

change from one state to the other, hence enabling or disabling concurrent processing, have limited usefulness in general, because this fails to allow for an object accepting some calls but not others, such as an empty stack object accepting push calls but not pop calls.

In [9], three cases of inheritance anomaly were posed.

- (a) *State Partitioning*. Suppose we have a class Stack and a sub-class DStack is defined with a new method PPop that takes the top only if there are two elements in the stack. Previously, we need to distinguish between the Empty and NonEmpty states of stack, permitting Pop calls only if a stack object is in NonEmpty state, but in a DStack object, the NonEmpty state is sub-divided into two: the One state which permits a Pop but not a PPop, and the More state in which both are permitted. A Push now changes One state to More state and Empty state to One state, while Pop changes states in reverse; hence, the introduction of the sub-class causes the redefinition of the methods of the parent class, instead of simply inheriting them, hence an inheritance anomaly. This is because states controlling the behaviour of the child class are partitions of the states of the parent class, whose methods must change to recognize these.
- (b) *History Sensitivity*. In this, the child class EStack has a QPop method that cannot be executed after a Push from the same object. Again, parent class methods must be modified to change an object between the AfterPush and AfterPop states.
- (c) *State Modification*. This concerns the introduction of orthogonal states with a sub-class that may lock an object and hence affect the processing of parent class methods.

These difficulties do not arise if parallel entries into an object are possible, because in the sub-class pre- and post-processing can be carried out with parent class methods to deal with the new situations, whereas with conditional entries, the condition and the parent method form an integrated unit. Again this reinforces the earlier discussed need that, with multiple threads in an object, some atomic structure must then be available in objects to enforce sequentialization, in shared access of wanted results or in suspending to await relevant conditions. We now discuss this issue.

6. Using BaLinda Objects

As explained earlier, the BaLinda Lisp/K parallel languages provided classes and objects within a functional framework [11, 12]. An efficient compiler for

BaLinda Lisp, including good load balancing and tuplespace distribution, is now available for a PC/Transputer system, a multi-processor SUN workstation and an IBM SP2 system. It was subsequently given a new user interface, resulting in the K version of the language which superficially resembles C. (“K” is “hard C”.) BaLinda C/C++ are also operational but examples written in these are more difficult to explain, so that K is used here.

As discussed earlier, a class call returns a pointer to the execution environment, which may contain public functions or methods. Calling such a public function causes an object reentry, and its execution changes the object’s state that persists even after the function returns. Objects in BaLinda K are not in themselves atomic, and multiple threads can be attached to methods (or even the same method) in a single object. However, each object has a private tuple space, and atomic execution can be enforced within an object using tuples. In short, an object offers a framework for atomicity, but is itself not atomic.

To show an example, below is a stack object definition. A call on Stack creates an object with entry points for the public functions Push and Pop:

```

...
{ CLASS Stack;
  LOCAL ...;
  PUBLIC { PROCEDURE Push < - New;
    LOCAL Pointer;
    IN ('Stack, - ? Pointer);
    OUT ('Stack, T, CONS (New, Pointer))
  },
  { FUNCTION Pop;
    LOCAL Pointer;
    IN ('Stack, T ? Pointer);
    OUT ('Stack, NOT NULL (&(Pointer)),
        &(Pointer));
    $(Pointer)
  };
  OUT ('Stack, FALSE, NIL)
};

```

```

X = Stack;
...
X.Push (something);
...
Y = X.Pop;
...

```

By having the stack pointer in a tuple, parallel tasks are forced to access the stack one at a time. If a Pop call on an empty stack occurs, the call suspends on the IN, until another task makes a Push call. Note that \$ is the Lisp CAR, taking the head of a list, while & takes the remainder, i.e., CDR. Each object has a private tuple space, so that its IN/OUT/RD operations only affect its own space. The tuple functions may be declared PUBLIC, so that another object may use X.IN, etc., to access the tuples of object X. The global tuple space is accessed from inside an object using .IN, .OUT, etc.

Because objects are created by a call/return on a class, there is no need for a make-instance function. Initial values are assigned by defining them in the LOCAL/PUBLIC statements, or by passing over appropriate arguments at object creation. The creation of an object can be done together with the attachment of a parallel thread, through a command like EXEC X = Stack (Y); which returns an object pointer to X as soon as the environment the class execution is established. The main program and the object now execute in parallel, and X is an active object rather than a passive one. If the object body of X suspends on an IN, resumption of X occurs immediately after the wanted tuple is provided. Normally, such a tuple would be provided from outside the object by a call on a method in the object. The body can repeatedly suspend to await tuples that trigger further processing. Thus, an active object may have its body's execution being observed or controlled from the outside through calls on its methods.

To take a simple example, we wish to compute the absolute values of a list of numbers and observe from time to time the percentage of negative values encountered. The list may be very long, or even unbound, e.g., the numbers are being produced and added at the end of the list while the computation is proceeding.

```

{ CLASS Absolute < - Alist;
  LOCAL { Total, Zeros, Negative } : INTEGER = 0,
  { PROCEDURE Loop < - Alist;
    IN (? Total, Zeros, Negative);

```

```

    { IF NULL (Alist)
      =>;
      $(Alist)<0
      => { Negative = Negative + 1;
          SET$ (Alist, -$(Alist))
          }
      $(Alist) == 0
      => Zeros = Zeros + 1
    }
    Total = Total + 1;
    OUT (Total, Zeros, Negative);
    Loop (&(Alist))
  };
PUBLIC { FUNCTION Extract < - Which;
  LOCAL Total, Zeros, Negative;
  RD (? Total, Zeros, Negative)
  { CASE < - Which;
    '+ => (Total-Zeros-Negative)/Total;
    '0=> Zeros/Total;
    '- => Negative/Total
  } };
  OUT (0,0,0);
  Loop (Alist)
}

```

After EXEC X = Absolute (Y), X.Extract would return one of three relevant percentages. The values are taken from a tuple in order to ensure consistency.

7. Speculative Processing

Speculative processing is a method to maximize the utilization of the processing capacity of a parallel system, by initiating tasks before knowing whether they will be actually needed, provided there is idle capacity in the system. If subsequently the results are indeed required, they may be immediately made