BASICS: Programming Language Aspects (Added 99/11)

November 22, 1999

The author gratefully acknowledges support from the taxpayers of Québec and of Canada who have paid his salary and research grants while this work was developed at McGill University, and from his students (who built the implementations and investigated the data structures and algorithms) and their funding agencies.

Although a lot of programs can be written in the relational and domain algebras as considered so far, they are a formalism rather than a programming language. We have not included any of the advanced programming language facilities, such as procedural and data abstraction, event handling, instantiation and inheritance, processes and concurrency, or even scoping or control structures.

Many proposals have been made and implemented to embed "database" facilities into programming languages, from invocations of DBTG within COBOL or of SQL within C++, to incorporating the relational algebra into modifications of languages such as Pascal, to adding "persistence" to languages such as Smalltalk or C++ to attain "object-oriented databases". The common understanding behind these systems is that databases support querying, through "query languages" which fall short of full programming capability. There is a need for more than query languages, and so these extensions have been made.

This understanding of databases is limited. It goes without saying that a language for secondary storage should offer full programming capabilities, and not be limited to "querying". Regrettably, this has not been obvious to builders of database systems, and, indeed, current vocabulary enshrines the distinction between the "two types" of language. For example, the term "on-line analytical processing" (*OLAP*) had to be coined to describe a supposed new dimension, distinct from "on-line transaction processing" (*OLTP*), to permit thinking about programming as opposed to querying, while, in fact, querying is just a special case of programming.

In the following, instead of making ad hoc bridges between two different worlds, we *unify* database and programming language concepts, obtaining generalizations of both which advance both.

The new facilities we discuss below do not necessarily enhance the power of Aldat. They are mostly implementations of generally useful concepts which save the programmer thought and code.

1 Computations

The most obvious limitation of our formalism so far is that it does not allow parametric abstraction: procedures and functions. If we think about functions the way mathematics does, they are special kinds of relation, namely relations in which the mapping is many-to-one. Since we have a fairly full treatment of relations in the relational algebra, we can hope to find the right perspective on programming language functions so that we can just use existing ideas and syntax.

This perspective is given by programming languages that offer "constraint programming". In these languages, a relationship between different variables can be defined as a predicate and used to calculate more than one subset of the variables given the complementary subset.

For example, the relationship between interest rates compounded over different periods is

$$1 + I = (1 + i)^p$$

where interest rate i compounded p times is equivalent to interest rate I over the whole period. (If p were 12, this relates annual interest, I, to monthly interest, i.) From this, given values for any two of the variables, the third can be found.

This relationship exhibits a symmetry reminiscent of the symmetry of database relations, in which any subset of attributes can be specified in a T-selector and the corresponding tuples (and so the values of the other attributes) found.

We formulate it as a *computation*, a new term for a parametrized expression which can be used to derive more than one of its parameters given values for the others.

comp *IntPerChg*(I, i, p) **is**
$\{I <\!\!-(1 + i) * *p - 1\}$
alt
$\{i <\!\!-(1 + I) * *(1.0/p) - 1\}$
alt
$\{p <\!\!-$ round$(\log(1 + I)/\log(1 + i))\}$;

Instead of attempting to provide an implementation which automatically works out from a single predicate all the functions it gives rise to, we give syntax, **alt**, which allows the programmer to combine heir own solutions. While this relies on the programmer to give correct and consistent alternatives, it avoids the extreme difficulties of trying to do it automatically. (If the constraint programming people find a general solution to the problem, we can adopt it.)

The computation, e.g., *IntPerChg*, has attributes, just as a relation does. Unlike relations, it has a *type*, and can only be used in ways consistent with this type. (A type system in a programming language is a redundancy mechanism used to help detect programmer errors, specifically in the use of parametric abstraction and of built-in operators. Relational operations have so far accepted any relation, and the only use a relational algebra implementation has been able to make of typing is to ensure union-compatibility between join attributes.)

Computations are intended to behave as far as possible like relations. Indeed, "**comp**" can be read as "compressed relation", and *IntPerChg* can be considered to be the infinite relation on all possible values consistent with the relationship.

IntPerChg(

I	i	p)
0	0	1
0.1	0.1	1
:	:	:
0	0	2
0.21	0.1	2
:	:	:

The typing restricts operations on *IntPerChg* so that they produce finite relations as results. (We could specify that a new computation should be produced by any operation that would otherwise lead to an infinite relation, but we avoid it for the same reason we avoid automatic generation of all functions from a predicate. Such a specification would be called "partial evaluation".)

Because a computation can be thought of as a typed relation, we do not need to add new syntax to invoke it. A T-selector will do.

Intint <− [p] **where** $I = 0.12$ & $i = 0.01$ **in** *IntPerChg*;

gives the result

Intint(

p
11

)

The T-selector needed is a very special kind. The attributes in the **where** clause are tested for equality with constant values and the tests are **and**ed together. The attributes in the projection list complement those tested. Because of this, and also because programmers are more familiar with a simpler syntax, we can introduce "syntactic sugar" for this special kind of T-selector.

Intint <− *IntPerChg*{0.12,0.01};

This syntax uses a *positional* convention, based on the order of the attributes in the declaration of the computation. Generally, *IntPerChg* has three positions, separated by commas, e.g., *IntPerChg*{0.12,0.01,}, with blank positions indicating the attribute to be projected. If the attribute to be projected is the last, the comma before it may be omitted, as above. [1] Thus

Intper <− [i] **where** $I = 0.12$ & $p = 12$ **in** *IntPerChg*;
(*Intper* <− *IntPerChg*{0.12,,12};)

giving

Intper(

i
0.00948882

)

and

intper <− [I] **where** $i = 0.01$ & $p = 12$ **in** *IntPerChg*;
(*intper* <− *IntPerChg*{,0.01,12};)

giving

[1] Since computations and relations are closely connected, this syntactic sugar is also available for relations. It gives the effect of *array* lookup. Although the syntax is *defined* to mean the corresponding T-selector, it does not have to be *implemented* that way. Indeed, for relations we can gain much speed from a separate implementation because this special T-selector does not require a scan of the whole relation, as does the general T-selector.

$intper($ I $)$

0.126825

T-selectore are not the only way a computation may be invoked. Since they are effectively relations, they may be used, with restrictions due to type, anywhere relations may. In particular, we can use a natural join to invoke a computation.

$IntPer($

I	p
0.06	12
0.06	24
0.07	12
0.07	24

$)$

$ipc <-$ *IntPerChg* **natjoin** *IntPer*;

$ipc($

I	p	i
0.06	12	0.00486755
0.06	24	0.0024308
0.07	12	0.0056541
0.07	24	0.00282311

$)$

Computations support recursion in the usual way.

comp $gcd(k, m, g)$ **is**
{ $g <-$ **if** $k < 0$ **then** $gcd\{-k, m\}$ **else**
if $m < k$ **then** $gcd\{m, k\}$ **else**
if $k = 0$ **then** m **else** $gcd\{m$ **mod** $k, k\}$
};

Procedures are top-level computations. They take relations as parameters (although so do the computations discussed above, in the context of nested relations) and are invoked as statements in their own right.

comp *ABCjoindecomp*(R, S, T) **is**
{ $T <-R$ **ijoin** S; }
alt
{ $R <-[A, B]$ **in** T;
$S <-[B, C]$ **in** T;
};

The invocation must include syntax for type checking and to specify which **alt** is to be invoked. Accordingly, parameters are invoked prefixed by **in** or **out**.

relation $R(A, B) <-$ {(0,0),(2,0),(0,1),(1,1)};
relation $S(B, C) <-$ {(0,0),(1,1),(0,1),(1,2)};
ABCjoindecomp(**in** R, **in** S, **out** T);
ABCjoindecomp(**out** U, **out** V, **in** T);

These prefixes are not needed in the earlier invocations, by T-selector or join, because the syntax of these makes clear which attributes are input and therefore which are output.

```
comp Closure(TC, Graph, Parent, Child) is
{ temp <- Graph;
  TC <- Graph;
  Closure(in TC, in Graph, in Parent, in Child);
} alt
{ temp <- temp[Child icomp Parent]Graph;
  if [] in temp then
  { TC <- TC ujoin temp;
    Closure(in TC, in Graph, in Parent, in Child)
  };
};                                          ≪ Rebecca Lui, 1996 ≫
```

Figure 1: Transitive Closure by Recursive Computation

If attribute names are used as parameters for a top-level computation, they are **in** or **out** according to need.

```
comp joindecomp(X, Y, Z, R, S, T) is
{ T <-R ijoin S; }
alt
{ R <-[X, Y] in T;
  S <-[Y, Z] in T;
};
joindecomp(out A, out B, out C, in U, in V, out W);
joindecomp(in A, in B, in C, out X, out Y, in W);
```

With this invocation syntax, we can write a recursive computation which is not just the equivalent of a recursive function call in other languages. Figure 1 shows a computation to find transitive closure of a *Graph* by combining edges with paths of length two with paths of length three, and so on to paths of maximum length.

This would be invoked by

```
Closure(out Ancestor, in Parent, in Sr, in Jr);
```

The types of the two **alt** blocks are, respectively, (**out**, **in**, **in**, **in**) and (**in**, **in**, **in**, **in**). So we see that this invokes the first **alt** block first. The first **alt** block invokes the second, which then invokes itself until all paths up to the maximum length have been processed. The code computes *Graph, Graph*^{2}*, Graph*^{3}*, ..* in succession and accumulates the union in *TC*.

Note that *temp* is a local variable, known only within the computation, but that it keeps its value between invocations. It is an "own" variable in Algol 60 parlance (where these first appeared historically), or "static" in C terminology.

Note also that *Closure* does not have to be invoked at top level as shown. If some relation, R, had a nested attribute compatible with *Graph*, *Closure* could be invoked in a T-selector or in a join to find the transitive closure of this relational attribute in every tuple of R.

2 Event Programming

Top-level computations without parameters can be used to implement event handlers. Event programming is asynchronous programming, in which the code is invoked by actions external to the program. Mouse events in a graphical user interface are such actions, leading to responses from the software to interpret them and carry out the interpretations.

Event programming should not be confused with concurrent programming. It is much more straightforward and does not need concurrent processes. In fact, the two basic ideas are

- an event is a system-generated procedure call, and
- an event handler is a procedure.

In databases, the obvious "external" actions to respond to are updates. Here is an inventory example.

relation *Inventory*
(*PartNo, Descr, QOH, Supplier, Thr, ROQ*) <− {
("1", "widget", 23, "Acme", 20, 50),
("2", "gizmo", 97, "Zedco", 10, 30)
};

in which we would like to support sales and consequent reordering if the quantity-on-hand, *QOH*, drops below a pre-set threshhold, *Thr*.

update *Inventory* **change**
QOH <− **if** *PartNo*="1" **then** *QOH* −9 **else** *QOH*;

Inventory

PartNo	*Descr*	*QOH*	*Supplier*	*Thr*	*ROQ*)
1	widget	14	Acme	20	50
2	gizmo	97	Zedco	10	30

SupplyHist

(*PartNo*	*Descr*	*Supplier*	*Ordr*	*Rcvd*	*Date*)
1	widget	Acme	50	DC	981103

SupplyHist records the order we have placed as a consequence, for 50 `widgets` (50 is the reorder quantity, *ROQ*), and must be generated by the code we are going to write.

We also want to be able to record receipts.

update *Inventory* **change**
QOH<−**if** *PartNo*="1" **then** *QOH*+45 **else** *QOH*;

Inventory

PartNo	*Descr*	*QOH*	*Supplier*	*Thr*	*ROQ*)
1	widget	59	Acme	20	50
2	gizmo	97	Zedco	10	30

SupplyHist

(*PartNo*	*Descr*	*Supplier*	*Ordr*	*Rcvd*	*Date*)
1	widget	Acme	50	DC	981103
1	widget	Acme	DC	45	981105

in which `acme` was able to supply only 45 of the 50 ordered, and *SupplyHist* records this.

The code to do this could be written as procedures for sales and receipts, but we do it instead with an event handler invoked automatically by the updates. Figure 2 gives the event

```
comp post:change:Inventory[QOH]() is
{ let Date be today();
  let oldQOH be QOH;
  let Ordr be ROQ;
  let Rcvd be (long dc);
  Reorder <- [PartNo,Descr,Supplier,Ordr,Rcvd,Date]
        where QOH < oldQOH in (
         (where QOH ≤ Thr and Thr < oldQOH in Inventory)
         ijoin [PartNo,oldQOH] in TRIGGER
        );
  let Date be today();
  let Ordr be (long dc);
  let Rcvd be QOH − oldQOH;
  Resupply <- [PartNo,Descr,Supplier,Ordr,Rcvd,Date]
        where QOH > oldQOH in (
         ([PartNo,Descr,Supplier,QOH] in Inventory)
         ijoin [PartNo,oldQOH] in TRIGGER
        );
  SupplyHist <+ Reorder;   SupplyHist <+ Resupply;
};
```

Figure 2: Inventory Event Handler

handler. This is a procedure body with a system-generated "name". The components are **post** to invoke the handler after the update has been done (**pre** is the alternative), **change** so the handler is invoked by a change (**add** and **delete** are the other options), and the names of the relation and attribute(s) changed.

The code creates either *Reorder* or *Resupply* depending on whether the change in *QOH* is negative or positive. This is detected by comparing *QOH* with *oldQOH*. The latter comes from the system-generated relation *TRIGGER*, which contains the old values of the changed relation, *Inventory* in this case. (*TRIGGER* is present in every event handler, but is local and available only internally.) For *Resupply*, further tests are needed, first to check that *QOH* has dropped below *Thr*, and second to check that this is the first time ($oldQOH > Thr$).

Of course, an event handler may contain a further update, even to the relation whose update invoked the handler. This last would amount to a recursive call of the handler, and can be useful. Here is a little exercise to generate the integers from 0 to 4, each a tuple in *iota*.

```
comp post:add:iota() is
{ let nmin1 be (red min of n) − 1;
 let n be nmin1;
 update iota add
   [n] in [nmin1] where nmin1>= 0 in iota;
};
```

The cascade of updates is started by

```
relation start(n) <- {(4)};
update iota add start;
```

3 Instantiation

For a start, we introduce *state* into computations. So far, the computations we have seen have been *functional* in the programming language sense of this word: the same input always gives the same output; there are no "side-effects"[2].

A useful example is a counter. This increments a state variable (*count*) every time it is called. As a computation, it has two modes of operation. The usual mode has only an output, the current value of *count*. The initialization mode has only an input, the value to which *count* is to be set. Because it is a computation, these modes can be implemented as two **alt** blocks.

comp *counter*(*ct*) **is**
 state *count* **intg**;
 { *count* <− *ct*
 } **alt**
 { *count* <− *count* + 1;
  ct <− *count*
 };

The **state** variable persists between invocations of *counter*, and is accessible by all **alt** blocks. This is an idea which commenced with the **own** variables of Algol 60, and continues in the `static` variables of C. (But it is not at all the `static` variable of Java, as we shall see.)

After initialization and a succession of calls,

c0 <− *counter*{0};
c1 <− *counter*{};
c2 <− *counter*{};
c3 <− *counter*{};

we can look at the latest value.

c3(*ct*)
3

The notion of state leads to a new requirement, *instantiation*. Functions (without state) or computations without state can be used for many different purposes without interference. For example, a `sqrt()` function may calculate the square root of 4 or of 10 or of π. (This may even happen at the same time for different programs: while program P is finding `sqrt(2.0)`, program Q may be calculating `sqrt(`e`)` and there will be no problem, because *re-entrant code* is usually used for "library" functions such as this.

However, a program cannot alternate invocations of *counter* to count apples on one hand and oranges on the other, unless special provisions have been made for the two different *count*s not to get mixed up: the state is not "re-entrant" and a copy must be made for each use.

In an "object-oriented" programming language, a keyword, **new**, is used to make new instances. This is a low-level approach. Dealing with one "object" at a time is akin to dealing with one tuple at a time[3]. Since Aldat has no notion of "tuple", it should also have no notion

[2]We have hitherto focussed on the many-to-one aspect of functions, without considering that they are "memoryless". This is the important aspect here. If the code in a parametric abstraction (loosely called a "function" in some programming languages) remembers its previous invocations through a state, it is not many-to-one. The same input values can result in different output values as a consequence of the history.

[3]The simple-minded mappings between relational and "object-oriented" databases exactly identify an "object" with a tuple, which is hardly necessary, given facilities to deal with whole subsets of the tuples in a relation, as we said about diagrams in section 4 of chapter 1.1.

of "object", and so we have no **new** operator.

The original intention of "object-orientation" was to be able to instantiate large numbers of states, such as might be used to describe the atoms in a kinetic model of a gas or the cells in an organism. Instantiation in connection with relations lends itself very easily to such a requirement, as we shall see.

But first we look at a mechanism for data abstraction. This is the second of the two pillars of "object-orientation", along with state and instantiation. It gives the ability to isolate a module of code from all other code in a program except through a specified interface. This is a list of names, often of functions and procedures, sometimes of ordinary variables, which the module "exports" and allows other modules to see and use. It also is echoed in the using module, in a similar list which specifies the subset of these names that it will use. The rest of the names within the supplying module, not to mention all the code, are *hidden* from other modules. This hiding is central to data abstraction [Par71] because it prevents any exploitation of the supplying module except through the interface, including uses that take advantage of special features of the implementation. Such modules are called *abstract data types*.

It has been argued [AM84] that special syntax is not needed for these modules, provided that "first-class" procedures exist and that they can be made persistent—to endure beyond the lifetime of the program that created them. A first-class data type is one which can be used anywhere any other data type can be used, in particular as return values from other procedures. In many programming languages, integers, say, are first-class, but functions are not.

The argument goes that the formal parameter list of a procedure and the actual paremeter list of the invocation provide fine export and import lists, and can include functions and further procedures if procedures are first-class, and specifically if they can be created within the procedure that serves as the abstract data type and can be returned to user procedures through the parameter lists. Futhermore, if the abstract data type is a persistent procedure, it can be placed in a library, or become a library, and its interface procedures and variables made available for the long term.

Persistence is not a problem for Aldat: everything (i.e., relations, e.g., computations) is potentially persistent. First-class computations are not a problem either: these are just computations, i.e., relations, which are attributes of other computations, i.e., relations, which is to say, they are nested. So in figure 3, we create a *class*, which is an abstract data type with state, using no new or special syntax.

Here, the computation *ba* is the class, with "methods" (computations) *BALANCE* and *DEPOSIT*. There is only one **alt** block in *ba*, so it is the ordinary class of most "object-oriented" programming. The *BALANCE* method simply returns the value of the state variable *bal*, which is a familiar and basic use of methods. The *DEPOSIT* method has two **alt** blocks and so is more unusual. The first **alt** block uses the parameter to change the balance. (The second **alt** block uses a second state variable, *oldbal*, to figure out what the last deposit was, and is a little redundant, except that it highlights the temptation computations present of trying to think about a procedure from all angles, not a harmful impulse at all.)

Now consider the issue of instantiation. We would like to have *bal* (and *oldbal*) states for each of a number of diffrent accounts, such as might be listed in a relation.

relation *accts*(*ACCNO, CLIENT*) <– { `(1729,"joe"),(4104,"sue")` };

We instantiate by putting the computation, *ba*, and the relation, *accts*, together. How else

```
domain DEPOSIT comp(DEP);
domain BALANCE comp(BAL);
comp ba(BALANCE, DEPOSIT) is
 state bal intg;
 state oldbal intg;
 { comp DEPOSIT(DEP) is
  { oldbal <- bal;
    bal <- bal + DEP;
  } alt
  { DEP <- bal - oldbal;          };
  comp BALANCE(BAL) is
  { BAL <- bal;           };
  bal <- 0;
  oldbal <- 0;
 };
```

Figure 3: A Class For Bank Accounting

but by a natural join?

accounts <− *accts* **ijoin** *ba*;

The result is the relation *accounts*(*ACCNO*, *CLIENT*, *BALANCE*, *DEPOSIT*). Two of these attributes are computations, which we can think of as nested relations. The result is the Cartesian product of *ba* and *accts*, because they share no attributes. There is no point in repeating the whole "value" of *BALANCE* and *DEPOSIT*, which is just their code. However, we do want each tuple of the result to have an instantiated state, consisting of *bal* and *oldbal*, so the **ijoin** between a relation and a class is implemented to do just this.

Except for the fact that *bal* and *oldbal* are hidden, i.e., inaccessible except, partially, through the interface names, *BALANCE* and *DEPOSIT*, the new relation can be thought of as

accounts			
(*ACCNO*	*CLIENT*	*bal*	*oldbal*)
1729	joe	0	0
4104	sue	0	0

where the 0 values come from the initialization code in the class, *ba*.

Reading the balance of each account is easy.

acctbal <− [*ACCNO*, [*BAL*] **in** *BALANCE*] **in** *accounts*;

On the other hand, if we want to make a deposit, we may not use domain algebra, because the domain algebra is functional (in the sense that it admits no side-effects). We must use **update** syntax instead, because we are changing the relation.

update *accounts* **change** *DEPOSIT*(100) **using where** *ACCNO* = 4104 **in** *accounts*;

We can use this to write a more elaborate procedure to transfer money between two accounts.

```
comp transfer(FROMACC, TOACC, AMT) is
 { update accounts change DEPOSIT(-AMT)
    using where ACCNO=FROMACC in accounts;
   update accounts change DEPOSIT(AMT)
```

using where *ACCNO* = *TOACC* **in** *accounts*;
}

Here we use it to move $50 between the two accounts.

transfer(**in** 1729, **in** 4104, **in** 50);

How would we write code to require a password to change or read any particular account?

4 Inheritance

With instantiation, we are well on the way to "object oriented" databases. Since "objects" are at the same low level of abstraction as tuples, and since Aldat has no notion of tuple, it is appropriate that we use higher-level terminology. Instances imply *classes* which they instantiate, so we will term what we are aiming at "class programming" for "class databases".

Classes in turn imply subclasses, and subclasses imply *inheritance*. This is a further code-saving mechanism which allows members of a subclass automatically to acquire the attributes of the parent class. Here is s simple example.

relation *Couch*(*Id, Length, Width*);
relation *Chair*(*Id, Base*);
relation *Furniture*(*Id, Manuf*);
Couch **isa** *Furniture*;
Chair **isa** *Furniture*;

Here, *Couch*es and *Chair*s form subclasses of *Furniture*. Although they do not explicitly have all the attributes of *Furniture*, they plainly should: if any piece of furniture has a manufacturer, so does any chair. *Couch*es and *Chair*s each have additional attributes, not shared by other furniture.

The significance of the **isa** syntax is that it makes the common attribute, *Manuf*, available to the inheriting subclasses. Any reference to this attribute in connection with *Couch*es, say, will automatically acquire the appropriate value from *Furniture*. Here, for example, is a projection.

[*Manuf*] **in** *Couch*

The effect of this is defined to be

[*Manuf*] **in** (*Couch* **natjoin** *Furniture*)

As far as *Couch* and *Chair* are concerned, *Manuf* can be thought of as a virtual attrubute, but, unlike those of the domain algebra, one whose value is taken from an actualization elsewhere. Figure 4 shows this viewpoint.

Not only can **isa** be *defined* in this way, it can also be *implemented* as a natural join. However, many "object-oriented" applications of this sort are *low-activity* (meaning that only a very small proportion of the tuples in the relation are needed by the transaction) and so better implemented by pointer-dereferencing instead of by the full join. That is okay: we have syntactic sugar (**isa** instead of **ijoin**) for a special case, and we can also provide a special implementation for the syntactic sugar. The *definition* in terms of joins makes the meaning precise, but does not oblige us to implement inheritance in this way.

We can extend the syntax in the expected way.

relation *Couch*(*Id, Length, Width*);
relation *Furniture*(*Fid, Manuf*);

Furniture		*Couch*			
(*Id*	*Manuf*)	(*Id*	*Length*	*Width*)	*Manuf*
1	Mobel	1	15	5	Mobel
2	Furn	2	17	5	Furn
3	Mobel	3	18	6	Mobel
21	Mobel				
22	Furn	*Chair*			
		(*Id*	*Base*)	*Manuf*	
		21	4	Mobel	
		22	5	Furn	

Figure 4: An Inherited Attribute

Couch [*Id* **isa** *Fid*] *Furniture*;

which permits

[*Manuf*] **in** *Couch*

to be syntactic sugar for

[*Manuf*] **in** (*Couch* [*Id* **natjoin** *Fid*] *Furniture*)

We note that only syntactic elements enter these definitions. This leaves the semantics unconstrained. There are two semantic constraints normally associated with inheritance, and obeyed by the above examples. First, the join attribute (*Id, Fid*) is a key of both parent and subclass. Second, there is the *inclusion dependence*,

[*Id*] **in** *Couch* $\subseteq$ [*Fid*] **in** *Furniture*

which asserts the subset relationship.

The syntactic definitions, above, guarantee neither of these semantic constraints. In the interest of leaving this generalization open for future interpretation and exploitation, we do not impose the constraints or give the guarantee.

5 Scoping

6 Concurrency

References

[AM84] M. P. Atkinson and R. Morrison. *Persistent First Class Procedures are Enough*, volume 181 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1984.

[Par71] D. L. Parnas. Information distribution aspects of design methodology. In *Proceedings of the 1971 IFIP Congress*, pages 339–44, Amsterdam, 1971. North-Holland Publ. Co.