a desperate need for a powerful methodology to help us think about programs. ... conventional languages create unnecessary confusion in the way we think about programs"

Finally, in his Turing award speech in 1980 Hoare [Hoa81] observed:

> "...there is one quality that cannot be purchased... — and that is reliability. The price of reliability is the pursuit of the utmost simplicity"

and

> "I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies. The first method is far more difficult."

This is the unfortunate truth:

<div align="center">

Simplicity is *Hard*

</div>

. . . but the purpose of this paper is to give some cause for optimism.

One final point is that the type of complexity we are discussing in this paper is that which makes large systems *hard to understand*. It is this

that causes *us* to expend huge resources in *creating and maintaining* such systems. This type of complexity has nothing to do with complexity theory — the branch of computer science which studies the resources consumed *by a machine executing* a program. The two are completely unrelated — it is a straightforward matter to write a small program in a few lines which is incredibly simple (in our sense) and yet is of the highest complexity class (in the complexity theory sense). From this point on we shall only discuss complexity of the first kind.

We shall look at what we consider to be the major common causes of complexity (things which make understanding difficult) after first discussing exactly how we normally attempt to *understand* systems.

3   Approaches to Understanding

We argued above that the danger of complexity came from its impact on our attempts to *understand* a system. Because of this, it is helpful to consider the mechanisms that are commonly used to try to understand systems. We can then later consider the impact that potential causes of complexity have on these approaches. There are two widely-used approaches to understanding systems (or components of systems):

Testing This is attempting to understand a system from the outside — as a "black box". Conclusions about the system are drawn on the basis of observations about how it behaves in certain specific situations. Testing may be performed either by human or by machine. The former is more common for whole-system testing, the latter more common for individual component testing.

Informal Reasoning This is attempting to understand the system by examining it from the inside. The hope is that by using the extra information available, a more accurate understanding can be gained.

Of the two informal reasoning is the most important by far. This is because — as we shall see below — there are inherent limits to what can be achieved by testing, and because informal reasoning (by virtue of being an inherent part of the development process) is *always* used. The other justification is that improvements in informal reasoning will lead to *less errors being created* whilst all that improvements in testing can do is to lead to *more errors being detected*. As Dijkstra said in his Turing award speech [Dij72, EWD340]:

4

> "Those who want really reliable software will discover that they must find means of avoiding the majority of bugs to start with."

and as O'Keefe (who also stressed the importance of "*understanding your problem*" and that "*Elegance is not optional*") said [O'K90]:

> "Our response to mistakes should be to look for ways that we can avoid making them, not to blame the nature of things."

The key problem with testing is that a test (of any kind) that uses one particular set of inputs tells you *nothing at all* about the behaviour of the system or component when it is given a different set of inputs. The huge number of different possible inputs usually rules out the possibility of testing them all, hence the unavoidable concern with testing will always be — *have you performed the* right *tests?*. The only certain answer you will ever get to this question is an answer in the negative — when the system breaks. Again, as Dijkstra observed [Dij71, EWD303]:

> "testing is hopelessly inadequate....(it) can be used very effectively to show the presence of bugs but never to show their absence."

We agree with Dijkstra. *Rely* on testing at your peril.

This is *not* to say that testing has no use. The bottom line is that *all* ways of attempting to understand a system have their limitations (and this includes both *informal reasoning* — which is limited in scope, imprecise and hence prone to error — as well as *formal reasoning* — which is dependent upon the accuracy of a specification). Because of these limitations it may often be prudent to employ both testing *and* reasoning together.

It is precisely *because* of the limitations of all these approaches that *simplicity* is vital. When considered next to testing and reasoning, simplicity is more important than either. Given a stark choice between investment in testing and investment in simplicity, the latter may often be the better choice because it will facilitate *all* future attempts to understand the system — attempts of any kind.

4   Causes of Complexity

In any non-trivial system there is some complexity inherent in the problem that needs to be solved. In real large systems however, we regularly encounter complexity whose status as "inherent in the problem" is open to some doubt. We now consider some of these causes of complexity.

4.1 Complexity caused by State

Anyone who has ever telephoned a support desk for a software system and been told to "try it again", or "reload the document", or "restart the program", or "reboot your computer" or "re-install the program" or even "re-install the operating system and then the program" has direct experience of the problems that $state$[1] causes for writing reliable, understandable software.

The reason these quotes will sound familiar to many people is that they are often suggested because they are often successful in resolving the problem. The reason that they are often successful in resolving the problem is that many systems have errors in their handling of state. The reason that many of these errors exist is that the presence of state makes programs hard to *understand*. It makes them complex.

When it comes to state, we are in broad agreement with Brooks' sentiment when he says [Bro86]:

> "From the complexity comes the difficulty of enumerating, much less understanding, all the possible states of the program, and from that comes the unreliability"

— we agree with this, but believe that it is the presence of many possible states which gives rise to the complexity in the first place, and:

> "computers...have very large numbers of states. This makes conceiving, describing, and testing them hard. Software systems have orders-of-magnitude more states than computers do."

4.1.1 Impact of State on Testing

The severity of the impact of *state* on testing noted by Brooks is hard to over-emphasise. State affects all types of testing — from system-level testing (where the tester will be at the mercy of the same problems as the hapless user just mentioned) through to component-level or unit testing. The key problem is that a test (of any kind) on a system or component that is in one particular *state* tells you *nothing at all* about the behaviour of that system or component when it happens to be in another *state*.

The common approach to testing a stateful system (either at the component or system levels) is to start it up such that it is in some kind of "clean" or "initial" (albeit mostly *hidden*) state, perform the desired tests using the

---

[1]By "state" we mean *mutable state* specifically — i.e. excluding things like (immutable) single-assignment variables which are provided by logic programming languages for example

test inputs and then rely upon the (often in the case of bugs ill-founded) assumption that the system would perform the same way — *regardless of its hidden internal state* — every time the test is run with those inputs.

In essence, this approach is simply sweeping the problem of state under the carpet. The reasons that this is done are firstly because it is often possible to get away with this approach and more crucially because there isn't really any alternative when you're testing a stateful system with a complicated internal hidden state.

The difficulty of course is that it's not *always* possible to "get away with it" — if some sequence of events (inputs) can cause the system to "get into a bad *state*" (specifically an internal hidden state which was *different* from the one in which the test was performed) then things can and do go wrong. This is what is happening to the hypothetical support-desk caller discussed at the beginning of this section. The proposed remedies are all attempts to force the system back into a "good internal state".

This problem (that a test in one *state* tells you *nothing at all* about the system in a different *state*) is a direct parallel to one of the fundamental problems with testing discussed above — namely that testing for one *set of inputs* tells you *nothing at all* about the behaviour with a different *set of inputs*. In fact the problem caused by state is typically worse — particularly when testing large chunks of a system — simply because even though the number of possible *inputs* may be very large, the number of possible *states* the system can be in is often even larger.

These two similar problems — one intrinsic to testing, the other coming from *state* — combine together *horribly*. Each introduces a huge amount of uncertainty, and we are left with very little about which we *can* be certain if the system/component under scrutiny is of a stateful nature.

4.1.2 Impact of State on Informal Reasoning

In addition to causing problems for understanding a system from the outside, state also hinders the developer who must attempt to *reason* (most commonly on an informal basis) about the expected behaviour of the system "from the inside".

The mental processes which are used to do this informal reasoning often revolve around a case-by-case mental simulation of behaviour: "if this variable is in this state, then this will happen — which is correct — otherwise that will happen — which is also correct". As the number of states — and hence the number of possible scenarios that must be considered — grows, the effectiveness of this mental approach buckles almost as quickly as test-

ing (it does achieve some advantage through abstraction over sets of similar values which can be seen to be treated identically).

One of the issues (that affects both testing and reasoning) is the exponential rate at which the number of possible states grows — for every single *bit* of state that we add we *double* the total number of possible states. Another issue — which is a particular problem for informal reasoning — is *contamination*.

Consider a system to be made up of procedures, some of which are stateful and others which aren't. We have already discussed the difficulties of understanding the bits which are stateful, but what we would hope is that the procedures which aren't themselves stateful would be more easy to comprehend. Alas, this is largely not the case. If the procedure in question (which is itself stateless) makes use of any other procedure which *is* stateful — *even indirectly* — then all bets are off, our procedure becomes *contaminated* and we can only understand it in the context of state. If we try to do anything else we will again run the risk of all the classic state-derived problems discussed above. As has been said, the problem with state is that "*when you let the nose of the camel into the tent, the rest of him tends to follow*".

As a result of all the above reasons it is our belief that the single biggest remaining cause of complexity in most contemporary large systems is *state*, and the more we can do to *limit* and *manage* state, the better.

4.2 Complexity caused by Control

Control is basically about the *order* in which things happen.

The problem with control is that very often we do not want to have to be concerned with this. Obviously — given that we want to construct a real system in which things will actually *happen* — at some point order is going to be relevant to someone, but there are significant risks in concerning ourselves with this issue unnecessarily.

Most traditional programming languages *do* force a concern with ordering — most often the ordering in which things will *happen* is controlled by the order in which the statements of the programming language are written in the textual form of the program. This order is then modified by explicit branching instructions (possibly with conditions attached), and subroutines are normally provided which will be invoked in an implicit stack.

Of course a variety of evaluation orders is possible, but there is little variation in this regard amongst widespread languages.

The difficulty is that when control is an implicit part of the language (as

it almost always is), then every single piece of program must be understood in that context — even when (as is often the case) the programmer may wish to say nothing about this. When a programmer is forced (through use of a language with implicit control flow) to specify the control, he or she is being forced to specify an aspect of *how* the system should work rather than simply *what* is desired. Effectively they are being forced to *over-specify* the problem. Consider the simple pseudo-code below:

```
a := b + 3
c := d + 2
e := f * 4
```

In this case it is clear that the programmer has no concern at all with the order in which (i.e. *how*) these things eventually happen. The programmer is only interested in specifying a *relationship* between certain values, but has been forced to say more than this by choosing an arbitrary control flow. Often in cases such as this a compiler may go to lengths to establish that such a requirement (ordering) — which the programmer has been forced to make because of the semantics of the language — can be safely ignored.

In simple cases like the above the issue is often given little consideration, but it is important to realise that two completely unnecessary things are happening — first an artificial ordering is being imposed, and then further work is done to remove it.

This seemingly innocuous occurrence can actually significantly complicate the process of informal reasoning. This is because the person reading the code above must effectively duplicate the work of the hypothetical compiler — they must (by virtue of the definition of the language semantics) start with the assumption that the ordering specified *is* significant, and then by further inspection determine that it is not (in cases less trivial than the one above determining this can become very difficult). The problem here is that mistakes in this determination can lead to the introduction of very subtle and hard-to-find bugs.

It is important to note that the problem is not in the *text* of the program above — after all that does have to be written down in some order — it is solely in the *semantics* of the hypothetical imperative language we have assumed. It *is* possible to consider the exact same program text as being a valid program in a language whose semantics did not define a run-time sequencing based upon textual ordering within the program[2].

---

[2]Indeed early versions of the Oz language (with *implicit* concurrency at the statement level) were somewhat of this kind [vRH04, p809].

Having considered the impact of control on *informal reasoning*, we now look at a second control-related problem, concurrency, which affects *testing* as well.

Like basic control such as branching, but as opposed to sequencing, concurrency is normally specified *explicitly* in most languages. The most common model is "shared-state concurrency" in which specification for explicit synchronization is provided. The impacts that this has for informal reasoning are well known, and the difficulty comes from adding further to the *number of scenarios* that must mentally be considered as the program is read. (In this respect the problem is similar to that of state which also adds to the number of scenarios for mental consideration as noted above).

Concurrency also affects testing, for in this case, we can no longer even be assured of result consistency when repeating tests on a system — even if we somehow ensure a consistent starting state. Running a test in the presence of concurrency with a known initial state and set of inputs tells you *nothing at all* about what will happen the next time you run that very same test with the very same inputs and the very same starting state... and things can't really get any worse than that.

4.3   Complexity caused by Code Volume

The final cause of complexity that we want to examine in any detail is sheer code volume.

This cause is basically in many ways a secondary effect — much code is simply concerned with managing *state* or specifying *control*. Because of this we shall often not mention code volume explicitly. It is however worth brief independent attention for at least two reasons — firstly because it is the easiest form of complexity to *measure*, and secondly because it interacts badly with the other causes of complexity and this is important to consider.

Brooks noted [Bro86]:

> "Many of the classic problems of developing software products derive from this essential complexity and its nonlinear increase with size"

We basically agree that *in most current systems* this is true (we disagree with the word "essential" as already noted) — i.e. in most systems complexity definitely *does* exhibit nonlinear increase with size (of the code). This nonlinearity in turn means that it's vital to reduce the amount of code to an absolute minimum.

We also want to draw attention to one of Dijkstra's [Dij72, EWD340] thoughts on this subject:

> "It has been suggested that there is some kind of law of nature telling us that the amount of intellectual effort needed grows with the square of program length. But, thank goodness, no one has been able to prove this law. And this is because it need not be true. ... I tend to the assumption — up till now not disproved by experience — that by suitable application of our powers of abstraction, the intellectual effort needed to conceive or to understand a program need not grow more than proportional to program length."

We agree with this — it is the reason for our "in most current systems" caveat above. We believe that — with the effective management of the two major complexity causes which we have discussed, state and control — it becomes *far* less clear that complexity increases with code volume in a non-linear way.

4.4   Other causes of complexity

Finally there are other causes, for example: duplicated code, code which is never actually used ("dead code"), **unnecessary abstraction**[3], missed abstraction, poor modularity, poor documentation. . .

All of these other causes come down to the following three (inter-related) principles:

Complexity breeds complexity There are a whole set of *secondary* causes of complexity. This covers all complexity introduced *as a result of* not being able to clearly understand a system. *Duplication* is a prime example of this — if (due to state, control or code volume) it is not clear that functionality already exists, or it is too *complex* to understand whether what exists does exactly what is required, there will be a strong tendency to duplicate. This is particularly true in the presence of time pressures.

Simplicity is *Hard* This was noted above — significant *effort* can be required to achieve simplicity. The first solution is often not the most simple, particularly if there is existing complexity, or time pressure. Simplicity can only be attained if it is recognised, sought and prized.

---

[3]Particularly unnecessary *data* abstraction. We examine an argument that this is actually *most* data abstraction in section 9.2.4.

Power corrupts What we mean by this is that, in the absence of language-enforced guarantees (i.e. restrictions on the *power* of the language) mistakes (and abuses) *will* happen. This is the reason that garbage collection is good — the *power* of manual memory management is removed. Exactly the same principle applies to *state* — another kind of *power*. In this case it means that we need to be very wary of any language that even *permits* state, regardless of how much it discourages its use (obvious examples are ML and Scheme). The bottom line is that the more *powerful* a language (i.e. the more that is *possible* within the language), the harder it is to *understand* systems constructed in it.

Some of these causes are of a human-nature, others due to environmental issues, but all can — we believe — be greatly alleviated by focusing on effective management of the complexity causes discussed in sections 4.1–4.3.

5   Classical approaches to managing complexity

The different classical approaches to managing complexity can perhaps best be understood by looking at how programming languages of each of the three major styles (imperative, functional, logic) approach the issue. (We take object-oriented languages as a commonly used example of the imperative style).

5.1   Object-Orientation

Object-orientation — whilst being a very broadly applied term (encompassing everything from Java-style class-based to Self-style prototype-based languages, from single-dispatch to CLOS-style multiple dispatch languages, and from traditional passive objects to the active / actor styles) — is essentially an imperative approach to programming. It has evolved as the dominant method of general software development for traditional (von-Neumann) computers, and many of its characteristics spring from a desire to facilitate von-Neumann style (i.e. state-based) computation.

5.1.1   State

In most forms of object-oriented programming (OOP) an object is seen as consisting of some state together with a set of procedures for accessing and manipulating that state.

This is essentially similar to the (earlier) idea of an *abstract data type (ADT)* and is one of the primary strengths of the OOP approach when compared with less structured imperative styles. In the OOP context this is referred to as the idea of *encapsulation*, and it allows the programmer to enforce integrity constraints over an object's state by regulating access to that state through the access procedures ("methods").

One problem with this is that, if several of the access procedures access or manipulate the same bit of state, then there may be several places where a given constraint must be enforced (these different access procedures may or may not be within the same file depending on the specific language and whether features, such as inheritance, are in use). Another major problem[4] is that encapsulation-based integrity constraint enforcement is strongly biased toward single-object constraints and it is awkward to enforce more complicated constraints involving multiple objects with this approach (for one thing it becomes unclear where such multiple-object constraints should reside).

Identity and State

There is one other intrinsic aspect of OOP which is intimately bound up with the issue of state, and that is the concept of *object identity*.

In OOP, each object is seen as being a uniquely identifiable entity regardless of its attributes. This is known as *intensional* identity (in contrast with *extensional* identity in which things are considered the same if their attributes are the same). As Baker observed [Bak93]:

> In a sense, object identity can be considered to be a rejection of the "relational algebra" view of the world in which two objects can only be distinguished through differing attributes.

Object identity *does* make sense when objects are used to provide a (mutable) stateful abstraction — because two distinct stateful objects can be mutated to contain different state even if their attributes (the contained state) happen initially to be the same.

However, in other situations where mutability is *not* required (such as — say — the need to represent a simple numeric value), the OOP approach is forced to adopt techniques such as the creation of "Value Objects", and an

---

[4]this particular problem doesn't really apply to object-oriented languages (such as CLOS) which are based upon generic functions — but they don't have the same concept of encapsulation.

attempt is made to de-emphasise the original intensional concept of *object identity* and re-introduce extensional identity. In these cases it is common to start using custom access procedures (methods) to determine whether two objects are equivalent in some other, domain-specific sense. (One risk — aside from the extra code volume required to support this — is that there can no longer be any guarantee that such domain-specific equivalence concepts conform to the standard idea of an equivalence relation — for example there is not necessarily any guarantee of transitivity).

The intrinsic concept of *object identity* stems directly from the use of state, and is (being part of the paradigm itself) unavoidable. This additional concept of identity adds complexity to the task of reasoning about systems developed in the OOP style (it is necessary to switch mentally between the two equivalence concepts — serious errors can result from confusion between the two).

State in OOP

The bottom line is that all forms of OOP rely on state (contained within objects) and in general all behaviour is affected by this state. As a result of this, OOP suffers directly from the problems associated with state described above, and as such we believe that it does not provide an adequate foundation for avoiding complexity.

5.1.2 Control

Most OOP languages offer standard sequential control flow, and many offer explicit classical "shared-state concurrency" mechanisms together with all the standard complexity problems that these can cause. One slight variation is that actor-style languages use the "message-passing" model of concurrency — they associate threads of control with individual objects and messages are passed between these. This can lead to easier informal reasoning in some cases, but the use of actor-style languages is not widespread.

5.1.3 Summary — OOP

Conventional imperative and object-oriented programs suffer greatly from both state-derived and control-derived complexity.

5.2   Functional Programming

Whilst OOP developed out of a desire to offer improved ways of managing and dealing with the classic stateful von-Neumann architecture, functional programming has its roots in the completely stateless lambda calculus of Church (we are ignoring the even simpler functional systems based on combinatory logic). The untyped lambda calculus is known to be equivalent in power to the standard stateful abstraction of computation — the Turing machine.

5.2.1   State

Modern functional programming languages are often classified as 'pure' — those such as Haskell[PJ$^+$03] which shun state and side-effects completely, and 'impure' — those such as ML which, whilst advocating the avoidance of state and side-effects in general, do permit their use. Where not explicitly mentioned we shall generally be considering functional programming in its pure form.

The primary strength of functional programming is that by avoiding state (and side-effects) the entire system gains the property of *referential transparency* — which implies that when supplied with a given set of arguments a function will *always* return exactly the same result (speaking loosely we could say that it will always behave in the same way). Everything which can possibly affect the result in any way is always immediately visible in the actual parameters.

It is this cast iron guarantee of *referential transparency* that obliterates one of the two crucial weaknesses of testing as discussed above. As a result, even though the other weakness of testing remains (testing for one set of inputs says *nothing at all* about behaviour with another set of inputs), testing does become far more effective if a system has been developed in a functional style.

By avoiding state functional programming also avoids all of the other state-related weaknesses discussed above, so — for example — informal reasoning also becomes much more effective.

5.2.2   Control

Most functional languages specify implicit (left-to-right) sequencing (of calculation of function arguments) and hence they face many of the same issues mentioned above. Functional languages do derive one slight benefit when

it comes to control because they encourage a more abstract use of control using functionals (such as `fold` / `map`) rather than explicit looping.

There are also concurrent versions of many functional languages, and the fact that state is generally avoided can give benefits in this area (for example in a pure functional language it will always be safe to evaluate all arguments to a function in parallel).

5.2.3 Kinds of State

In most of this paper when we refer to "state" what we really mean is *mutable state.*

In languages which do not support (or discourage) mutable state it is common to achieve somewhat similar effects by means of passing extra parameters to procedures (functions). Consider a procedure which performs some internal stateful computation and returns a result — perhaps the procedure implements a counter and returns an incremented value each time it is called:

```
procedure int getNextCounter()
  // 'counter' is declared and initialized elsewhere in the code
  counter := counter + 1
  return counter
```

The way that this is typically implemented in a basic functional programming language is to replace the stateful procedure which took no arguments and returned one result with a function which takes one argument and returns a pair of values as a result.

```
function (int,int) getNextCounter(int oldCounter)
  let int result = oldCounter + 1
  let int newCounter = oldCounter + 1
  return (newCounter, result)
```

There is then an obligation upon the caller of the function to make sure that the next time the `getNextCounter` function gets called it is supplied with the `newCounter` returned from the previous invocation.Effectively what is happening is that the *mutable* state that was hidden inside the `getNextCounter` procedure is replaced by an extra parameter on both the input and output of the `getNextCounter` function. This extra parameter is not *mutable* in any way (the entity which is referred to by `oldCounter` is a different *value* each time the function is called).

As we have discussed, the functional version of this program *is* referentially transparent, and the imperative version is not (hence the caller of the `getNextCounter` *procedure* has no idea what may influence the result he gets — it could in principle be dependent upon many, many different hidden mutable variables — but the caller of the `getNextCounter` *function* can instantly see exactly that the result can depend only on the single value supplied to the function).

Despite this, the fact is that we are using functional values to simulate state. There is *in principle* nothing to stop functional programs from passing a single extra parameter into and out of every single function in the entire system. If this extra parameter were a collection (compound value) of some kind then it could be used to *simulate* an arbitrarily large set of mutable variables. In effect this approach recreates a single pool of global variables — hence, even though referential transparency is maintained, ease of reasoning is lost (we still know that each function is dependent only upon its arguments, but one of them has become so large *and contains irrelevant values* that the benefit of this knowledge as an aid to understanding is almost nothing). This is however an extreme example and does not detract from the general power of the functional approach.

It is worth noting in passing that — even though it would be no substitute for a *guarantee* of referential transparency — there is no reason why the functional style of programming cannot be adopted in stateful languages (i.e. imperative as well as impure functional ones). More generally, we would argue that — whatever the language being used — there are large benefits to be had from avoiding hidden, implicit, mutable state.

5.2.4   State and Modularity

It is sometimes argued (e.g. [vRH04, p315]) that state is important because it permits a particular kind of modularity. This is certainly true. Working within a stateful framework it is possible to add state to any component without adjusting the components which invoke it. Working within a functional framework the same effect can only be achieved by adjusting every single component that invokes it to carry the additional information around (as with the `getNextCounter` function above).

There is a fundamental trade off between the two approaches. In the functional approach (when trying to achieve state-like results) you are forced to make changes to every part of the program that could be affected (adding the relevant extra parameter), in the stateful you are not.

But what this means is that in a functional program *you can always tell*

17

exactly what will control the outcome of a procedure (i.e. function) simply by looking at the arguments supplied where it is invoked. In a stateful program this property (again a consequence of *referential transparency*) is completely destroyed, you can never tell what will control the outcome, and *potentially* have to look at every single piece of code in the *entire system* to determine this information.

The trade-off is between *complexity* (with the ability to take a shortcut when making some specific types of change) and *simplicity* (with *huge* improvements in both testing and reasoning). As with the discipline of (static) typing, it is trading a one-off up-front cost for continuing future gains and safety ("one-off" because each piece of code is written once but is read, reasoned about and tested on a continuing basis).

A further problem with the modularity argument is that some examples — such as the use of procedure (function) invocation counts for debugging / performance-tuning purposes — seem to be better addressed within the supporting infrastructure / language, rather than within the system itself (we prefer to advocate a clear separation between such administrative/diagnostic information and the core logic of the system).

Still, the fact remains that such arguments have been insufficient to result in widespread adoption of functional programming. We must therefore conclude that the main weakness of functional programming is the flip side of its main strength — namely that problems arise when (as is often the case) the system to be built must maintain state of some kind.

The question inevitably arises of whether we can find some way to "have our cake and eat it". One potential approach is the elegant system of monads used by Haskell [Wad95]. This does basically allow you to avoid the problem described above, but it can very easily be abused to create a stateful, side-effecting sub-language (and hence re-introduce all the problems we are seeking to avoid) inside Haskell — albeit one that is marked by its type. Again, despite their huge strengths, monads have as yet been insufficient to give rise to widespread adoption of functional techniques.

5.2.5   Summary — Functional Programming

Functional programming goes a long way towards avoiding the problems of state-derived complexity. This has very significant benefits for testing (avoiding what is normally one of testing's biggest weaknesses) as well as for reasoning.

5.3 Logic Programming

Together with functional programming, logic programming is considered to be a *declarative* style of programming because the emphasis is on specifying *what* needs to be done rather than exactly *how* to do it. Also as with functional programming — and in contrast with OOP — its principles and the way of thinking encouraged do *not* derive from the stateful von-Neumann architecture.

Pure logic programming is the approach of doing nothing more than making statements *about* the problem (and desired solutions). This is done by stating a set of *axioms* which *describe* the problem and the attributes required of something for it to be considered a solution. The ideal of logic programming is that there should be an infrastructure which can take the raw axioms and use them to find or check solutions. All solutions are formal logical consequences of the axioms supplied, and "running" the system is equivalent to the construction of a formal *proof* of each solution.

The seminal "logic programming" language was Prolog. Prolog is best seen as a *pure logical core* (pure Prolog) with various *extra-logical*[5] extensions. Pure Prolog is close to the ideals of logic programming, but there are important differences. Every pure Prolog program can be "read" in two ways — either as a *pure set of logical axioms* (i.e. assertions about the problem domain — this is the pure logic programming reading), or *operationally* — as a sequence of commands which are applied (in a particular order) to determine whether a goal can be deduced (from the axioms). This second reading corresponds to the actual way that pure Prolog will make use of the axioms when it tries to prove goals. It is worth noting that a single Prolog program can be both correct when read in the first way, and incorrect (for example due to non-termination) when read in the second.

It is for this reason that Prolog falls short of the ideals of logic programming. Specifically it is necessary to be concerned with the operational interpretation of the program whilst writing the axioms.

5.3.1 State

Pure logic programming makes no use of mutable state, and for this reason profits from the same advantages in understandability that accrue to pure functional programming. Many languages based on the paradigm do however provide some stateful mechanisms. In the extra-logical part of Prolog

---

[5]We are using the term here to cover *everything* apart from the pure core of Prolog — for example we include what are sometimes referred to as the *meta-logical* features

for example there are facilities for adjusting the program itself by adding new axioms for example. Other languages such as Oz (which has its roots in logic programming but has been extended to become "multi-paradigm") provide mutable state in a traditional way — similar to the way it is provided by *impure* functional languages.

All of these approaches to state sacrifice referential transparency and hence potentially suffer from the same drawbacks as imperative languages in this regard. The one advantage that all these impure non-von-Neumann derived languages can claim is that — whilst state is permitted its use is generally discouraged (which is in stark contrast to the stateful von-Neumann world). Still, without purity there are no guarantees and all the same state-related problems can sometimes occur.

5.3.2 Control

In the case of pure Prolog the language specifies both an *implicit* ordering for the processing of sub-goals (left to right), and also an *implicit* ordering of clause application (top down) — these basically correspond to an operational commitment to *process* the program in the same order as it is read textually (in a depth first manner). This means that some particular ways of writing down the program can lead to non-termination, and — when combined with the fact that some extra-logical features of the language permit side-effects — leads inevitably to the standard difficulty for informal reasoning caused by control flow. (Note that these reasoning difficulties do *not* arise in ideal world of logic programming where there simply is no specified control — as distinct from in pure Prolog programming where there is).

As for Prolog's other extra-logical features, some of them further widen the gap between the language and logic programming in its ideal form. One example of this is the provision of "cuts" which offer *explicit* restriction of control flow. These explicit restrictions are intertwined with the pure logic component of the system and inevitably have an adverse affect on attempts to reason about the program (misunderstandings of the effects of cuts are recognised to be a major source of bugs in Prolog programs [SS94, p190]).

It is worth noting that some more modern languages of the logic programming family offer more flexibility over control than the implicit depth-first search used by Prolog. One example would be Oz which offers the ability to program specific control strategies which can then be applied to different problems as desired. This is a very useful feature because it allows significant *explicit* control flexibility to be specified *separately* from the main program (i.e. without contaminating it through the addition of control complexity).

5.3.3 Summary — Logic Programming

One of the most interesting things about logic programming is that (despite the limitations of some actual logic-based languages) it offers the tantalising promise of the ability to escape from the complexity problems caused by control.

6   Accidents and Essence

Brooks defined difficulties of "essence" as those inherent in the nature of software and classified the rest as "accidents".

We shall basically use the terms in the same sense — but prefer to start by considering the complexity of the problem itself before software has even entered the picture. Hence we define the following two types of complexity:

Essential Complexity is inherent in, and the essence of, the *problem* (as seen by the *users*).

Accidental Complexity is all the rest — complexity with which the development team would not have to deal in the ideal world (e.g. complexity arising from performance issues and from suboptimal language and infrastructure).

Note that the definition of *essential* is deliberately more strict than common usage. Specifically when we use the term *essential* we will mean strictly essential *to the users' problem* (as opposed to — perhaps — essential *to some specific, implemented, system*, or even — essential *to software in general*). For example — according to the terminology we shall use in this paper — bits, bytes, transistors, electricity and computers themselves are *not* in any way essential (because they have nothing to do with the users' problem).

Also, the term "accident" is more commonly used with the connotation of "mishap". Here (as with Brooks) we use it in the more general sense of "something non-essential which is present".

In order to justify these two definitions we start by considering the role of a software development team — namely to produce (using some given language and infrastructure) and maintain a software system which serves the purposes of its users. The complexity in which we are interested is the complexity involved in this task, and it is this which we seek to classify as accidental or essential. We hence see essential complexity as "the complexity with which the team will *have* to be concerned, even in the ideal world".

Note that the "have to" part of this observation is critical — if there is any *possible* way that the team could produce a system that the users will consider correct *without* having to be concerned with a given type of complexity then that complexity is not essential.

Given that in the real world not all *possible* ways are practical, the implication is that any real development *will* need to contend with *some* accidental complexity. The definition does not seek to deny this — merely to identify its secondary nature.

Ultimately (as we shall see below in section 7) our definition is equivalent to saying that what is essential to the team is what the *users* have to be concerned with. This is because in the ideal world we would be using language and infrastructure which would let us express the users' problem directly without having to express anything else — and this is how we arrive at the definitions given above.

The argument might be presented that in the *ideal* world we could just find infrastructure which already solves the users' problem completely. Whilst it is possible to imagine that someone has done the work already, it is not particularly enlightening — it may be best to consider an implicit restriction that the hypothetical language and infrastructure be general purpose and domain-neutral.

One implication of this definition is that if the user doesn't even *know what something is* (e.g. a thread pool or a loop counter — to pick two arbitrary examples) then it cannot possibly be essential by our definition (we are assuming of course — alas possibly with some optimism — that the users do in fact know and understand the problem that they want solved).

Brooks asserts [Bro86] (and others such as Booch agree [Boo91]) that "The complexity of software is an essential property, not an accidental one". This would suggest that the majority (at least) of the complexity that we find in contemporary large systems is of the essential type.

We disagree. Complexity itself is not an inherent (or essential) property of software (it is perfectly possible to write software which is simple and yet is still software), and further, much complexity that we do see in existing software is not essential (to the problem). When it comes to accidental and essential complexity we firmly believe that the former exists and that the goal of software engineering must be both to eliminate as much of it as possible, and to assist with the latter.

Because of this it is vital that we carefully scrutinize *accidental complexity*. We now attempt to classify occurrences of complexity as either accidental or essential.

7 Recommended General Approach

Given that our main recommendations revolve around trying to *avoid* as much accidental complexity as possible, we now need to look at *which* bits of the complexity must be considered accidental and which essential.

We shall answer this by considering exactly what complexity could not *possibly* be avoided even in the ideal world (this is basically how we *define* essential). We then follow this up with a look at just how realistic this ideal world really is before finally giving some recommendations.

7.1 Ideal World

In the ideal world we are not concerned with performance, and our language and infrastructure provide all the general support we desire. It is against this background that we are going to examine *state* and *control*. Specifically, we are going to identify state as *accidental state* if we can omit it in this ideal world, and the same applies to control.

Even in the ideal world we need to start somewhere, and it seems reasonable to assume that we need to start with a set of *informal requirements* from the prospective users.

Our next observation is that because we ultimately need something to *happen* — i.e. we are going to need to have our system processed mechanically (on a computer) — we are going to need *formality*. We are going to need to derive formal requirements from the informal ones.

So, taken together, this means that even in the ideal world we have:

$$\text{Informal requirements} \rightarrow \text{Formal requirements}$$

Note that given that we're aiming for simplicity, it is crucial that the formalisation be done without adding any *accidental* aspects at all. Specifically this means that in the ideal world, formalisation must be done with *no view to execution whatsoever*. The *sole* concern when producing the formal requirements must be to ensure that there is no *relevant*[6] ambiguity in the informal requirements (i.e. that it has no omissions).

So, having produced the formalised requirements, what should the next step be? Given that we are considering the ideal world, it is not unreasonable

---

[6]We include the word "relevant" here because in many cases there may be many possible acceptable solutions — and in such cases the requirements can be ambiguous in that regard, however that is not considered to be a "relevant" ambiguity, i.e. it does not correspond to an erroneous *omission* from the requirements.

to assume that the next step is simply to *execute* these formal requirements directly on our underlying general purpose infrastructure.[7]

This state of affairs is *absolute* simplicity — it does not seem conceivable that we can do any better than this even in an ideal world.

It is interesting to note that effectively what we have just described is in fact the very *essence* of *declarative programming* — i.e. that you need only specify *what* you require, not *how* it must be achieved.

We now consider the implications of this "ideal" approach for the causes of complexity discussed above.

7.1.1   State in the ideal world

Our main aim for state in the ideal world is to get rid of it — i.e. we are hoping that most state will turn out to be *accidental state*.

We start from the perspective of the users' informal requirements. These will mention data of various kinds — some of which can give rise to *state* — and it is these kinds which we now classify.

All data will either be provided directly to the system (*input*) or *derived*. Additionally, derived data is either *immutable* (if the data is intended only for display) or *mutable* (if explicit reference is made within the requirements to the ability of users to update that data).

All data mentioned in the users' informal requirements is of concern to the users, and is as such *essential*. The fact that all such data is *essential* does *not* however mean that it will all unavoidably correspond to *essential state*. It may well be possible to avoid storing some such data, instead dealing with it in some other essential aspect of the system (such as the *logic*) — this is the case with derived data, as we shall see. In cases where this is possible the data corresponds to *accidental state*.

Input Data

Data which is provided directly (input) will have to have been included in the informal requirements and as such is deemed *essential*. There are basically two cases:

- There is (according to the requirements) a possibility that the system may be required to refer to the data in the future.

- There is no such possibility.

---

[7]In the presence of *irrelevant* ambiguities this will mean that the infrastructure must *choose* one of the possibilities, or perhaps even provide all possible solutions

In the first case, even in the ideal world, the system must retain the data and as such it corresponds to *essential state*.

In the second case (which will most often happen when the input is designed simply to cause some side-effect) the data need not be maintained at all.

Essential Derived Data — Immutable

Data of this kind can always be re-derived (from the input data — i.e. from the *essential state*) whenever required. As a result we do *not* need to store it in the ideal world (we just re-derive it when it is required) and it is clearly *accidental state*.

Essential Derived Data — Mutable

As with immutable essential derived data, this can be excluded (and the data re-derived on demand) and hence corresponds to *accidental state*.

Mutability of derived data makes sense only where the function (logic) used to derive the data has an inverse (otherwise — given its mutability — the data cannot be considered *derived* on an ongoing basis, and it is effectively *input*). An inverse often exists where the derived data represents simple restructurings of the input data. In this situation modifications to the data can simply be treated identically to the corresponding modifications to the existing *essential state*.

Accidental Derived Data

State which is *derived* but *not* in the users' requirements is also *accidental state*. Consider the following imperative pseudo-code:

```
procedure int doCalculation(int y)
  // 'subsidaryCalcCache' is declared and initialized
  // elsewhere in the code
  if (subsidaryCalcCache.contains(y) == false) {
    subsidaryCalcCache.y := slowSubsidaryCalculation(y)
  }
  return 3 * (4 + subsidaryCalcCache.y)
```

The above use of state in the `doCalculation` procedure seems to be unnecessary (in the ideal world), and hence of the *accidental* variety. We

25

Data Essentiality	Data Type	Data Mutability	Classification
Essential	Input	-	Essential State
Essential	Derived	Immutable	Accidental State
Essential	Derived	Mutable	Accidental State
Accidental	Derived	-	Accidental State

Table 1: Data and State

cannot actually be sure without knowing whether and how the `subsidary-CalcCache` is used elsewhere in the program, but for this example we shall assume that there are no other uses aside from initialization. The above procedure is thus equivalent to:

```
procedure int doCalculation(int y)
  return 3 * (4 + slowSubsidaryCalculation(y))
```

It is almost certain that this use of state would *not* have been part of the users' informal requirements. It is also *derived*. Hence, it is quite clear that we can eliminate it completely from our ideal world, and that hence it *is* accidental.

Summary — State in the ideal world

For our ideal approach to state, we largely follow the example of functional programming which shows how mutable state can be avoided. We need to remember though that:

1. even in the ideal world we *are* going to have *some* essential state — as we have just established

2. pure functional programs can effectively simulate accidental state in the same way that they can simulate essential state (using techniques such as the one discussed above in section 5.2.3) — we obviously want to avoid this in the ideal world.

The data type classifications are summarized in Table 1. Wherever the table shows data as corresponding to *accidental state* it means that it can be excluded from the ideal world (by re-deriving the data as required).

The obvious implication of the above is that there are large amounts of *accidental state* in typical systems. In fact, it is our belief that the vast majority of state (as encountered in typical contemporary systems) simply

isn't needed (in this ideal world). Because of this, and the huge complexity which state can cause, the ideal world removes *all* non-essential state. There is no other state at all. No caches, no stores of derived calculations of any kind. One effect of this is that *all* the state in the system is *visible* to the user of (or person testing) the system (because inputs can reasonably be expected to be visible in ways which internal cached state normally is not).

7.1.2 Control in the ideal world

Whereas we have seen that some state *is* essential, control generally can be completely omitted from the ideal world and as such is considered entirely *accidental*. It typically won't be mentioned in the informal requirements and hence should not appear in the formal requirements (because these are derived with *no view to execution*).

What do we mean by this? Clearly if the program is ever to run, *some* control will be needed somewhere because things will have to happen in some order — but this should no more be our concern than the fact that the chances are some electricity will be needed somewhere. The important thing is that we (as developers of the system) should not have to worry about the control flow in the system. Specifically the *results* of the system should be independent of the actual control mechanism which is finally used.

These are precisely the lessons which logic programming teaches us, and because of this we would like to take the lead for our ideal approach to control from logic programming which shows that control can be separated completely.

It is worth noting that because typically the informal requirements will not mention concurrency, that too is normally of an accidental nature. In an ideal world we can assume that finite (stateless) computations take zero time[8] and as such it is immaterial to a user whether they happen in sequence or in parallel.

7.1.3 Summary

In the ideal world we have been able to avoid large amounts of complexity — both state and control. As a result, it is clear that a lot of complexity is *accidental*. This gives us hope that it may be possible to significantly reduce the complexity of *real* large systems. The question is — how close is it possible to get to the ideal world in the real one?

---

[8]this assumption is generally known as the "synchrony hypothesis"

7.2 Theoretical and Practical Limitations

The real world is not of course ideal. In this section we examine a few of the assumptions made in the section 7.1 and see where they break down.

As already noted, our vision of an ideal world is similar in many ways to the vision of *declarative programming* that lies behind functional and logic programming.

Unfortunately we have seen that functional and logic programming ultimately had to confront both state and control. We should note that the reasons for having to confront each are slightly different. State is required simply because most systems do have some state as part of their true essence. Control generally *is* accidental (the users normally are not concerned about it at all) but the ability to restrict and influence it is often required from a practical point of view. Additionally practical (e.g. efficiency) concerns will often dictate the use of some accidental state.

These observations give some indication of where we can expect to encounter difficulties.

7.2.1 Formal Specification Languages

First of all, we want to consider two problems (one of a theoretical kind, the other practical) that arise in connection with the ideal-world *formal requirements*.

In that section we discussed the need for *formal requirements* derived directly from the *informal requirements*. We observed that in the ideal world we would like to be able to execute the formal requirements without first having to translate them into some other language.

The phrase "formal requirements" is basically synonymous with "formal specification", so what effectively we're saying would be ideal are *executable specifications*. Indeed both the declarative programming paradigms discussed above (functional programming and logic programming) have been proposed as approaches for executable specifications.

Before we consider the problems with executing them, we want to comment that *the way in which* the ideal world formal specifications were derived — *directly* from the users' informal requirements — was critical. Formal specifications can be derived in various other ways (some of which risk the introduction of accidental complexity), and can be of various different kinds.

Traditionally formal specification has been categorized into two main camps:

Property-based approaches focus (in a declarative manner) on *what* is

required rather than *how* the requirements should be achieved. These approaches include the *algebraic* (*equational axiomatic semantics*) approaches such as Larch and OBJ.

Model-based (or State-based) approaches construct a potential model for the system (often a stateful model) and specify how that model must behave. These approaches (which include Z and VDM) can hence be used to specify how a stateful, imperative language solution must behave to satisfy the requirements. (We discussed the weaknesses of stateful imperative languages in section 5).

The first problem that we want to discuss in this section is the more theoretical one. Arguments (which focus more on the model-based approaches) have been put forward *against* the concept of executable specifications [HJ89]. The main objection is that requiring a specification language to be executable can directly restrict its expressiveness (for example when specifying requirements for a variable x it may be desirable to assert something like $\neg\exists y | f(y, x)$ which clearly has no direct operational interpretation).

In response to this objection, we would say firstly that in our experience a requirement for this kind of expressivity does not seem to be common in many problem domains. Secondly it would seem sensible that where such specifications *do* occur they should be maintained in their natural form but supplemented with a *separate* operational component. Indeed in this situation it would not seem too unreasonable to consider the required operational component to be accidental in nature (of course the reality is that in cases like this the boundary between what is accidental and essential, what is reasonable to hope for in an "ideal" world, becomes less clear). Some specification languages address this issue by having an executable subset.

Finally, it is the *property-based* approaches that seem to have the greatest similarity to what we have in mind when we talk about *executable specifications* in the ideal world. It certainly *is* possible to execute algebraic specifications — deriving an operational semantics by choosing a direction for each of the equational axioms.[9]

In summary, the first problem is that consideration of specification languages highlights the (theoretically) fuzzy boundary between what is essential and what is accidental — specifically it challenges the validity of our definition of *essential* (which we identified closely with requirements from the *users*) by observing that it is possible to specify things which are *not*

---

[9]Care must be taken that the resulting reduction rules are *confluent* and *terminating*.

directly executable. For the reasons given above (and in section 6) we think that — from the practical point of view — our definition is still viable, import and justified.

The second problem is of a more practical nature — namely that even when specifications *are* directly executable, this can be impractical for efficiency reasons. Our response to this is that whilst it is undoubtedly true, we believe that it is very important (for understanding and hence for avoiding complexity) not to lose the distinction we have defined between what is *accidental* and *essential*. As a result, this means that we will *require* some accidental components as we shall see in section 7.2.3.

7.2.2   Ease of Expression

There is one final practical problem that we want to consider — even though we believe it is fairly rare in most application domains. In section 7.1.1 we argued that immutable, derived data would correspond to *accidental state* and could be omitted (because the *logic* of the system could always be used to derive the data on-demand).

Whilst this is true, there are occasionally situations where the ideal world approach (of having no accidental state, and using on-demand derivation) does not give rise to the most natural modelling of the problem.

One possible situation of this kind is for derived data which is dependent upon *both* a whole series of user inputs over time, *and* its own previous values. In such cases it can be advantageous[10] to *maintain* the *accidental state* even in the ideal world.

An example of this would be the derived data representing the position state of a computer-controlled opponent in an interactive game — it is at all times *derivable* by a function of both all prior user movements and the initial starting positions,[11] but this is not the way it is most naturally expressed.

7.2.3   Required Accidental Complexity

We have seen two possible reasons why in practice — even with optimal language and infrastructure — we may *require* complexity which strictly is *accidental*. These reasons are:

Performance making use of accidental state and control can be required for efficiency — as we saw in the second problem of section 7.2.1.

---

[10]because it can make the *logic* easier to express — as we shall see in section 7.3.2

[11]We are implicitly considering *time* as an additional input.

Ease of Expression making use of accidental state can be the most natural way to express logic in some cases — as we saw in section 7.2.2.

Of the two, we believe that *performance* will be the most common.

It is of course vital to be aware that as soon as we re-introduce this accidental complexity, we are again becoming exposed to the dangers discussed in sections 4.1 and 4.2. Specifically we can see that if we add in *accidental state* which has to be managed explicitly by the logic of the system, then we become at risk of the possibility of the system entering an *inconsistent state* (or "bad state") due to errors in that explicit logic. This is a very serious concern, and is one that we address in our recommendations below.

7.3   Recommendations

We believe that — despite the existence of required accidental complexity — it *is* possible to retain most of the simplicity of the ideal world (section 7.1) in the real one. We now look at how this might be achievable.

Our recommendations for dealing with complexity (as exemplified by both state and control) can be summed up as:

- Avoid

- Separate

Specifically the overriding aim must be to *avoid* state and control where they are not absolutely and truly essential.

The recommendation of avoidance is however tempered by the acknowledgement that there will sometimes be complexity that either is truly essential (section 7.1.1) or, whilst not *truly* essential, is useful from a practical point of view (section 7.2.3). Such complexity must be separated out from the rest of the system — and this gives us our second recommendation.

There is nothing particularly profound in these recommendations, but they are worth stating because they are emphatically *not* the way most software is developed today. It is the fact that current established practice does *not* use these as central overriding principles for software development that leads directly to the complexity that we see everywhere, and as already argued, it is that complexity which leads to the software crisis[12].

In addition to not being profound, the principles behind these recommendations are not really new. In fact, in a classic 1979 paper Kowalski

---

[12]There *is* some limited similarity between our goal of "Separate" and the goal of *separation of concerns* as promoted by proponents of Aspect Oriented Programming — but as we shall see in section 7.3.2, exactly what is meant by separation is critical.

(co-inventor of Prolog) argued in exactly this direction [Kow79]. The title of his paper was the equation:

$$\text{``}Algorithm = Logic + Control\text{''}$$

. . . and this separation that he advocated is close to the heart of what we're recommending.

7.3.1 Required Accidental Complexity

In section 7.2.3 we noted two possible reasons for requiring accidental complexity (even in the presence of optimal language and infrastructure). We now consider the most appropriate way of handling each.

Performance

We have seen that there are many serious risks which arise from accidental complexity — particularly when introduced in an undisciplined manner. To mitigate these risks we take two defensive measures.

The first is with regard to the risks of explicit management of accidental state (which we have argued is actually the *majority* of state). The recommendation here is that we completely *avoid* explicit management of the accidental state — instead we should restrict ourselves to simply *declaring* what accidental state should be used, and leave it to a completely separate infrastructure (on which our system will eventually run) to maintain. This is reasonable because the infrastructure can make use of the (separate) system logic which specifies how accidental data must be derived.

By doing this we eliminate any risk of state inconsistency (bugs in the infrastructure aside of course). Indeed, as we shall see (in section 7.3.2), from the point of view of the *logic* of the system, we can effectively forget that the *accidental state* even exists. More specific examples of this approach are given in the second half of this paper.

The other defensive action we take is "Separate". We examine separation after first looking at the other possible reason for requiring accidental complexity.

Ease of Expression

This problem (see section 7.2.2) fundamentally arises when derived (i.e. *accidental*) state offers the most natural way to express parts of the logic of the system.

Complexity	Type	Recommendation
Essential Logic		Separate
Essential Complexity	State	Separate
Accidental Useful Complexity	State / Control	Separate
Accidental Useless Complexity	State / Control	Avoid

Table 2: Types of complexity within a system

The difficulty then arises that this requirement (to *use* the *accidental state* in a fairly direct manner inside the system logic) clashes with the goal of *separation* that we have just discussed. This very *separation* is *critical* when it comes to avoiding complexity, so we do not want to sacrifice it for this (probably fairly rare) situation.

Instead what we recommend is that, in cases where it really is the only natural thing to do, we should *pretend* that the *accidental state* is really *essential state* for the purposes of the *separation* discussed below. One straightforward way to do this is to make use of an external component which observes the derived data in question and creates the illusion of the *user* typing that same (derived, *accidental*) data back in as input data (we touch on this issue again in section 9.1.4).

7.3.2   Separation and the relationship between the components

In the above we deliberately glossed over *exactly* what we meant by our second recommendation: "Separate". This is because it actually encompasses two things.

The first thing that we're doing is to advocate separating out *all* complexity of any kind from the pure logic of the system (which — having nothing to do with either state or control — we're not really considering part of the complexity). This could be referred to as the *logic / state* split (although of course state is just one aspect of complexity — albeit the main one).

The second is that we're further dividing the complexity which we do retain into accidental and essential. This could be referred to as the *accidental / essential* split. These two splits can more clearly be seen by considering the Table 2. (N.B. We do not consider there to be any essential control).

The essential bits correspond to the requirements in the ideal world of section 7.1 — i.e. we are recommending that the formal requirements adopt the *logic / state* split.

The top three rows of the table correspond to components which we

expect to exist in most practical systems (some systems may not actually require any essential state, but we include it here for generality). i.e. These are the three things which will need to be specified (in terms of a given underlying language and infrastructure) by the development team.

"Separate" is basically advocating clean distinction between all three of these components. It is additionally advocating a split between the state and control components of the "Useful" Accidental Complexity — but this distinction is less important than the others.

One implication of this overall structure is that the system (essential + accidental but useful) should still function *completely correctly* if the "accidental but useful" bits are removed (leaving only the two *essential* components) — albeit possibly unacceptably slowly. As Kowalski (who — writing in a Prolog-context — was not really considering any essential state) says:

> "The logic component determines the meaning ...whereas the control component only affects its efficiency".

A consequence of *separation* is that the separately specified components will each be of a *very* different nature, and as a result it may be ideal to use *different languages* for each. These languages would each be oriented (i.e. *restricted*) to their specific goal — there is no sense in having control specification primitives in a language for specifying state. This notion of *restricting the power* of the individual languages is an important one — the weaker the language, the more simple it is to reason about. This has something in common with the ideas behind "Domain Specific Languages" — one exception being that the domains in question are of a fairly abstract nature and combine to form a general-purpose platform.

The vital importance of separation comes simply from the fact that it is separation that allows us to "*restrict the power*" of each of the components independently. The restricted power of the respective languages with which each component is expressed facilitates reasoning about them individually. The very fact that the three are separated from each other facilitates reasoning about them as a whole (e.g. you do not have to think about accidental state at all when you are working on the essential logic of your system[13]).

Figure 1 shows the same three expected components of a system in a different way (compare with Table 2). Each box in the diagram corresponds to some aspect of the system which will *need to be specified* by the development team. Specifically, it will be necessary to specify what the essential

---

[13]indeed it should be perfectly possible for different users of the same essential system to employ different accidental components — each designed for their particular needs
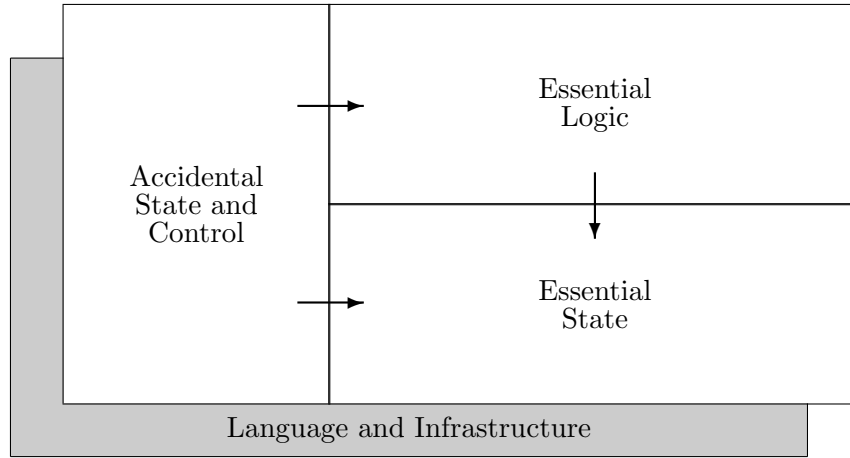
Figure 1: Recommended Architecture (arrows show static references)

state can be, what must always be logically true, and finally what accidental use can be made of state and control (typically for performance reasons).

The differing nature of what is specified by each of the components leads naturally to certain relationships between them, to *restrictions* on the ways in which they can or cannot refer to each other. These restrictions are absolute, and because of this provide a huge aid to understanding the different components of the system independently.

Essential State This can be seen as the foundation of the system. The specification of the required state is completely self-contained — it can *make no reference* to *either* of the other parts which must be specified. One implication of this is that changes to the essential state specification itself may require changes in both the other specifications, but changes in either of the other specifications may never require changes to the specification of essential state.

Essential Logic This is in some ways the "heart" of the system — it expresses what is sometimes termed the "business" logic. This logic expresses — in terms of the state — what must be true. It does *not* say anything about how, when, or why the state might change dynamically — indeed it wouldn't make *sense* for the logic to be able to change the state in any way.

Changes to the essential state specification may require changes to the logic specification, and changes to the logic specification may require changes to the specification for accidental state and control. The logic specification will *make no reference* to *any* part of the accidental

35

specification. Changes in the accidental specification can hence never require any change to the essential logic.

Accidental State and Control This (by virtue of its accidental nature) is conceptually the least important part of the system. Changes to it can *never* affect the other specifications (because neither of them make any reference to any part of it), but changes to *either* of the others may require changes here.

Together the goals of *avoid* and *separate* give us reason to hope that we may well be able to retain much of the simplicity of the ideal world in the real one.

7.4   Summary

This first part of the paper has done two main things. It has given arguments for the overriding danger of complexity, and it has given some hope that much of the complexity may be avoided or controlled.

The key difference between what we are advocating and existing approaches (as embodied by the various styles of programming language) is a high level *separation* into three components — each specified in a different language[14]. It is this *separation* which allows us to *restrict* the power of each individual component, and it is this use of *restricted* languages which is vital in making the overall system easier to comprehend (as we argued in section 4.4 — *power corrupts*).

Doing this separation when building a system may not be easy, but we believe that for any large system it will be *significantly* less difficult than dealing with the complexity that arises otherwise.

It is hard to overstate the dangers of complexity. If it is not controlled it spreads. The *only* way to escape this risk is to place the goals of *avoid* and *separate* at the top of the design objectives for a system. It is not sufficient simply to pay heed to these two objectives — it is crucial that they be the *overriding* consideration. This is because complexity breeds complexity and one or two early "compromises" can spell complexity disaster in the long run.

It is worth noting in particular the risks of "designing for performance". The dangers of "premature optimisation" are as real as ever — there can be no comparison between the difficulty of improving the performance of a

---

[14]or different subsets of the same language, *provided* it is possible to *forcibly restrict* each component to the relevant subset.

slow system designed for simplicity and that of removing complexity from a complex system which was designed to be fast (and quite possibly isn't even that because of myriad inefficiencies hiding within its complexity).

In the second half of this paper we shall consider a possible approach based on these recommendations.

8 The Relational Model

The relational model [Cod70] has — despite its origins — nothing *intrinsically* to do with databases. Rather it is an elegant approach to *structuring* data, a means for *manipulating* such data, and a mechanism for maintaining *integrity* and consistency of state. These features are applicable to state and data in any context.

In addition to these three broad areas [Cod79, section 2.1], [Dat04, p109], a fourth strength of the relational model is its insistence on a clear separation between the logical and physical layers of the system. This means that the concerns of designing a logical model (minimizing the complexity) are addressed separately from the concerns of designing an efficient physical storage model and mapping between that and the logical model[15]. This principle is called *data independence* and is a crucial part of the relational model [Cod70, section 1.1].

We see the relational model as having the following four aspects:

Structure the use of *relations* as the means for representing all data

Manipulation a means to specify derived data

Integrity a means to specify certain inviolable restrictions on the data

Data Independence a clear separation is enforced between the logical data and its physical representation

We will look briefly at each of these aspects. [Dat04] provides a more thorough overview of the relational model.

As a final comment, it is widely recognised that SQL (of any version) — despite its widespread use — is *not* an accurate reflection of the relational model [Cod90, p371, Serious flaws in SQL], [Dat04, p xxiv] so the reader is warned against equating the two.

---

[15]Unfortunately most contemporary DBMSs are somewhat limited in the degree of flexibility permitted by the physical/logical mapping. This has the unhappy result that physical performance concerns can invade the logical design even though *avoiding* exactly this was one of Codd's most important original goals.

8.1 Structure

8.1.1 Relations

As mentioned above, relations provide the sole means for structuring data in the relational model. A relation is best seen as a homogeneous *set of records*, each record itself consisting of a heterogeneous set of uniquely named *attributes* (this is slightly different from the general mathematical definition of a relation as a set of tuples whose components are identified by position rather than name).

Implications of this definition include the fact that — by virtue of being a set — a relation can contain no duplicates, and it has no ordering. Both of these restrictions are in contrast with the common usage of the word *table* which can obviously contain duplicate rows (and column names), and — by virtue of being a visual entity on a page — inevitably has both an ordering of its rows and of its columns.

Relations can be of the following kinds:

Base Relations are those which are stored directly

Derived Relations (also known as *Views*) are those which are defined in terms of other relations (base or derived) — see section 8.2

Following Date [Dat04] it is useful to think of a relation as being a single (albeit compound) *value*, and to consider any mutable state not as a "mutable relation" but rather as a *variable* which at any time can contain a particular relation *value*. Date calls these variables *relation variables* or *relvars*, leading to the terms *base relvar* and *derived relvar*, and we shall use this terminology later. (Note however that our definition of relation is slightly different from his in that — following standard static typing practice — we do not consider the type to be part of the value).

8.1.2 Structuring benefits of Relations — Access path independence

The idea of structuring data using relations is appealing because no *subjective,* up-front decisions need to be made about the *access paths* that will later be used to query and process the data.

To understand what is meant by *access path*, let us consider a simple example. Suppose we are trying to represent information about employees and the departments in which they work. A system in which choosing the structure for the data involves setting up "routes" between data instances

(such as *from* a particular employee *to* a particular department) is *access path dependent.*

The two main data structuring approaches which preceded the relational model (the network and hierarchical models) were both access path dependent in this way. For example, in the hierarchical model a subjective choice would be forced early on as to whether departments would form the top level (with each department "containing" its employees) or the other way round (with employees "containing" their departments). The choice made would impact all future use of the data. If the first alternative was selected, then users of the data would find it easy to retrieve all employees within a given department (following the access path), but they would find it harder to retrieve the department of a given employee (and would have to use some other technique corresponding to a search of all departments). If the second alternative was selected then the problem was simply reversed.

The network model alleviated the problem to some degree by allowing multiple access paths between data instances (so the choice could be made to provide both an access path from department to employee and an access path from employee to department). The problem of course is that it is impossible to predict in advance what all the future required access paths will be, and because of this there will always be a disparity between:

Primary retrieval requirements which were foreseen, and can be satisfied simply by following the provided access paths

Secondary retrieval requirements which were either unforeseen, or at least not specially supported, and hence can only be satisfied by some alternative mechanism such as search

The ability of the relational model to *avoid* access paths completely was one of the primary reasons for its success over the network and hierarchical models.

It is also interesting to consider briefly what is involved when taking an object-oriented (OOP) approach to our example. We can choose between the following options:

- Give Employee objects a reference to their Department

- Give Department objects a set (or array) of references to their Employees

- Both of the above

If we choose the third option, then we at best expose ourselves to extra work in maintaining the redundant references, and at worst expose ourselves to bugs.

There are disturbing similarities between the data structuring approaches of OOP and XML on the one hand and the network and hierarchical models on the other.

A final advantage of using relations for the structure — in contrast with approaches such as Chen's ER-modelling [Che76] — is that no distinction is made between *entities* and *relationships*. (Using such a distinction can be problematic because whether something is an *entity* or a *relationship* can be a very subjective question).

8.2   Manipulation

Codd introduced two different mechanisms for expressing the manipulation aspects of the relational model — the relational calculus and the relational algebra. They are formally equivalent (in that expressions in each can be converted into equivalent expressions in the other), and we shall only consider the algebra.

The relational algebra (which is now normally considered in a slightly different form from the one used originally by Codd) consists of the following eight operations:

Restrict is a unary operation which allows the selection of a subset of the records in a relation according to some desired criteria

Project is a unary operation which creates a new relation corresponding to the old relation with various attributes removed from the records

Product is a binary operation corresponding to the cartesian product of mathematics

Union is a binary operation which creates a relation consisting of all records in either argument relation

Intersection is a binary operation which creates a relation consisting of all records in both argument relations

Difference is a binary operation which creates a relation consisting of all records in the first but not the second argument relation

Join is a binary operation which constructs all possible records that result from matching identical attributes of the records of the argument relations

Divide is a ternary operation which returns all records of the first argument which occur in the second argument associated with *each* record of the third argument

One significant benefit of this manipulation language (aside from its simplicity) is that it has the property of *closure* — that all operands and results are of the same kind (relations) — hence the operations can be nested in arbitrary ways (indeed this property is inherent in any single-sorted algebra).

8.3   Integrity

Integrity in the relational model is maintained simply by specifying — in a purely declarative way — a set of constraints which must hold at all times.

Any infrastructure implementing the relational model must ensure that these constraints always hold — specifically attempts to modify the state which would result in violation of the constraints must be either rejected outright or restricted to operate within the bounds of the constraints.

The most common types of constraint are those identifying *candidate* or *primary* keys and *foreign* keys. Constraints may in fact be arbitrarily complex, involve multiple relations, and be constructed from either the relational calculus or the relational algebra.

Finally, many commercially available DBMSs provide *imperative* mechanisms such as triggers for maintaining integrity — such mechanisms suffer from control-flow concerns (see section 4.2) and are *not* considered to be part of the relational model.

8.4   Data Independence

Data independence is the principle of separating the logical model from the physical storage representation, and was one of the original motivations for the relational model.

It is interesting to note that the *data independence* principle is in fact a very close parallel to the *accidental / essential* split recommended above (section 7.3.2). This is one of several reasons that motivate the adoption of the relational model in Functional Relational Programming (see section 9 below).

8.5 Extensions

The relational algebra — whilst flexible — is a restrictive language in computational terms (it is not Turing-complete) and is normally augmented in various ways when used in practice. Common extensions include:

General computation capabilities for example simple arithmetical operations, possibly along with user-defined computations.

Aggregate operators such as MAX, MIN, COUNT, SUM, etc.

Grouping and Summarization capabilities to allow for easy application of aggregate operations to relations

Renaming capabilities the ability to generate derived relations by changing attribute names

9 Functional Relational Programming

The approach of functional relational programming (FRP[16]) derives its name from the fact that the *essential* components of the system (the logic and the essential state) are based upon functional programming and the relational model (see Figure 2).

FRP is currently a purely hypothetical[17] approach to system architecture that has not in any way been proven in practice. It is however based firmly on principles from other areas (the relational model, functional and logic programming) which *have* been widely proven.

In FRP all *essential state* takes the form of relations, and the *essential logic* is expressed using relational algebra extended with (pure) user-defined[18] functions.

The primary, overriding goal behind the FRP architecture (and indeed this whole paper) is of course *elimination of complexity*.

---

[16]Not to be confused with functional *reactive* programming [EH97] which does in fact have some similarities to this approach, but has no intrinsic focus on relations or the relational model

[17]Aside from token experimental implementations of FRP infrastructures created by the authors.

[18]By *user-defined* we mean *specific to this particular FRP system* (as opposed to pre-provided by an underlying infrastructure).
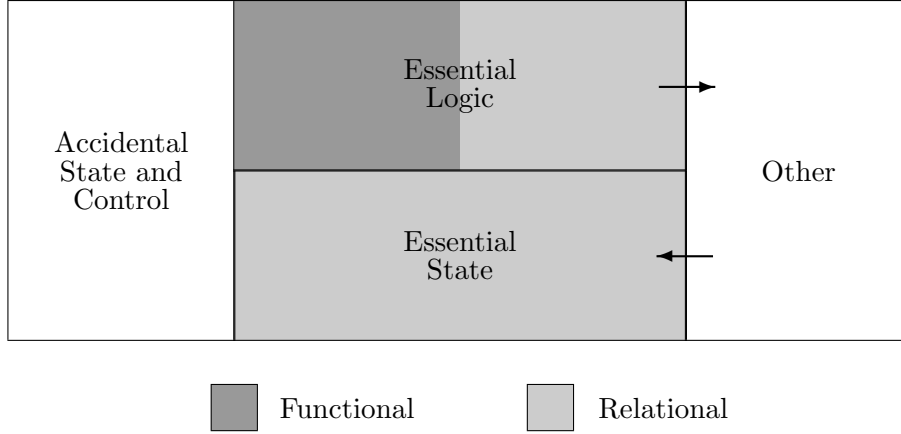
Figure 2: The components of an FRP system (infrastructure not shown, arrows show dynamic data flow)

9.1 Architecture

We describe the architecture of an FRP system by first looking at *what must be specified* for each of the components when constructing a system in this manner. Then we look at what infrastructure needs to be available in order to be able to construct systems in this fashion.

In accordance with the first half of this paper, FRP recommends that the system be constructed from *separate* specifications for each of the following components:

Essential State A Relational definition of the stateful components of the system

Essential Logic Derived-relation definitions, integrity constraints and (pure) functions

Accidental State and Control A declarative specification of a set of performance optimizations for the system

Other A specification of the required interfaces to the outside world (user and system interfaces)

Speaking somewhat loosely, the first two components can be seen as corresponding to "State" and "Behaviour" respectively, whilst the third con-

43

centrates on "Performance". In contrast with the object-oriented approach, FRP emphasises a clear *separation* of state and behaviour[19].

9.1.1 Essential State ("State")

This component consists solely of a specification of the essential state for the system in terms of base relvars[20] (in FRP *all* state is stored solely in terms of relations — there are no exceptions to this). Specifically it is the *names* and *types* of the base relvars that are specified here, not their actual contents. The contents of the relvars (i.e. the relations themselves) will of course be crucial when the system is used, but here we are discussing only the *static* structure of the system.

In accordance with section 7.1.1, FRP strongly encourages that data be treated as essential state *only*[21] when it has been *input directly by a user*.[22]

9.1.2 Essential Logic ("Behaviour")

The essential logic comprises both functional and algebraic (relational) parts. The main (in the sense that it provides the overall structure for the component) part is the relational one, and consists of derived-relvar names and definitions. These definitions consist of applications of the relational algebra operators to other relvars (both derived relvars and the base relvars which make up the essential state).

In addition to the relational algebra, the definitions can make use of an arbitrary set of pure user-defined functions which make up the functional part of the essential logic.

Finally (in accordance with the standard relational model) the logic specifies a set of *integrity constraints* — boolean expressions which must hold at all times. (These can include everything from simple foreign key constraints to complicated multiple-relvar requirements making use of user-defined functions). Any attempt to modify the essential state (see section 9.1.4) will always be subject to these integrity constraints.

Much of the standard theory of relational database design can obviously be used as a guide for the relational parts of these two *essential* components. For example, normalization of the relvars will allow consistent updates (see

---

[19]Equally, traditional OOP pays little attention to the accidental / essential split which was also discussed in section 7.3.2.

[20]see section 8.1.1 for a definition of this term.

[21]aside from the *ease of expression* issue discussed in section 9.1.4.

[22]Other systems connected electronically are considered equivalent to users inputting data for these purposes.

section 9.1.4) to the state to be more easily expressed. Note that — assuming no integrity constraints have been accidentally omitted — normalization does *not* in any way help to preserve the integrity of our relvars — that is after all what the constraints do, and if the constraints are not violated (and the infrastructure must always ensure this) then the relvars have integrity by definition. What *is* true is that more normalized designs do impose *implicit* restrictions, and this can reduce the number of (explicit) integrity constraints that must be specified.

Having raised the issue of design, it is vital to note that absolutely *NO* consideration is paid to performance issues of any kind here. Concepts such as "denormalization for performance" make absolutely no sense in this context because they contain the implicit assumption that the physical storage used will closely mirror the relational structure which is being specified here. This is absolutely not the case (it is only the *accidental state and control* component — see below — which is concerned with efficiency of storage structures).

9.1.3 Accidental State and Control ("Performance")

This component fundamentally consists of a series of isolated (in the sense that they cannot refer to each other in any way) performance "hints". These hints — which should be declarative in nature — are intended to provide guidance to the infrastructure responsible for running the FRP system.

On the *state* side, this component is concerned with both *accidental state* itself and *accidental aspects of* state. Firstly, it provides a means to specify *what state* (of the accidental variety) should exist. Secondly it provides (if desired) a means to specify *what physical storage mechanisms* should be used for storing state (of both kinds) — i.e. the *accidental* aspects of storage. This second aspect is the *flexible mapping* providing physical / logical *data independence* as required by the relational model (section 8).

An example of the first kind of state-related hint might be a simple directive that a particular derived-relvar should actually be stored (rather than continually recalculated), so that it is always quickly available.

An example of the second kind of state-related hint might be that an infrequently used subset of the attributes of a particular relvar (either derived or base) should be stored separately for performance reasons. The use of indices or other custom storage strategies would also be examples of this second kind of state-related hint. The exact types of hint available here will depend entirely on what is provided by the underlying infrastructure.

On the *control* side, recommendations for parallel evaluation of derived-

relvars might be given. Also, declarative hints could be given about whether the derived relvars should be computed eagerly (as soon as the essential state changes), lazily (when the infrastructure is forced to provide them), or some combination of different policies for different relvars.

All hints are incapable of referring to each other, but do refer to the relevant (essential, base and derived) relvars by name.

9.1.4 Other (Interfacing)

The primary consideration not addressed by the above is that of *interfacing with the outside world*.

Specifically, all input must be converted into relational assignments (which replace the old relvar values in the *essential state* with new ones), and all output (and side-effects) must be driven from changes to the values of relvars (primarily derived relvars).

The exact nature of this task is likely to be highly application-dependent, but we can say that there will probably be a requirement for a series of *feeder* (or *input*) and *observer* (or *output*) components. These may well be defined, at least partially, in a traditional, imperative way if custom interfacing is required. There will be cases when it is necessary for a given interfacing component to act in both capacities (if for example a message must be observed and sent to another system, then a response received, recorded and fed back in).

The expectation is that all of these components will be of a *minimal* nature — performing only the necessary translations to and from relations.

Feeders

Feeders are components which convert input into relational assignments — i.e. cause changes to the *essential state*. In order to be able to cause these state changes, *feeders* will need to specify them in some form of *state manipulation language* provided by the infrastructure. At a minimum, this language can consist of just a relational assignment command which assigns to a relvar a whole new relation value in its entirety:

```
relvar := newRelationValue
```

The infrastructure which eventually runs the FRP system will ensure that the command respects the integrity constraints[23] — either by rejecting

---

[23]In fact one implication of this is that it is in fact necessary for the assignment command to support multiple *simultaneous* assignment of several distinct relation values to several

non-conformant commands, or possibly in some cases by *modifying* them to ensure conformance.

Observers

Observers are components which generate output in response to changes which they observe in the values of the (derived) relvars. At a minimum, *observers* will only need to specify the *name* of the relvar which they wish to observe. The infrastructure which runs the system will ensure that the observer is invoked (with the new relation value) whenever it changes. In this way *observers* act both as what are sometimes called *live-queries* and also as *triggers*.

Despite this the intention is *not* for *observers* to be used as a substitute for true integrity constraints. Specifically, hybrid *feeders / observers* should *not* act as triggers which directly update the *essential state* (this would by definition be creating *derived* and hence *accidental* state). The only (occasional) exceptions to this should be of the *ease of expression* kind discussed in sections 7.2.2 and 7.3.1.

Summary

The most complicated scenario when interfacing the core relational system with the outside world is likely to come when the interfacing requires highly structured input or output (this is most likely to occur when interfacing with other systems rather than with people).

In this situation, the *feeders* or *observers* are forced to convert between structured data and flat relations[24].

9.1.5 Infrastructure

In several places above we have referred to the "infrastructure which runs the FRP system". The *FRP system* is the specification — comprising of the four components above, the *infrastructure* is what is needed to execute this specification (by interpretation, compilation or some mixture).

---

distinct relvars — this is to avoid temporary inconsistencies which could otherwise occur with integrity constraints that involved multiple relvars.

[24]Some systems — for example the Kleisli system used in bio-informatics [Won00] — seek to avoid this conversion by providing support for more complex structures such as nested relations. We believe that the simplicity gained from having flat relations throughout the system is worth the effort sometimes involved at the system edges (section 9.2.4 describes some of the rationale behind this).

The different components of an FRP system lead to different requirements on the infrastructure which is going to support them.

Infrastructure for Essential State

1. some means of storing and retrieving data in the form of relations assigned to named relvars

2. a state manipulation language which allows the stored relvars to be updated (within the bounds of the integrity constraints)

3. optionally (depending on the exact range of FRP systems which the infrastructure is intended to support) secondary (e.g. disk-based) storage in addition to the primary (in memory) storage

4. a base set of generally useful types (typically integer, boolean, string, date etc)

Infrastructure for Essential Logic

1. a means to evaluate relational expressions

2. a base set of generally useful functions (for things such as basic arithmetic etc)

3. a language to allow specification (and evaluation) of the user-defined functions in the FRP system. (It does not have to be a functional language, but the infrastructure must only allow it to be used in a functional way)

4. optionally a means of type inference (this will also require a mechanism for declaring the types of the user-defined functions in the FRP system)

5. a means to express and enforce integrity constraints

Infrastructure for Accidental State and Control

1. a means to specify which *derived* relvars should actually be stored, along with the ability to store such relvars *and ensure that the stored values are accurately up-to-date at all times*

2. a *flexible* means to specify physical storage mechanisms to be used by a relvar. This is a vital part of the infrastructure — without it the infrastructure must store relvars in a way which closely mirrors their

logical (essential) definitions, and that inevitably leads to accidental (performance) concerns corrupting the essential parts of the system. Specifically, procedures such as *normalization* or "*de-normalization*" at the logical (essential) level should have no intrinsic performance implications because of the presence of this mechanism.

Infrastructure for Feeders and Observers

The minimum requirement on the infrastructure (specifically on the state manipulation language) from *feeders* is for it to be able to process relational assignment commands (containing complete new relation values) and reject them if necessary. Practical extensions that could be useful include the ability to accept commands which specify new relvar values in terms of their previous values — typically in the form of INSERT / UPDATE / DELETE commands.

The minimum requirement on the infrastructure from *observers* is for it to be able to supply the new value of a relvar whenever it changes. Practical extensions that could be useful are the ability to provide deltas, throttling and coalescing capabilities (if the *observers* are viewed as *live query-handlers*, then these extensions represent potential *query meta-data*).

Another possible extension is the ability to observe general relational expressions rather than just relvars from the essential logic (this is not a significant extension as it is basically equivalent to a short-term addition to the essential logic's set of derived relvars — the only difference being that the expression in question would be anonymous).

Finally, the ability to access arbitrary *historical* relvar values would obviously be a useful extension in some scenarios.

Summary

If a system is to be based upon the FRP architecture it will be necessary either to obtain an FRP infrastructure from a third party, or to develop one with existing tools and techniques.

Currently of course no real FRP infrastructures exist and so at present the choice is clear. However, even in the presence of third party infrastructures there may in fact be compelling reasons for large systems to adopt the custom route. Firstly, the effort involved in doing so need not be huge[25], and

---

[25]A prototype implementation of the essential state and essential logic infrastructure — the most significant parts — was developed in a mere 1500 lines of Scheme. In fact this prototype supported not only the relational algebra but also some temporal extensions.

secondly the custom approach leads to the ability to tailor the *hints* available (for use in the *accidental state and control* component) to the exact requirements of the application domain.

Finally, it is of course perfectly possible to develop an FRP infrastructure in *any* general purpose language — be it object-oriented, functional or logic.

9.2 Benefits of this approach

FRP follows the guidelines of *avoid* and *separate* as recommended in section 7 and hence gains all the benefits which derive from that. We now examine how FRP helps to avoid complexity from the common causes.

9.2.1 Benefits for State

The architecture is explicitly designed to *avoid* useless accidental state, and to *avoid* even the *possibility* of an FRP system ever getting into a "bad state".

Specifically derived state is not normally stored (is not treated as essential state). In normal circumstances[26] hybrid feeders/observers *never* feed back in the exact same data which they observed — they only ever feed in some externally generated input or response. *So long as this principle is observed* errors in the logic of the system can never cause it to get into a "bad state" — the only thing required to fix such errors[27] is to correct the logic, there is no need to perform an exhaustive search through and correction of the essential state. This also means that (aside from errors in the *infrastructure*) the system can *never* require "restarting" / "rebooting" etc.

When it comes to *separation*, the architecture clearly exhibits both the *logic / state* split and the *accidental / essential* split recommended in section 7. An example of what this means is that *you do not have to* think about *any accidental state when concentrating on the logic of your system*. In fact, you do not really have to think about the *essential state* as being state either — from the point of view of the logic, the *essential state* is seen as *constant*.

Furthermore, the *functional* component (of the logic) has *no* access to any state at all (even the essential state) — it is totally referentially trans-

---

The effort involved in this is insignificant when compared to the hundreds of man-years often involved in large systems.

[26]The exception might be in the kind of highly interactive scenario considered in sections 7.2.2 and 7.3.1

[27]We're talking here solely about fixing the system itself — of course FRP can't guarantee that errors in the logic won't escape and affect the real world via *observers*!

parent, can only access what is supplied in the function arguments, and hence offers *hugely* better prospects for testing (as mentioned earlier in section 4.1.1).

Additionally, there are major advantages gained from adopting a relational representation of data — specifically, there is no introduction of subjective bias into the data, no concern with data access paths. This is in contrast with approaches such as OOP or XML (as we saw in section 8.1.2).

Finally, integrity constraints provide big benefits for maintaining consistency of state in a *declarative* manner:

> *The fact that we can impose the integrity constraints of our system in a* purely declarative *manner (without requiring triggers or worse, methods / procedures) is one of the key benefits of the FRP approach. It means that the addition of new constraints increases the complexity of the system only* linearly *because the constraints do not — indeed cannot — interact in any way at all. (Constraints can make use of user-defined functions — but they have no way of referring to other constraints). This is in stark contrast with more imperative approaches such as object oriented programming where interaction between methods causes the complexity to grow at a far greater rate.*

Furthermore, the declarative nature of the integrity constraints opens the door to the possibility of a suitably sophisticated infrastructure making use of them for performance reasons (to give a trivial example, there is no need to compute the relational `intersection` of two relvars at all if it can be established that their integrity constraints are mutually exclusive — because then the result is guaranteed to be empty). This type of optimisation is just not possible if the integrity is maintained in an imperative way.

9.2.2 Benefits for Control

Control is *avoided* completely in the relational component which constitutes the top level of the essential logic. In FRP this logic consists simply of a set of equations (equating derived relvars with the relations calculated by their expressions) which have no intrinsic ordering or control flow at all.

FRP also avoids any *explicit* parallelism in the essential components but provides for the possibility of *separated* accidental control should that be required.

An *infrastructure* which supports FRP may well make use of *implicit* parallelism to improve its performance — but this shouldn't be the concern

of anyone other than the implementor of the *infrastructure* — certainly it is not the concern of someone developing an FRP *system*.

A final advantage (which isn't particularly related to control) is that the uniform nature of the representation of data as relations makes it much easier to create distributed implementations of an FRP infrastructure should that be required (e.g. there are no pointers or other access paths to maintain).

9.2.3 Benefits for Code Volume

FRP addresses this in two ways. The first is that a sharp focus on true essentials and *avoiding* useless accidental complexity inevitably leads to less code.

The second way is that the FRP approach reduces the harm that large volumes of code cause through its use of *separation* (see section 4.3).

9.2.4 Benefits for Data Abstraction

Data Abstraction is something which we have only mentioned in passing (in section 4.4) so far. By *data abstraction* we basically mean the creation of compound data types and the use of the corresponding compound values (whose internal contents are *hidden*).

We believe that in many cases, un-needed data abstraction actually represents another common (and serious) cause of complexity. This is for two reasons:

Subjectivity Firstly the grouping of data items together into larger compound data abstractions is an inherently *subjective* business (Ungar and Smith discuss this problem in the context of Self in [SU96]). Groupings which make sense for one purpose will inevitably differ from those most natural for other uses, yet the presence of pre-existing data abstractions all too easily leads to *inappropriate* reuse.

Data Hiding Secondly, large and heavily structured data abstractions can seriously erode the benefits of referential transparency (section 5.2.1) in exactly the manner of the extreme example discussed in section 5.2.3. This problem occurs both because data abstractions will often cause un-needed, irrelevant data to be supplied to a function, and because the data which *does* get used (and hence influences the result of a function) is *hidden* at the function call site. This hidden and excessive data leads to problems for testing as well as informal reasoning in ways very similar to state (see section 4.1).

52

One of the primary strengths of the relational model (inherited by FRP) is that it involves only minimal commitment to any subjective groupings (basically just the structure chosen for the base relations), and this commitment has only minimal impact on the rest of the system. Derived relvars offer a straightforward way for different (application-specific) groupings to be used *alongside* the base groupings. The benefits in terms of subjectivity are closely related to the benefits of access path independence (section 8.1.2).

FRP also offers benefits in the area of data hiding, simply by discouraging it. Specifically, FRP offers no support for nested relations or for creating product types (as we shall see in section 9.3).

9.2.5   Other Benefits

The previous sections considered the benefits offered by FRP for minimizing complexity. Other potential benefits include performance (as mentioned briefly under section 9.2.1) and the possibility that development teams themselves could be organised around the different components — for example one team could focus on the accidental aspects of the system, one on the essential aspects, one on the interfacing, and another on providing the infrastructure.

9.3   Types

A final comment is that — in addition to a fairly typical set of standard types — FRP provides a limited ability to define new user types for use in the *essential state* and *essential logic* components.

Specifically it permits the creation of disjoint union types (sometimes known as "enumeration" types) but does *not* permit the creation of new product types (types with multiple subsidiary components). This is because (as mentioned above) we have a strong desire to *avoid* any unnecessary data abstraction.

Finally, it probably makes sense for infrastructures to provide *type inference* for the *essential logic*. Interesting work in this area has been carried out in the Machiavelli system [OB88].

10   Example of an FRP system

We now examine a simple example FRP system. The system is designed to support an estate agency (real estate) business. It will keep track of properties which are being sold, offers which are made on the properties,

decisions made on the offers by the owners, and commission fees earnt by the individual agency employees from their successful sales. The example should serve to highlight the declarative nature of the components of an FRP system.

To keep things simple, this system operates under some restrictions:

1. Sales only — no rentals / lettings

2. People only have one home, and the owners reside at the property they are selling

3. Rooms are perfectly rectangular

4. Offer acceptance is binding (ie an accepted offer constitutes a sale)

The example will use syntax from a hypothetical FRP infrastructure (which supports not only the relational algebra but also some of the common extensions from section 8.5) — typewriter font is used for this.

10.1   User-defined Types

The example system makes use of a small number of custom types (see section 9.3), some of which are just aliases for types provided by the infrastructure:

```
def alias address :  string
def alias agent :  string
def alias name :  string
def alias price :  double
def enum roomType :  KITCHEN | BATHROOM | LIVING_ROOM
def enum priceBand :  LOW | MED | HIGH | PREMIUM
def enum areaCode :  CITY | SUBURBAN | RURAL
def enum speedBand :  VERY_FAST | FAST | MEDIUM | SLOW |
VERY_SLOW
```

10.2   Essential State

The essential state (see section 9.1.1) consists of the definitions of the *types* of the base relvars (the types of the attributes are shown in *italics*).

```
def relvar Property :: {address:address price:price
    photo:filename agent:agent dateRegistered:date}
```

```
def relvar Offer :: {address:address offerPrice:price
    offerDate:date bidderName:name bidderAddress:address}

def relvar Decision :: {address:address offerDate:date
    bidderName:name bidderAddress:address decisionDate:date
    accepted:bool}

def relvar Room :: {address:address roomName:string
    width:double breadth:double type:roomType}

def relvar Floor :: {address:address roomName:string
    floor:int}

def relvar Commission :: {priceBand:priceBand
    areaCode:areaCode saleSpeed:speedBand commission:double}
```

The example makes use of six base relations, most of which are self-explanatory.

The Property relation stores all properties sold or for-sale. As will be seen in section 10.3.3, properties are uniquely identified by their *address*. The *price* is the desired sale price, the *agent* is the agency employee responsible for selling the Property, and the *dateRegistered* is the date that the Property was registred for sale with the agency.

The Offer relation records the history of all offers ever made. The *address* represents the Property on which the Offer is being made (by the *bidderName* who lives at *bidderAddress*). The *offerDate* attribute records the date when the offer was made, and the *offerPrice* records the price offered. Offers are uniquely identified by an (*address*, *offerDate*, *bidderName*, *bidderAddress*) combination.

The Decision relation records the decisions made by the owner on the Offers that have been made. The Offer in question is identified by the (*address*, *offerDate*, *bidderName*, *bidderAddress*) attributes, and the date and outcome of the decision are recorded by (*decisionDate* and *accepted*).

The Room relation records information (*width*, *breadth*, *type*) about the rooms that exist at each Property. The Property is of course represented by the *address*. One point worthy of note (because it's slightly artificial) is that an assumption is made that every Room in each Property has a unique (within the scope of that Property) *roomName*. This is necessary because many properties may have more than one room of a given *type* (and size).

The Floor relation records which *floor* each Room (*roomName*, *address*) is on.

Finally, the Commission relation stores *commission* fees that can be earned by the agency employees. The commission fees are assigned on the basis of sale *price*s divided into different *priceBand*s, Property *address*es categorized into *areaCode*s and ratings of the *saleSpeed*. (The decision has been made to represent commission rates as a base relation — rather than as a function — so that the commission fees can be queried and easily adjusted).

10.3 Essential Logic

This is the heart of the system (see section 9.1.2) and corresponds to the "business logic".

10.3.1 Functions

We do not give the actual function definitions here, we just describe their operation informally. In reality we would supply the function definitions in terms of some language provided by the infrastructure.

priceBandForPrice Converts a price into a *priceBand* (which will be used in the commission calculations)

areaCodeForAddress Converts an address into an *areaCode*

datesToSpeedBand Converts a pair of dates into a *speedBand* (reflecting the speed of sale after taking into account the time of year)

10.3.2 Derived Relations

There are thirteen derived relations in the system. These can be very loosely classified as *internal* or *external* according to whether their main purpose is simply to facilitate the definition of other derived relations (and constraints) or to provide information to the users. We consider the definition and purpose of each in turn.

As an aid to understanding, the types of the derived relations are shown in comments (delimited by /* and */). In reality these types would be derived (or checked) by an infrastructure-provided type inference mechanism.

Internal

The ten *internal* derived relations exist mainly to help with the later definition of the three *external* ones.

```
/* RoomInfo :: {address:address roomName:string width:double
                breadth:double type:roomType roomSize:double} */
RoomInfo = extend(Room, (roomSize = width*breadth))
```

The RoomInfo derived relation simply extends the Room base relation
with an extra attribute *roomSize* which gives the area of each room.

```
/* Acceptance :: {address:address offerDate:date bidderName:name
                  bidderAddress:address decisionDate:date} */
Acceptance = project_away(restrict(Decision | accepted == true),
                          accepted)
```

The Acceptance derived relation simply selects the positive entries from
the Decision base relation, and then strips away the *accepted* attribute (the
`project_away` operation is the dual of the `project` operation — it removes
the listed attributes rather than keeping them).

```
/* Rejection :: {address:address offerDate:date bidderName:name
                 bidderAddress:address decisionDate:date} */
Rejection = project_away(restrict(Decision | accepted == false),
                         accepted)
```

The Rejection derived relation simply selects the negative decisions and
removes the *accepted* attribute.

```
/* PropertyInfo :: {address:address price:price photo:filename
                    agent:agent dateRegistered:date
                    priceBand:priceBand areaCode:areaCode
                    numberOfRooms:int squareFeet:double} */
PropertyInfo =
extend(Property,
       (priceBand = priceBandForPrice(price)),
       (areaCode = areaCodeForAddress(address)),
       (numberOfRooms = count(restrict(RoomInfo |
                                       address == address))),
       (squareFeet = sum(roomSize, restrict(RoomInfo |
                                       address == address))))
```

The PropertyInfo derived relation extends the Property base relation
with four new attributes. The first — called *priceBand* — indicates which
of the estate agency's price bands the property is in. The price band of the
```

final sale price will affect the commission derived by the agent for selling the property. The *areaCode* attribute indicates the area code, which also affects the commission an agent may earn. The *numberOfRooms* is calculated by counting the number of rooms (actually the number of entries in the RoomInfo derived relation at the corresponding address), and the *squareFeet* is computed by summing up the relevant *roomSizes*.

```
/* CurrentOffer :: {address:address offerPrice:price
                    offerDate:date bidderName:name
                    bidderAddress:address} */
CurrentOffer =
summarize(Offer,
          project(Offer, address bidderName bidderAddress),
          quota(offerDate,1))
```

The purpose of the CurrentOffer derived relation is to filter out old offers which have been superceded by newer ones (e.g. if the bidder has submitted a revised — higher or lower — offer, then we are no longer interested in older offers they may have made on the same property).

The definition summarizes the Offer base relation, taking the most recent (ie the single greatest *offerDate*) offer made by each bidder on a property (ie per unique *address*, *bidderName*, *bidderAddress* combination). Because both *bidderName* and *bidderAddress* are included, the system supports the (admittedly unusual) possibility of different people living in the same place (*bidderAddress*) submitting different offers on the same property (*address*).

```
/* RawSales :: {address:address offerPrice:price
               decisionDate:date agent:agent
               dateRegistered:date} */
RawSales =
project_away(join(Acceptance,
                  join(CurrentOffer,
                       project(Property, address agent
                                         dateRegistered))),
             offerDate bidderName bidderAddress)
```

For the purposes of this example, sales are seen as corresponding directly to accepted offers. As a result the definition of the RawSales relation is in terms of the Acceptance relation. These accepted offers are augmented (joined) with the CurrentOffer information (which includes the agreed *offerPrice*) and with information (*agent*, *dateRegistered*) from the Property relation.

```
/* SoldProperty :: {address:address} */
SoldProperty = project(RawSales, address)
```

The SoldProperty relation simply contains the *address* of all Properties
on which a sale has been agreed (ie the properties in the RawSales relation).

```
/* UnsoldProperty :: {address:address} */
UnsoldProperty = minus(project(Property, address), SoldProperty)
```

The UnsoldProperty is obviously just the Property which is not Sold-
Property (i.e. all Property addresses minus the SoldProperty addresses).

```
/* SalesInfo :: {address:address agent:agent areaCode:areaCode
                 saleSpeed:speedBand priceBand:priceBand} */
SalesInfo =
project(extend(RawSales,
               (areaCode = areaCodeForAddress(address)),
               (saleSpeed = datesToSpeedBand(dateRegistered,
                                             decisionDate)),
               (priceBand = priceBandForPrice(offerPrice))),
        address agent areaCode saleSpeed priceBand)
```

The SalesInfo relation is based on the RawSales relation, but extends
it with *areaCode*, *saleSpeed* and *priceBand* information by calling the three
relevant functions.

```
/* SalesCommissions :: {address:address agent:agent
                        commission:double} */
SalesCommissions =
project(join(SalesInfo, Commission),
        address agent commission)
```

The SalesCommissions which are due to the agents are derived simply by
joining together the SalesInfo with the Commission base relation. This gives
the amount of *commission* due to each *agent* on each Property (represented
by *address*).

External

Having now defined all the *internal* derived relations, we are now in a posi-
tion to define the *external* derived relations — these are the ones which will
be of most direct interest to the users of the system.

```
/* OpenOffers :: {address:address offerPrice:price
                  offerDate:date bidderName:name
                  bidderAddress:address} */
OpenOffers =
join(CurrentOffer,
     minus(project_away(CurrentOffer, offerPrice),
           project_away(Decision, accepted decisionDate)))
```

The OpenOffers relation gives details of the CurrentOffers on which the owner has not yet made a Decision. This is calculated by joining the CurrentOffer information (which includes *offerPrice*) with those CurrentOffers (excluding the price information) that do not have corresponding Decisions. `project_away` is used here because `minus` requires its arguments to be of the same type.

```
/* PropertyForWebSite :: {address:address price:price
                          photo:filename numberOfRooms:int
                          squareFeet:double} */
PropertyForWebSite = project( join(UnsoldProperty, PropertyInfo),
                              address price photo
                              numberOfRooms squareFeet )
```

The business wants to display the information from PropertyInfo on their external website. However, they only want to show unsold property (this is achieved simply by a `join`), and they only want to show a subset of the attributes (this is achieved with a `project`).

```
/* CommissionDue :: {agent:agent totalCommission:double} */
CommissionDue =
project(summarize(SalesCommissions,
                  project(SalesCommissions, agent),
                  totalCommission = sum(commission)),
        agent totalCommission)
```

Finally, the total commission due to each agent is calculated by simply summing up the *commission* attribute of the SalesCommissions relation on a per *agent* basis to give the *totalCommission* attribute.

10.3.3   Integrity

Integrity constraints are given in the form of relational algebra or relational calculus expressions. As already noted, our hypothetical FRP infrastructure

provides common relational algebra extensions (see section 8.5). It also provides special syntax for *candidate* and *foreign* key constraints. (This syntax is effectively just a shorthand for the underlying algebra or calculus expression).

We consider the standard (key) constraints first:

```
candidate key Property = (address)
candidate key Offer = (address, offerDate,
                       bidderName, bidderAddress)
candidate key Decision = (address, offerDate,
                          bidderName, bidderAddress)
candidate key Room = (address, roomName)
candidate key Floor = (address, roomName)
candidate key Commision = (priceBand, areaCode, saleSpeed)

foreign key Offer (address) in Property
foreign key Decision (address, offerDate,
                      bidderName, bidderAddress) in Offer
foreign key Room (address) in Property
foreign key Floor (address) in Property
```

There are also some slightly more interesting, domain-specific constraints. The first insists that all properties must have at least one room:

```
count(restrict(PropertyInfo | numberOfRooms < 1)) == 0
```

The next ensures that people cannot submit bids on their own property (owners are assumed to be residing at the property they are selling):

```
count(restrict(Offer | bidderAddress == address)) == 0
```

This constraint prohibits the submission of any Offers on a property (*address*) after a sale has happened (i.e. after an Acceptance has occurred for the *address*):

```
count(restrict(join(Offer,
                    project(Acceptance, address decisionDate))
              | offerDate > decisionDate)) == 0
```

The next constraint ensures that there are never more than 50 properties advertised on the website in the PREMIUM price band:

```
count(restrict(extend(PropertyForWebSite,
                      (priceBand = priceBandForPrice(price)))
              | priceBand == PREMIUM)) < 50
```

This is an interesting constraint because it depends (directly as it happens) on a user-defined function (`priceBandForPrice`). One implication of this is that changes to function definitions (as well as changes to *essential state*) could — if unchecked — cause the system to violate its constraints. No FRP infrastructure can allow this.

Fortunately there are two straightforward approaches to solving this. The first is that the infrastructure could treat function definitions as data (essential state) and apply the same kind of modification checks. The alternative is that it could refuse to run a system with a new function version which causes existing data to be considered invalid. In this latter case manual state changes would be required to restore integrity and to allow the system became operational again.

Finally, no single bidder can submit more than 10 offers (over time) on a single Property. This constraint works by first computing the number of offers made by each bidder (*bidderName*, *bidderAddress*) on each Property (*address*), and ensuring that this is never more than 10:

```
count(restrict(summarize(Offer,
                         project(Offer, address bidderName
                                                bidderAddress),
                         numberOfOffers = count())
              | numberOfOffers > 10)) == 0
```

Once the system is deployed, the FRP infrastructure will reject any state modification attempts which would violate any of these integrity constraints.

10.4   Accidental State and Control

The *accidental state and control* component of an FRP system consists solely of a set of declarations which represent performance hints for the infrastructure (see section 9.1.3). In this example the *accidental state and control* is a set of three hint declarations.

```
declare store PropertyInfo
```

This declaration is simply a hint to the infrastructure to request that the PropertyInfo derived relation is actually stored (ie cached) rather than continually recalculated.

```
declare store shared Room Floor
```

This hint instructs the infrastructure to *denormalize* the Room and Floor relations into a single shared storage structure. (Note that because we are able to express this as part of the *accidental state and control* we have not been forced to compromise the *essential* parts of our system which still treat Room and Floor separately).

```
declare store separate Property (photo)
```

This hint instructs the infrastructure to store the *photo* attribute of the Property relation separately from its other attributes (because it is not frequently used).

These three hints have all focused on *state* (PropertyInfo is *accidental state*, and the other two declarations are concerned with *accidental* aspects *of* state). Larger systems would probably also include *accidental control* specifications for performance reasons.

10.5   Other

The *feeders* and *observers* for this system would be fairly simple — *feeding* user input into Decisions, Offers etc., and directly *observing* and displaying the various derived relations as output (e.g. OpenOffers, PropertyForWeb-Site and CommisionDue).

Because of this it is reasonable to expect that the *feeders* and *observers* would require no custom coding at all, but could instead be specified in a completely declarative fashion.

One extension which might require a custom *observer* would be a requirement to connect CommissionDue into an external payroll system.

11   Related Work

FRP draws some influence from the ideas of [DD00]. In contrast with this work however, FRP is aimed at general purpose, large-scale *application* programming. Additionally FRP focuses on a *separate*, *functional*, sub-language and has different ideas about the use of types. Finally the *accidental* component of FRP has a broader range than the physical / logical mapping of traditional DBMSs.

There are also some similarities to Backus' Applicative State Transition systems [Bac78], and to the Aldat project at McGill [Mer85] which investigated general purpose applications of relational algebra.

12　Conclusions

We have argued that *complexity* causes more problems in large software systems than anything else. We have also argued that it *can* be tamed — but only through a concerted effort to *avoid* it where possible, and to *separate* it where not. Specifically we have argued that a system can usefully be *separated* into three main parts: the *essential state*, the *essential logic*, and the *accidental state and control*.

We believe that taking these principles and applying them to the top level of a system design — effectively using different specialised *languages* for the different components — can offer more in terms simplicity than can the unstructured adoption of any single general language (be it imperative, logic or functional). In making this argument we briefly surveyed each of the common programming paradigms, paying some attention to the weaknesses of object-orientation as a particular example of an imperative approach.

In cases (such as existing large systems) where this *separation* cannot be directly applied we believe the focus should be on avoiding state, avoiding explicit control where possible, and striving at all costs to *get rid of code*.

So, what is the way out of the tar pit? What is the silver bullet? . . . it may not be FRP, but we believe there can be no doubt that it is *simplicity*.

References

[Bac78]　John W. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[Bak93]　Henry G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *Journal of Object-Oriented Programming*, 4(4):2–27, October 1993.

[Boo91]　G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings, 1991.

[Bro86]　Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *Information Processing 1986, Proceedings of the Tenth World Computing Conference*, H.-J. Kugler, ed.: 1069–76. Reprinted in IEEE Computer, 20(4):10-19, April 1987, and in Brooks, The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, Chapter 16, Addison-Wesley, 1995.

[Che76] P. P. Chen. "The Entity-Relationship Model". *ACM Trans. on Database Systems (TODS)*, 1:9–36, 1976.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Comm. ACM*, 13(6):377–387, June 1970.

[Cod79] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. on Database Sys.*, 4(4):397, December 1979.

[Cod90] E. F. Codd. *The Relational Model for Database Management, Version 2.* Addison-Wesley, 1990.

[Cor91] Fernando J. Corbató. On building systems that will fail. *Commun. ACM*, 34(9):72–81, 1991.

[Dat04] C. J. Date. *An Introduction to Database Systems.* Addison Wesley, 8th edition, 2004.

[DD00] Hugh Darwen and C. J. Date. *Foundation for Future Database Systems: The Third Manifesto.* Addison-Wesley, 2nd edition, 2000.

[Dij71] Edsger W. Dijkstra. On the reliability of programs. circulated privately, 1971.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[Dij97] Dijkstra. The tide, not the waves. In Peter J. Denning and Robert M. Metcalfe, editors, *Beyond Calculation: The Next Fifty Years of Computing, Copernicus, 1997.* 1997.

[Eco04] Managing complexity. *The Economist*, 373(8403):89–91, 2004.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP-97)*, volume 32,8 of *ACM SIGPLAN Notices*, pages 263–273, New York, June 9–11 1997. ACM Press.

[HJ89] I. Hayes and C. Jones. Specifications are not (necessarily) executable. *IEE Software Engineering Journal*, 4(6):330–338, November 1989.

[Hoa81] C. A. R. Hoare. The emperor's old clothes. *Commun. ACM*, 24(2):75–83, 1981.

[Kow79] Robert A. Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, 1979.

[Mer85] T. H. Merrett. Persistence and Aldat. In *Data Types and Persistence (Appin)*, pages 173–188, 1985.

[NR69] P. Naur and B. Randell. Software engineering report of a conference sponsored by the NATO science committee Garmisch Germany 7th-11th October 1968, January 01 1969.

[OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming, Snowbird, UT*, pages 174–183, New York, NY, 1988. ACM.

[O'K90] Richard A. O'Keefe. *The Craft of Prolog.* The MIT Press, Cambridge, 1990.

[PJ+03] Simon Peyton Jones et al., editors. *Haskell 98 Language and Libraries, the Revised Report.* CUP, April 2003.

[SS94] Leon Sterling and Ehud Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed.* MIT Press, 1994.

[SU96] Randall B. Smith and David Ungar. A simple and unifying approach to subjective objects. *TAPOS*, 2(3):161–178, 1996.

[vRH04] Peter van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, 2004.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52, 1995.

[Won00] Limsoon Wong. Kleisli, a functional query system. *J. Funct. Program*, 10(1):19–56, 2000.