

```

=> EXEC A
};
...

```

Note this one does not use A's tuple space, because if A hangs, any thread waiting for its tuples would also hang so that it would not be able to return NIL to kill A. Also note that, like in example c., if a number of parallel tasks are to be monitored and possibly killed, then a separate, parallel Boolean guard is needed for each, because only then can each independently return a NIL to kill the attached speculative object.

8. Object as Function Families

We suggest another application of objects, as function families. Consider the example of statistical averages: given a set of values X_1 to X_n , we compute weighted mean M , variance V and standard deviation S , using the weights W_1 to W_n (it is assumed the weights add up to 1):

$$M = \sum_{i=1}^N W_i X_i$$

$$V = \sum_{i=1}^N W_i (X_i - M)^2$$

$$S = V^{1/2}$$

Thus, we have several functions that share a common set of parameters W , which are initialized for each run by calling the following class:

```

{ CLASS Averages
  < - N : INTEGER,
    W : ARRAY [1..N] OF REAL;
  PUBLIC
  { FUNCTION Mean : REAL
    < - X : ARRAY [1..N] OF REAL;
    { FOR Sum = 0.,

```

```

        I = 1 TO N RETURN Sum
    DO CONTINUE (Sum + W[I] * X[I], -)
} }
{ FUNCTION Var : REAL
  < - X : ARRAY [1..N] OF REAL,
    Sum : REAL;
  { IF NULL (Sum) => Sum = Mean (X) }
  { FOR Square = 0.,
    I = 1 TO N
    RETURN Square - Sum*Sum
  DO { CONTINUE (Square + W[I] * X[I] * X[I], -)
}} }
{ FUNCTION StDev : REAL
  < - X : ARRAY [1..N] OF REAL;
  SQRT (Var (X))
} }

```

A BaLinda K FOR loop is actually a function, with the FOR..DO part establishing the initial values of the arguments, plus perhaps the changes to an argument with each recursive call (increment I by 1), the termination condition (I > N) and the result returned upon termination (Sum). CONTINUE specifies a recursive call and new argument values.

Note that the body of the class is empty: a call on Averages, like $Z = \text{Averages}(\text{Length}, \text{Weights})$ merely creates an object instance containing the three functions Mean, Var and StDev, and the parameters N and W. There is no need to perform any execution, but only to establish the execution environment and return it. Afterwards, we can call $Z.\text{Mean}(Y)$ to compute the weighted mean of Y, $Z.\text{StDev}(Y)$ to compute its standard deviation, and so on. One can easily imagine having additional functions for regression, analysis of variance, factor analysis, interpolation, curve fitting, etc.

A feature of our method is that the functions can change the parameters as computation proceeds, so that the family of functions gradually evolve in light of new information, e.g., initially all the weights of the average functions are constant ($1/N$), but after some data sets have been processed, it is found that certain points tend to be outliers (i.e., deviate significantly more from the

mean than others) and should be given lesser weights. In other words, $X_i - M$ is calculated for each run and is used to modify W_i . As another example, consider the possibility of a family of functions that depend on a common branch table. As execution proceeds, information is collected to modify the content of the table and change the behaviour of the functions. Such a scheme may be considered as a form of self-modifying code, but within a well structured object and function framework and the primary purpose here is to provide functions. While the same can be done with global parameters and call arguments, the object structure for function families provides a more convenient and secure framework.

To implement the idea, we can use active objects, whose body executes concurrently with calls on the object methods, and interacts with them by the exchange of tuples. The object body acts as the execution monitor of the function family, continuously resetting the parameters to modify the behaviour of the family members. Applying these ideas to the average functions, we get

```
{ CLASS Averages < - N : INTEGER;
  LOCAL
  {W, V} : ARRAY [1..N] OF REAL,
  NN : REAL = FLOAT (N),
  { PROCEDURE Loop;
    LOCAL Vr : REAL;
    OUT ('Evolve, NIL, 0);
    IN ('Change, T ? Vr);
    W[I] = W[I] * (NN-1)/NN
      + (Vr-V[I])/(Vr * (NN-1.) * NN);
    Loop };
  PUBLIC
  { FUNCTION Mean : REAL
    < - X : ARRAY [1..N] OF REAL;
    LOCAL M: INTEGER,
      B: BOOLEAN;
    IN ('Evolve, NIL ? M);
    OUT ('Evolve, NIL, M + 1);
    { FOR Sum = 0.,
```

```

        I = 1 TO N RETURN Sum
    DO CONTINUE (Sum + W[I] * X[I], -)
    }
    IN ('Evolve ? B, M)
    OUT ('Evolve, B, M - 1)
    }
{ FUNCTION Var : REAL
    < - X : ARRAY [1..N] OF REAL,
        Sum : REAL;
    LOCAL Temp : REAL,
        M: INTEGER
    { IF NULL (Sum) => Sum = Mean (X) }
    IN ('Evolve, NIL ? M);
    OUT ('Evolve, T, M)
    { FOR Square = 0.,
        I = 1 TO N
            RETURN Temp = Square
        DO { Temp = X[I] - Sum;
            Temp = Temp * Temp * W[I];
            V[I] = Temp;
            CONTINUE (Square + Temp, -)
        } }
    IN ('Evolve, T ? M)
{ IF M > 0
    => { OUT ('Evolve, T, M - 1);
        IN ('Evolve, T, 0)
    } }
    OUT ('Change, T, Temp);
    Temp }
{ FUNCTION StDev : REAL
    < - X : ARRAY [1..N] OF REAL;
    SQRT (Var (X, NIL))
}
{ FOR I = 1 TO N DO W[I] = 1./NN }
Loop }

```

Initially the weighting factor $W[I]$ is constant for all I . After each use of the variance function, an Evolve tuple is emitted to cause the object body to execute one loop, in which the contribution of each point to the variance, $V[I]$, is used to adjust $W[I]$. A point that contributes more would have a smaller $V_r - V[I]$. When summed over I , $(V_r - V[I])/Var$ produces $N - 1$; hence, $(V_r - V[I])/Var(N - 1)$ sums to 1 and satisfies the property of weighting factor. We then combine $(N - 1)W[I]/N$ with $(V_r - V[I])/V_r N(N - 1)$ to produce a new weighting factor array.

The idea of evolving function families can also be applied to linear feedback systems and recursive estimation using the Kalman filters, but these are not discussed here.

9. Performance Information

Active objects, explained here using BaLinda K, have been given efficient implementations in a BaLinda C++ system operational on a shared memory SUN multi-processor and a distributed memory Fujitsu AP3000 MPP system. The following four tables show the performance data collected for some common object operations and for the Queens program. Noteworthy points are

- (a) While object creation cost is fairly reasonable, in view of the need to attach private tuple spaces and establish object directory entries, but tuple operations on remote objects appear to be rather high; it remains to be seen whether the object search can be shortened and the tuple access speeded up for the most common cases.

Test Case	Performance
Creating a local object	0.54 ms
Creating a remote object	0.77 ms
Object's private tuplespace operation	0.45 ms

- (b) Multi-threaded versions of the program, even when running on a single processor, out performs the sequential version, presumably because of better code and data modularity; in particular, smaller modules of code and data fit into the cache better, thus reducing memory access time.
- (c) The C program, in which all threads share a single tuple space, is considerably worse than the C++ program in which child objects share the