Concurrent Meld

Gail E. Kaiser[1]
Columbia University
Department of Computer Science
New York, NY 10027

Our original goal was to design a programming language for writing software engineering environments. The most important requirements were reusability and the ability to integrate separately developed tools [1]. Our scope was later expanded to general applications, and then to parallel and distributed systems. Our current focus is on 'growing' distributed software environments and tools, that is, building a core environment or tool assuming a long-term evolution path.

MELD is a multiparadigm language that combines object-oriented, macro dataflow, transaction processing and module interconnection styles of programming [2]. The most unusual aspect is the dataflow at the source level among the inputs and outputs of statements. Classes may define constraints in addition to instance variables and methods, which are triggered by changes to instance variables and interleaved along with the statements of a method. MELD's constraints are unidirectional, and similar in purpose to active values.

```
CLASS C ::= a, b, d: t

    /* constraint */
    a := F(b)          /* 2nd */

    /* method */
    O (c, e: t) -->
    { b := G(c)        /* 1st */
      d := H(e) }      /* any time */
```

Figure 1: Parallel Block

```
CLASS D ::= a, b, d: t

    /* methods */
    M () -->
    { b := G(d) }      /* 2nd */

    N (c: t) -->
    { a := F(b)        /* 3rd */
      d := H(c) }      /* 1st */
```

Figure 2: Concurrent Methods

Figure 1 gives a trivial example. When a message O is received by an instance of class C, the two statements in method O and any constraints affected by either statement are executed in dataflow order. This means that instance variable b is computed by O as a function of argument c, and only then the constraint recomputes a as a function of the new value of b. Simultaneously with either of these computations, or before or after or in between, d is assigned to the result of a function of argument e; there is no data dependency between this statement and either the other statement in the method or the constraint.

MELD's multiple paradigms lead to three granularities of concurrency:

- Statements from the macro dataflow for fine to medium grain concurrency within a method or among methods. This permits a smaller granularity than dataflow among the inputs and outputs of methods. Atomic methods, which provide a low-level form of concurrency control, force a larger granularity.

[1]Supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, DEC, IBM, Siemens, and Sun, by the Center of Advanced Technology and by the Center for Telecommunications Research.

- Objects for medium to large grain parallelism, with synchronous and asynchronous message passing among remote or local objects. Many other concurrent object-oriented languages provide synchronous or asynchronous message passing, but not both.
- Atomic transactions for high-level concurrency control among users (including tools where interleaved operation is inappropriate). Most other object-based systems provide only one form of concurrency control.

A method may be invoked synchronously or asynchronously. In the synchronous case, the caller waits for return with respect to its own thread of control. In the asynchronous case, the caller continues and the invocation creates a new thread of control. Programs may involve an arbitrary number of threads created dynamically during program execution. Several threads may operate within the same address space and one thread may, in effect, operate across multiple address spaces. In either case, the invoked method runs concurrently with any other methods currently active on the same object. These other methods may be reading and writing the same instance variables. The default synchronization among such methods is done by dataflow, as within a single method.

There is a serious problem with this approach. Figure 2 operates as indicated if M and N happen to begin execution at exactly the same time, due to simultaneous arrival of messages M and N from other objects. d is computed from the value of c given as argument to N, b is computed from the new value of d, and a from the new value of b. However, if message N arrives a bit later than M, then b is computed from the old value of d, and then the new value of d is computed from the argument c and a is computed from the new value of b. Both statements in N could be executed concurrently since there is no dataflow between them. On the other hand, if message M arrives a bit later than N, then b may be computed from either the old or new value of d and a may be computed from either the old or new value of b. This is obviously rather bewildering for the programmer, since it is necessary that the resulting computation be deemed 'correct' in all of these cases.

We support this is to permit maximum concurrency for those applications (and programmers!) that can handle the non-determinism. Our intuition is that non-determinism will not be a problem in many cases. The programmer has in mind a parallel algorithm where he thinks in terms of the dataflow necessary to produce the correct solution. He typically uses a conventional language, such as Fortran or C, to implement his algorithm. The parallelizing compiler must then uncover the parallelism again using dataflow techniques. We avoid this hide-and-seek by allowing the programmer to make the dataflow explicit.

```
CLASS C ::= a, b, d: t

    /* constraint */
    a := F(b)          /* after 1st */

    /* method */
    O (c, e: t) -->
    [ b := G(c)        /* 1st */
      d := H(e) ]      /* 2nd */
```

Figure 3: Sequential Block

```
CLASS D ::= a, b, d: t

    /* methods */
    M () -->
    [ b := G(d) ]      /* 1st */

    N (c: t) -->
    [ a := F(b)        /* 2nd */
      d := H(c) ]      /* 3rd */
```

Figure 4: Concurrent Methods

There are many programs, however, written without cognizance of the dataflow. We also support these programmers with sequential blocks and atomic blocks. Figure 3 shows method O of class C written as a sequential block rather than a parallel block (i.e., a parallel block is enclosed in curly braces and a sequential block in square brackets). In this example, all of method O executes in the order in which the

statements are written. Any constraints whose inputs are changed during the execution of O are triggered as before.

Sequential blocks remove concurrency within methods, making them easier to write without the need for the single-assignment mindset, but do not affect concurrency among methods. Figure 4 indicates the ordering if M and N happen to start at the same time. b is computed from the old value of d, a from the new value of b and then the new value of d from the argument c. But if there is a race condition, a may see the new value of b or b may see the new value of d, but not both.

```
CLASS C ::= a, b, d: t

    /* constraint */
    a := F(b)          /* 2nd */

    /* method */
    O (c, e: t) -->
    ( b := G(c)        /* 1st */
      d := H(e) )
```

Figure 5: Atomic Block

```
CLASS D ::= a, b, d: t

    /* methods */
    M () -->
    ( b := G(d) )      /* 1st or 2nd */

    N (c: t) -->
    ( a := F(b)        /* 1st or 2nd */
      d := H(c) )
```

Figure 6: Concurrent Methods

In Figure 5, method O executes atomically with respect to the receiver object. Atomic blocks are indicated with parentheses. Both b and d are updated, and only after O terminates is the constraint triggered. The two statements in O may themselves be executed in either sequential or dataflow order, since it is necessary to include them within an inner sequential or parallel block, not shown, if the atomic block is used. In this case it does not matter.

Methods M and N are serialized in Figure 6, so they appear atomic to each other. We currently support pessimistic concurrency control by locking the object at the computational grain size of individual methods, or a block within a method. We are integrating real transactions that cut across methods and objects/ We use distributed optimistic concurrency control with multiple versions, since we expect a majority of read-only transactions (in our primary application domain of distributed software development tools). Atomic blocks using validation rather than locking will be surrounded by angle brackets rather than parentheses, and may be either sequential or parallel; `begin_transaction`, `abort_transaction` and `commit_transaction` statements are provided for transactions that begin in one method and may end in another according to circumstances determined at run-time.

References

[1] Gail E. Kaiser and David Garlan.
Melding Software Systems from Reusable Building Blocks.
IEEE Software :17-24, July, 1987.

[2] Gail E. Kaiser and David Garlan.
MELDing Data Flow and Object-Oriented Programming.
In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 254-267. Orlando FL, October, 1987.
Special issue of *SIGPlan Notices*, 22(12), December 1987.