Concurrent Clustered Programming*

Vijay Saraswat** and Radha Jagadeesan***

No Institute Given

1 Introduction

A holy grail of concurrency and theoretical programming languages is the development of clean but real concurrent languages. Real enough that they can be used for regular programming tasks by millions of programmers. Clean enough that they can be formalized, theorems proven, correct compilers, transformation systems, program development methodologies, interactive refactoring tools developed.

On the theoretical front, there has always been considerable research in concurrency in programming languages – CCS, CSP, process algebras, CCP etc. On the practical front, in imperative languages, CILK[1,2] has introduced some novel ideas (work-stealing for SMPs). Titanium [3], Co-Array Fortran [4] and UPC [5] have introduced the *partitioned global address space* model [6] in JAVA, Fortran and C respectively (albeit in an SPMD framework). However, the state of the art in concurrent high performance computing continues to be library based (e.g. OpenMP [7] and MPI [8]) rather than language based. Mainstream languages have been slow to adopt concurrency. JAVA^{TM} [9] has the best thought out model (some recent work has been proposed on a memory model for C++ [10]), but it suffers from several problems. A single global heap does not scale – complex memory models [11] are needed to enable efficient implementation on modern multi-processors. As is widely accepted, lock-based synchronization is very brittle – leading to underlocking/overlocking and bugs that are very hard to find. For HPCS computation, JAVA does not support multidimensional arrays, user-definable value types, relaxed exception model, aggregate operations etc [12,13].

A number of ideas have come together now which promise a breakthrough. Over the last decade, (1) JAVA-like languages [1] have established themselves as robust, reliable vehicles for millions of programmers. They provides a clean basis for dealing with sequential computing. (2) Eclipse [14] offer an elegant framework in which program manipulation tools, such as advanced refactoring tools [15] can expect to find widespread use. (3) The exciting new idea of *atomic blocks* [16,17,18] has raised the possibility that the promise of robust, reliable parallel imperative programming may be at hand.

* We thank Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Doug Lea, Maged Michael, Robert O'Callahan, Christoph von Praun, Vivek Sarkar, and Jan Vitek for many discussions on the topic of this paper.

** IBM T.J. Watson Research Lab. Research supported in part by DARPA No. NBCH30390004.

*** School of CTI, DePaul University. Research supported in part by NSF 0430175.

[1] In this category we include C#. Roughly this class of languages includes memory-safe, statically-typed, class-based, single-inheritance, imperative OO languages with a managed run-time.

A fundamental new opportunity presents itself with the development of the next generation of high performance computers (e.g. capable of $O(10^{15})$ operations per second). These will be based on scale-out techniques rather than clock rate increases (because of power and heat dissipation issues). This leads to a notion of *clustered computing*: a single computer may contain hundreds of thousands of tightly coupled (multi-threaded/SMP) nodes. Unlike a distributed model, failure of a single node is tantamount to failure of the entire machine (and all nodes may be assumed to lie in the same trust boundary). However because of latency and bandwidth, the notion of a single uniform shared memory is no longer appropriate for such machines.

Under the aegis of the DARPA HPCS programme, we and our colleagues have designed a concurrent programming language for clustered computing, X10 [19,20]. The fundamental goal of X10 is to enable scalable, high-performance, high-productivity programming for high-end computers – for traditional numerical computation workloads (such as weather simulation, molecular dynamics, particle transport problems etc) as well as commercial server workloads. We have designed X10 as a simple and clean, modern, statically-typed OO programming language. In place of JAVA's threads and locks, we have added a few carefully chosen orthogonal notions: *places* to express locality, atomic blocks for synchronization, asyncs for concurrency, clocks for repeated quiescence detection, `finish` for termination detection, and distributions for multi-dimensional, distributed arrays. These capture the central elements of shared memory access (e.g. OpenMP) and message passing (e.g. MPI [8]).

We are developing a complete formal operational semantics for X10, called FOUNDATIONAL X10 (FX10), [21], intended to capture the essence of X10. The semantics is intended to be used as a basis for informal reasoning with programs, program development methodologies, advanced compiler optimizations, and program refactoring.

The purpose of this paper is to present the core concurrency and distribution features of FX10. The primary contributions of this paper are as follows. (1) We present a simple programming model (within a JAVA-like setting) for clustered computing, organized around a few orthogonal constructs for managing space (places, distributions) and (logical) time (clocks, `finish`). (2) We show that programs in a rich subset, UFX10 – including `finish` and nested clocks – cannot deadlock. (3) We show that lock-free computations, CILK programs, systolic arrays and MPI computations can be expressed in this subset. (4) We formalize a compositional operational semantics based on bisimulation. (5) We establish other basic properties of the programming model: equational laws for various constructs; the correctness of programs is not affected by the number of places; clock quiescence is stable.

The model is formalized in the style of previous JAVA-centric calculi focusing on types (FJ [22]) and (sequential) imperative programming (MJ [23], Classic Java [24]), and may also be seen as a core concurrent object-oriented calculus, suitable for the study of concurrency in OO languages. In this initial report, we do not include details about the static semantics, the development of the (place-based) type system, and many aspects (such as static state, visibility, overloading etc) of the underlying object system. Instead we focus on the basic properties of the new control constructs we introduce.

2 Basic Programming model

The core sequential constructs for FX10 are similar to those for JAVA: the user defines types (classes and interfaces); classes are organized in a single inheritance hierarchy (but a class may implement multiple interfaces); an interface extends multiple interfaces; class definitions may be nested; exceptions are non-resumptive; the standard set of control constructs is available. Unlike JAVA, FX10 has an integrated type system with explicitly specified `nullable` types, and introduces a notion of *value* classes (all the fields of a class are `final`; thus instances of such classes may be freely copied).[2]

2.1 Places and activities

The central problem for a very large memory machine is the memory model: what guarantees should be made about how writes by one thread may be made visible to reads by other threads [25,26]. The simplest guarantee – sequentially consistent execution [27] – is too expensive on modern multiprocessors. It requires barrier syncs which may cost hundreds of cycles (as opposed to load/stores which cost one cycle). Unfortunately, attempts to provide a "weaker" semantics have proven very difficult to formalize and understand (cf the work on the Java memory model [9,11]).

Our approach to this conundrum is to introduce the notion of a *place*. A place consists of a collection of data and activities that operate on the data. (A computation may consist of millions of places.) Each activity has a sequentially consistent view of the data at that place and may operate only on the data at the place. It may reference data at other places, but must operate on them only by launching asynchronous activities (at the place where the data lives). Thus X10 supports a *globally asynchronous, locally synchronous* computation model.[3]

Any activity may use the place expression `here` to reference the current place. Places are assumed to be totally ordered; if `p` is a place expression, then `p.next` is a place expression denoting the next place in the order. There are no expressions for creating a new place, rather each computation is initiated with a fixed number of places.[4]

Based on the work on ZPL[28], X10 supports multi-dimensional arrays based on the notion of explicit *regions* (sets of multi-dimensional index points) and *distributions* (maps from regions to places, e.g. blocked distribution, cyclic distribution etc). An array is specified by an underlying base type and a distribution; its elements are scattered across places according to the distribution. For reasons of brevity we omit the formalization of arrays from this presentation. All the results of this paper continue to hold in the presence of such arrays.

---

[2] The current version of X10 does not support user-defined generic types; this is planned for future work.

[3] Unlike other PGAS languages such as Titanium, Co-Array Fortran and UPC, X10 is not based on an SPMD model. It permits *multiple* activities per place.

[4] This is consistent with most MPI programs that are started with a fixed number of processes.

2.2 Atomic blocks

X10 introduces statements of the form `when(c) s` where `s` is a statement ([17,29,30]). Such a statement blocks until (if ever) a state is reached in which `c` evaluates to `true`; in this state `s` is executed atomically – in a single step as if all other activities are frozen.

We use the shorthand `atomic s` for `when(true) s`, and `halt;` for `when(false) s`.[5] We permit the modifier `atomic` on method definitions and take that to mean that the body of the method is enclosed in an `atomic`.

The type system ensures that such a statement will dynamically access only local data.

What properties should such a construct be expected to have? Let us say that two statements `s1` and `s2` are to be considered *congruent*, written `s1` $\equiv$ `s2` if for any statement-context $C[.]$, the statements $C[$`s1`$]$ and $C[$`s2`$]$ are going to be operationally indistinguishable. Intuitively, any result obtained by the first can be obtained by the second and vice-versa. (In Section 4.1 we formalize this through contextual bisimulation.)

We expect the following laws to hold:

$$\texttt{when(c) when(d) s} \equiv \texttt{when(c\&\&d) s} \tag{1}$$

$$\texttt{if(c) when(c) s} \equiv \texttt{if(c) atomic\{s\}} \tag{2}$$

$$\texttt{atomic \{ s1 atomic s2\}} \equiv \texttt{atomic \{s1 s2\}} \tag{3}$$

For future reference, we define the language UFX10 to be the fragment of FX10 which permits `atomic` but not `when`. (Similarly UX10 is the corresponding fragment of X10.) The basic result of this paper is that programs in UFX10 are deadlock-free (Theorem 19).

EXAMPLE 1 (CAS). The following class implements a *compare and swap* (CAS) operation. In the code below `target` is defined in the lexically enclosing environment.

```
atomic boolean CAS(Object old, Object new){
   if (target.equals(old)){
     target = new;
     return true;
   }
   return false;
}
```

CAS is the basis for many highly concurrent, non-blocking (lock-free, wait-free) datastructures (e.g. [31,32]). These algorithms can be expressed directly in UFX10.

2.3 Asynchronous activities

An asynchronous activity is created by a statement `async(p)s` where `p` is a place expression and `s` is a statement. Such a statement is executed by spawning an activity

[5] Note that a "multi-armed" version `when(c1) s1 else (c2) s2 ... else (ck) sk` can be defined using `when` and a conditional statement.

at the place designated by `p` to execute statement `s`. An activity is created in a place and remains resident at that place for its lifetime. `s` (and `p`) may access lexically scoped variables.

Below, we use `async s` as shorthand for `async(here)s`, and `s1 || s2` as shorthand for `async s1 async s2`. We expect the following laws to hold:

$$\texttt{async}(P)\{s\};\texttt{async}(Q)\{s_1\} \equiv \texttt{async}(Q)\{s_1\};\texttt{async}(P)\{s\} \quad (4)$$

$$\texttt{async}(P)\{s_1\};\texttt{async}(P)\{s_2\} \equiv \texttt{async}(P)\{s_1||s_2\} \quad (5)$$

$$\texttt{async}(P)\{\texttt{async}(Q)\{s\}\} \equiv \texttt{async}(Q[\texttt{here}/P])\{s\} \quad (6)$$

$$\texttt{async}(P)\{\texttt{async}(Q)\{s\}s_1\} \equiv \texttt{async}(Q[\texttt{here}/P])\{s\};\texttt{async}(P)\{s_1\} \quad (7)$$

The first two laws reflect the intuition that an async call terminates instantaneously. The next two laws reflect the commutativity of nested asyncs, modulo a substitution for `here`.

EXAMPLE 2 (LATCH). A latch is an object which is initially *unlatched*, and may become latched. Once it is latched it stays latched. It may be implemented in FX10 thus:[6]

```
class Latch {
  boolean forced = false;
  nullable Object result = null;
  atomic
    boolean setValue( nullable Object val){
    if ( forced ) return false;
    this.result = val;
    this.z = z;
    this.forced = true;
    return true;
  }
  Object force() {
    when ( forced ) {
      return result;
    }
  }
}
```

2.4 `finish`

`finish` converts global termination to local termination. A statement `finish s` executes `s`; it terminates only when `s` terminates globally. `s` is said to terminate globally iff it terminates locally and all activities spawned dynamically during its execution terminate globally. An activity terminates globally if its statement terminates globally.

We expect `finish` to satisfy the following laws:

$$\texttt{finish}\,\{\texttt{s s1}\} \equiv \texttt{finish s finish s1} \quad (8)$$

$$\texttt{finish halt} \equiv \texttt{halt} \quad (9)$$

$$\texttt{finish when(c) s} \equiv \texttt{when(c) finish s} \quad (10)$$

$$\texttt{finish if(c)\{s\} else \{s1\}} \equiv \texttt{if (c)\{finish s\} else\{finish s1\}} \quad (11)$$

$$\texttt{finish x=e} \equiv \texttt{x=e} \quad (12)$$

$$\texttt{finish async(p) \{\}} \equiv \{\} \quad (13)$$

[6] In FX10, reference types do not contain `null` by default (unlike JAVA), instead the `nullable` type constructor must be used to construct a type with the value `null`. This is one of the sequential features of FX10 we do not discuss in this paper for lack of space.

EXAMPLE 3 (FUTURES). Consider a new expression of the form `future (p){e}` where `e` is of type `T`. (As with async, we will use the syntax `future{e}` for `future(here){e}`.) It is desired that this stand for a value of type `future<T>`. When this is `forced`, it will return a value of type `T` which is the result of evaluating the expression `e` in the place `p`. Such an expression may be implemented as a new latch `L`, with the following statement executed in parallel:

```
async(p){
  finish T X = e;
  async(L.place){
    L.setValue(X);
  }
}
```

This example shows how distributed datastructures may be created in FX10 (even without using distributed arrays); the field of an object may contain a reference to an object at a different place.

2.5 Clocks

In concurrent programming, it is often necessary for an activity to repeatedly detect quiescence of arbitrary programmer-specified, data-dependent set of activities. For instance, in a molecular dynamics application, it may be necessary for a controller activity to determine that (the activity associated with) each molecule has computed the force incident on it from all other molecules, and hence its instantaneous acceleration a. The controller may then advance simulation time, causing each molecule to determine its new position p and velocity v (as a function of its mass m, a and old p and v).

Such coordination is typically done through *barriers*. X10 *clocks* are designed to offer the functionality of multiple barriers in a dynamic context while still supporting determinate, deadlock-free parallel computation.

At any given time an activity is *registered* with zero or more clocks (the set of clocks an activity is registered with is called its *clock set*). The only way in which an activity can be registered with a clock is if it is created with that clock or it creates the clock; there is no explicit registration statement. An activity creates a new clock `x` by executing the statement `clock x is new;`. An activity created with the statement `async(p) clocked(c1,...,cn) s` is registered with the clocks `c1, ..., cn` when it is created.

In any configuration of the system a clock is in a given phase. It may advance to the next phase only when it has *quiesced*. A clock quiesces only when all its registered activities have quiesced. Once a clock has quiesced it may autonomously decide to advance to its next phase. An activity quiesces on a particular clock `c` by executing the statement `resume c;`. This is a non-blocking operation; the activity may continue to execute other statements. (Thus X10 supports *split-phase* clocks.) An activity may resume a clock more than once in the current phase; this is not considered an error.

An activity may suspend until such time as all clocks in its clock set have advanced to the next phase by executing `next;`. This automatically quiesces the activity on all clocks in its clock set. X10 does not have a statement of the form `next c;` which

would allow an activity to choose the clock that it wishes to suspend on; such a statement is error-prone and can easily cause deadlock.

An activity may deregister itself with a clock `c` by executing `drop c;`. Subsequently the clock and the activity are oblivious of each other. Note that once an activity is deregistered from `c`, it can never be re-registered with `c`; X10 has no `register c;` statement.

Static conditions. Two static rules apply to clocks.[7] (1) An activity may transmit only those clocks that it is registered with and has not quiesced on. (2) The statement `s` in a `finish s` may not transmit *any* clocks to a new activity. (The new activity may create clocks and pass them on to other activities it creates.)

Property (1) ensure that clock quiescence in the current phase is a *stable* property [33]: once it holds it will continue to hold. Example 4 in the Appendix illustrates the necessity of Property (1). Property (2) is discussed in Section 4.3.

EXAMPLE 4 (TRANSMISSION REQUIRES REGISTRATION.). The program on the left is illegal; it violates Property (1). An activity registered only on the clock `c` attempts to pass the clock `d` to an async it is spawning. The phase in which Activity 2 will be registered with `d` is not determined; specifically `s2` may not execute in the same phase as `s4`. Clock quiescence is not stable: the main activity may execute `s4` and then `next`. Now `d` has quiesced because the only activity registered with it is the main activity (which has quiesced). However, Activity 2 may now be spawned causing `d` to become not quiescent.

On the other hand the (legal) program on the RHS guarantees `s4` is executed in the current phase of `d`, as is `s2`. Activity 2 is spawned by an activity that is registered on `d` and has not quiesced on `d`; hence Activity 2 is spawned when `d` is not quiescent.

Note that `next; next;` is not the same as `next;`. The first will wait for clocks to advance twice, and the second once.

EXAMPLE 5 (WAREHOUSE SIMULATION,[34]). The schematic code for the heart of the SPECjbb2000 warehouse simulation is shown below.

The simulation involves a master activity that sequences stages (e.g. `RAMP_UP`)) and `w*t` slave clients. They synchronize through a single clock, `c`. The master communicates the current mode to the slaves by atomically updating the shared variable `mode`.

The first synchronization point ensures that all clients have been initialized. Once all clients are initialized, they enter a think-act loop, recording the data associated with the transaction when the system is in `RECORDING` mode (which means it is not in `RAMP_UP` or `RAMP_DOWN` mode. In `RAMP_DOWN` mode each client continues to run transactions (albeit silently); hence the use of `resume;` and not `next;`. Once the master detects all clients are in `RAMP_DOWN`, it moves to `STOP`, causing all clients to exit their loop, process the data for their run and terminate.

[7] The actual formalization of the static conditions is outside the scope of this paper. Please see the forthcoming [21] for details.

```
clock c is new;
clock d is new;
async clocked(c) {
   async clocked(d){
     //Activity 2
     s2
     next;
   }
   s3
   next;
}
s4
next; // Race
}
```

Disallowed

```
clock c is new;
clock d is new;
async clocked(c, d) {
   async clocked(d) {
     //Activity 2
     s2
     next;
   }
   drop d;
   s3
   next;
}
s4
next;
}
```

OK

Table 1. Transmission requires registration

Once the master and the client activities have `finish`'ed, the main activity can print results. See Table 2.

Clocks may be used to obtain oversampling through nesting.

EXAMPLE 6 (OVERSAMPLING: HIERARCHICALLY NESTED CLOCKS). Suppose we desire to clock two activities on a clock `c`, and require that the first spawn several activities which must execute several steps in synchrony *within* the current phase of `c`. This may be expressed in FX10 using multiple clocks:

```
clock c is new;
async clocked(c) {s}
async clocked(c) {
   finish async {
     clock d is new;
     async clocked(d){s1}
     async clocked(d){s2}
   }
   next;
   s3
}
```

EXAMPLE 7 (NOW). The user may require that execution of a statement `s` terminate completely in the current phase of the clock `c`. This can be written:

```
async clocked(c){
    finish async s;
}
```

The outer activity is registered on `c`; hence `c` cannot advance until it performs a `next` or terminates. It cannot terminate until the `finish` is completed. An async is used to ensure that the execution of `s` is done in an activity which is not registered with any old clock. Thus any `next` performed by `s` will interact only with "new" clocks (produced during the execution of `s`).

Below, let us call such a statement `now(c) s`.

```
finish async {                          // master activity
  clock c is new;                       next; //1.
  Company company = createCompany(...); company.mode = RAMP_UP;
  for (int w : 0:wh_num)                sleep rampuptime;
    for (int t: 0:term_num)             company.mode = RECORDING;
      async clocked(c) { // a client    sleep recordingtime;
        initialize;                     company.mode = RAMP_DOWN;
        next; //1.                      next; //2.
        while (company.mode!=STOP){     // All clients in RAMP_DOWN
          select a transaction;         company.mode = STOP;
          think;                      } // finish
          process the transaction;    // Simulation completed.
          if (company.mode==RECORDING)  print results.
            record data;
          if (company.mode==RAMP_DOWN){
            resume c; //2.
          }
        }
        // company.mode == STOP
        gather global data;
      } // a client
```

Table 2. Schematic Code for SpecJBB

EXAMPLE 8 (SYSTOLIC ARRAYS). The time-space patterns of read-write accesses of systolic algorithms can be represented in UFX10. For example, consider the core of the ASCI Benchmark Sweep3D program for computing solutions to mass transport problems (Table 3). The core computation is a triply nested sequential loop over a 2*d* array. In the loop a value of a variable in the current iteration is dependent on the values of variables to the west and north in the previous iteration. Such a problem can be parallelized through pipelining. One visualizes a diagonal wavefront sweeping through the array from the top left.

An X10 version of the program may be described as follows. (For lack of space, we present a single-place version of this program.) There is a 2*d* array `W` supported by two 2*d* arrays of clocks of the same dimension `W` and `N`. The clock `W[i,j]` (`N[i,j]`) is used to synchronize the read of the value `A[i,j]` by the activity to the east (south) of `(i,j)`. As soon as an activity reads a value it resumes the appropriate clock.

Consider the west and north boundary elements. For simplicity we will assume that they compute some function independent of their neighbors. The value of `A[i,j]` is written by an activity registered with the clocks `W[i,j]` and `N[i,j]`. The activity merely computes and writes the value, and then performs a `next;` (to wait for the value to be read from the east and the south) and then repeats. A two-place cyclic buffer is maintained to ensure that writes are performed to the location which is not being read in the current phase.

Consider the inner elements of `A` (i.e. those not on the west or north fringe). The value of `A[i,j]` is written by an activity registered with the clocks `W[i-1,j]`, `N[i,j-1]`, `W[i,j]`,`N[i,j]`. As soon as it has read the value it resumes the ap-

This example uses array notation from X10 that is not part of the FX10 fragment presented in this paper.

```
finish async {
  region R = [0:N, 0:N];
  region West = [0:1, 0:N];
  region North = [0:N, N-1:N];
  double[R][0:1] A = ...;
  clock W[R] is new;
  clock N[R] is new;
  for(point[i,j]: West || North)){
     async clocked(W[i,j],N[i,j]){
        int k = 0;
        for (int t : 1:TMax){
          k = (k+1) mod 2;
          A[i,j][k] = ...;
          next;
        }
     }
  }
```

```
for(point [i,j]: (R-(West||North)))
   async clocked (W[i-1,j],N[i,j-1],
                  W[i,j],N[i,j]) {
      int k = 0;
      for (int t : 1:TMax){
        double west, north;
        finish {
          finish async west = A[i-1,j][k];
          resume W[i-1,j];
          finish async north = A[i,j-1][k];
          resume N[i,j-1];
        }
        k = (k+1) mod 2;
        now (W[i,j]) A[i,j][k]
            = compute(west, north);
        next;
      }
   }
}
```

We use `async` to read the value (to the west and north) in an `async` because in general the value could be located in a different place, and an `async` would be necessary.

Table 3. A pipelined computation for Sweep3D

propriate clock; as soon as it has written a value it performs a `next;` (which resumes the clocks `W[i,j],N[i,j]`).

This establishes the invariant that each activity will proceed to the next iteration only when values are ready for it to read on the north and the west, and the value that it has produced has been read by the cell on the south and east.

EXAMPLE 9 (JACOBI). Consider a "red/black" 1-dimensional Jacobi iteration. Given a 1-d int array `red`, it is desired to repeatedly average neighbours until the absolute value of the difference between the current value and the value in the previous iteration, across the entire array, is below a threshold, `epsilon`.

The code is shown below. The main activity creates a new clock `c` and spawns a slave activity for each element of the array, clocked on `c`. This activity will alternately average the neighbors of the `red` array into the `black` array, and vice versa. All activities must use the clock to synchronize after they have performed their update. The main activity uses a parallel reduction operation on the delta array to determine whether the threshold has been reached, using the clock to determine when all activities have performed their work in the current phase.[8]

[8] The actual code in X10 can be made much more efficient than this, spawning an activity for each block of *k* elements, and block-distributing the array across all places. We elide the details for lack of space.

```
clock c = new clock();
int iters = 0;
final boolean done = new boolean(false);
for (int i : 0: N) async clocked(c) {
  do {
    // red
    black[i] = (red[i-1]+ red[i+1])/2;
    dBlack[i] =
       abs(black[i]-red[i]);
    next; // black
    if (done.isTrue()) {
      break;
    }
    red[i] = (black[i-1]+black[i+1])/2;
    dRed[i] = abs(black[i]-red[i]);
    next; // red
    if (done.isTrue()) {
      break;
    }
  } while (true);
}
```

```
do {
  next; // black
  if (dBlack.reduce(Max)
       <= epsilon) {
     done.setTrue();
     break;
  }
  next; // red
  if (dRed.reduce(Max)
       <= epsilon) {
     done.setTrue();
     break;
  }
} while (true);
```

2.6 CILK programs as X10 programs

CILK [1] introduces the idea that in a purely sequential language such as `C` method calls may be marked with `spawn` to indicate that they should be executed asynchronously. The `sync;` statement may be used to require that the method invocation be delayed until all spawn'ed activities have terminated. An efficient work-stealing scheduler is presented for implementing CILK on SMPs [2].

UFX10 may be thought of as CILK developed on top of JAVA, extended with multiple places, distributed arrays, and clocks, and atomic blocks, with `spawn` simulated by `async` and `sync` by `finish`.[9] FX10 permits the possibility that an `async` may continue to execute even when the spawning method has terminated. We believe the work-stealing scheduler can be adapted to this more general setting.

2.7 OpenMP/Java programs as X10 programs

JAVA programs with `synchronize` (but without `wait/notify`) can be translated into UX10: every `synchronize(o){s}` statement is translated to `atomic{s}`.

Many JAVA benchmarks can be written directly in UX10. For instance SPECjbb2000 [34] uses wait/notifyall to implement barrier synchronization; these can be translated into the use of a single clock as discussed above. All the Java Grande Forum benchmarks [35] that use threads (crypt, lufact, moldyn, montecarlo, raytracer, series, sor)

[9] One minor difference is that in FX10 when an `async` is spawned it *copies* (the relevant portions of) the variable stack of the parent activity, rather than sharing it. The spawned `async` may only access `final` variables in the lexically enclosing environment, like JAVA. CILK shares the variable stack. CILK behavior can be emulated in FX10 by "boxing" the values in the variable stack.

have been ported to UX10. All the SpecJVM98 benchmarks that use Threads can be ported to UX10.

2.8 MPI programs as X10 programs

An MPI program may be represented in X10 with a place per MPI process, running a single main activity. Two-way blocking send/receive may be implemented using bounded buffers that are easy to program in X10. The MPI-2 one-way communication primitives can be directly implemented with `asyncs`. X10 distributed arrays, clocks and collective operations permit a much more direct representation of the basic idioms underlying MPI communicators and collectives. We omit details for lack of space.

3 Formalization of the model

(Program)	p	$::=$	$cd_1 \ldots cd_n; \bar{s}$	
(Class)	cd	$::=$	`class` C `extends` C	
			$\{$ $\mathrm{fd}_1 \ldots \mathrm{fd}_k$ cnd $\mathrm{md}_1 \ldots \mathrm{md}_k$ $\}$	
(Field)	fd	$::=$	$T\ f;$	
(Constructor)	cnd	$::=$	$C(T_1\ x_1, \ldots T_j\ x_j)$	
			$\{$`super`$(e_1, \ldots, e_k); s_1 \ldots s_n$ $\}$	
(Method)	md	$::=$	$\tau\ m(T_1\ x_1, \ldots T_j\ x_j)$	
			$\{s_1 \ldots s_k\}$	
(Return type)	τ	$::=$	T | `void`	
(Type)	T	$::==$	C	
(Expression)	e	$::=$	x	*Variable*
			`null`	*Null*
			$e.f$	*Field access*
			(C) e	*Cast*
			pe	*Promotable expression*
(Promotable exp)	pe	$::=$	$e.m(e_1, \ldots, e_k)$	*Method invocation*
			`new` $C(e_1, \ldots, e_k)$	*Object creation*
(Statement)	s	$::=$	*;*	*No-op*
			pe	*Promoted expression*
			`if` *(e==e)* $\{s_1 \ldots s_k\}$	
			`else` $\{s_{k+1} \ldots s_n\}$	*Conditional*
			e.f = e;	*Field assignment*
			T x;	*Local variable declaration*
			x = e;	*Variable assignment*
			`return` *e;*	*Return*
			$\{s_1, \ldots, s_n\}$	*Block*

The core MJ calculus does not represent packages, import statements, interfaces, arrays, built-in types, method overloading, static state, try/catch/throws, loops, multi-threading.

Table 4. Syntax for MJ

Our presentation is built on top of the MJ calculus [23]. The core MJ calculus, described in the appendix, includes mutable state, block structured values and basic object-oriented features. It does not however represent packages, import statements, interfaces, arrays, built-in types, method overloading, static state, try/catch/throws, loops, multi-threading. The syntax of MJ is described in Table 4.

Figure 5 describes the features we add to core MJ.

(Place Expression)		(FX10 Configuration)	FX10config	::=	(H,σ,Δ)
pe ::= `here` | `pe.next` | `v.place`		(Open Activity)	a	::=	$p : (\mathrm{VS}, \mathrm{CF}, \mathrm{FS})$
(Statement)		(Closed Activity)	ca	::=	$\mathrm{FS}' : s$
s ::= `when(c) s`		(Terminated Activity)	ta	::=	$p : (\mathrm{VS}, ;, [])$
`async(p)clocked(`$\bar{c}$`)s`		(Places)	p	::=	*int*
`finish` s		(Closed Frame)	CF	::=	s | `wait;`
`next;`		(Open Frame)	OF	::=	`async(`$\bullet$`)` s
`clock x is new`					| `when(`$\bullet$`)` s
`resume` c		(Values)	v	::=	`null` | o | p
`drop` c					

Table 5. Syntax and Configurations for FX10

The transition relation relates configurations. In a configuration (H,σ,Δ), H is a shared heap and σ is a shared constraint stores whose role will become clear in the description of clocks. In a basic configuration, Δ is an activity that includes a place designator, a frame (representing the continuation) and a variable stack (representing block-structured program variables). In a composite FX10 configurations, Δ is a tree each of whose nodes n is labeled with an *open activity* or a *closed activity*.

3.1 Transition system

The transition relation for the sequential fragment of FX10 is adapted from MJ and is described in the appendix. In the main text, we focus on concurrency and location.

Place sensitive MJ transitions. The variable "here" evaluates to the current place.

$$\frac{}{(H,\sigma,p:(VS,\texttt{here},FS)) \longrightarrow (H,\sigma,p:(VS,p,FS))}\ (\textsc{Here})$$

The heap has place information for each object: this is recoverable using `o.place`. Field access is permitted only on objects at the same place. Access to objects located at

a different place leads to PlaceError.

(NEW)
$$\frac{cnBody(C) = (\bar{x},\bar{s}), \Delta_c(C) = \bar{C}, o \notin dom(H), \mathcal{F} = [f \mapsto null, f \in fields(C)], BS = [this \mapsto (o,C), \bar{x} \mapsto (\bar{v},\bar{C})]}{(H,\sigma,p:(VS,\texttt{new}\ C(\bar{v}),FS)) \longrightarrow (H[o \mapsto p:(C,\mathcal{F}),\sigma,p:((BS\circ[])\circ VS,\bar{s},(return\ o;)\circ FS))}$$

(FIELDACCESS)
$$\frac{}{(H,\sigma,p:(VS,o.f,FS)) \longrightarrow (H,\sigma,p:(VS,v,FS))}\ H(o) = p:((C,\mathcal{F})), \mathcal{F}(f) = v,$$

(FIELDACCESSPLACE)
$$\frac{}{(H,\sigma,p:(VS,v.\texttt{place},FS)) \longrightarrow (H,\sigma,p:(VS,p,FS))}\ H(o) = p:((C,\mathcal{F}))$$

(FIELDACCESS)
$$\frac{}{(H,\sigma,p:(VS,o.f,FS)) \longrightarrow PlaceError}\ H(o) = p':((C,\mathcal{F})), p \neq p'$$

Tree transitions. The transition relation on composite configurations is described as a tree transformation. Let $\bar{\Delta}$ be the (possibly empty) sequence $\Delta_0,\ldots,\Delta_{n-1}$. We use the syntax $n \rhd \bar{\Delta}$ to indicate a tree with root node n and subtrees $\Delta_0,\ldots,\Delta_{n-1}$.

A rule $\Delta[\Delta_1] \longrightarrow \Delta[\Delta_2]$ is understood as saying that a tree Δ containing a subtree Δ_1 can transition to a tree which is the same as Δ except that the subtree Δ_1 is replaced by Δ_2. Thus if Δ is the tree $1(2(3,4),5(6))$ then an application of the rule $\Delta[2] \longrightarrow \Delta[8(9)]$ gives the tree $1(8(9,3,4),5(6))$. An application of the rule $\Delta[2 \rhd \Delta'] \longrightarrow \Delta[8(9)]$ gives the tree $1(8(9),5(6))$ (the entire subtree at 2 is replaced).

(COMPOSITE)
$$\frac{(H,\sigma,\Delta_1) \longrightarrow (H',\sigma',\Delta_2)}{(H,\sigma,\Delta[\Delta_1]) \longrightarrow (H',\sigma',\Delta[\Delta_2])}$$

Atomic blocks $\texttt{when}(c)\ s$ completes in one step if and when c evaluates to true in the current store and without interruption s completes execution.

(ATOMIC)
$$\frac{(H,\sigma,p:(VS,e,[])) \xrightarrow{\star} (H_1,\sigma_1,p:(VS_1,true,[])),\ \ (H_1,\sigma_1,p:(VS,s,[])) \xrightarrow{\star} (H_2,\sigma_2,p:(VS_2,;,[]))}{(H,\sigma,p:(VS,\texttt{when}(e)\ s,FS)) \longrightarrow (H_2,\sigma_2,p:(VS_2,;,FS))}$$

Async without clocks In $\texttt{async}(e)\ s$, the expression e must be evaluated first. It is considered locally terminated after it has spawned the new activity. The spawned activity is started with an empty continuation, but is given the variable stack of the spawning environment.

(ASYNC1)
$$\frac{}{(H,\sigma,p:(VS,\texttt{async}(e)\ s,FS)) \longrightarrow (H,\sigma,p:(VS,e,\texttt{async}(\bullet)\ s\circ FS))}$$

(ASYNC2)
$$\frac{}{(H,\sigma,p:(VS,\texttt{async}(p')\ s,FS)) \longrightarrow (H,\sigma,p:(VS,;,FS) \rhd p':(VS,s,[]))}$$

Finish The finish rule creates a closed activity.

(FINISH1)
$$\frac{}{(H,\sigma,p:(VS,\texttt{finish}(s),FS)) \longrightarrow (H,\sigma,FS:p:(VS,s,[]))}$$

In the rule below, the entire subtree is replaced with a single node.

(FINISH2)
$$\frac{\Delta \text{ is a tree of terminated activities}}{(H,\sigma,FS:p:(VS,;,[])\rhd\Delta) \longrightarrow (H,\sigma,p:(VS,;,FS))}$$

Clocks To specify the semantics of clocks, we use the *streamed short circuit* technique for detecting stable properties of distributed systems from concurrent logic programming [33,36]. This technique makes the proof of the Clock Quiescence Stability theorem (Theorem 10) immediate. We note that this technique is used purely to specify the *semantics* of clocks. Clocks may be implemented using semaphores with counters.

We identify a certain class of objects as *promises*. These represent logical terms. The *constraint store* σ is a set of equality constraints on promises, equipped with a function **var** which represents the set of variables over which the constraints are defined. If X is does not occur in σ, then we write $\sigma+X$ to indicate a constraint store identical to σ except that $\mathbf{var}(\sigma+X) = \mathbf{var}(\sigma)\cup\{X\}$. One may think of $\sigma+X$ as $\sigma\cup\{X=X\}$.

$$\begin{array}{lll} \textit{(Promise)} & p ::= & X \\ & & p.\texttt{rest} \\ & & p.\texttt{first} \\ \textit{(Constraint Store)}\sigma & ::= & \varepsilon \mid p=p \mid \sigma,\sigma \end{array}$$

A *clock* c is a quadruple of promises $c = \langle h,t,myH,myT\rangle$. The pair (h,t) is called the *root switchstream* and the pair (myH,myT) the *local switchstream*. Each activity will indicate quiescence in the current phase of a clock by asserting $X=Y$ for the current switch (X,Y) in the local switchstream of the clock. It will detect quiescence of all activities in the current phase on a clock by checking that the current switch (X,Y) in the *root* switchstream is closed. A clock is passed to a newly created activity by *splitting* (see below). We augment the state of each activity with a *clock map* (a finite partial function from variables to clocks); the range of the clock map represents the set of clocks the activity is registered with.

(NEW CLOCK)
$$\frac{}{(H,\sigma,p:(VS,\texttt{clock x is new;},FS,K)) \longrightarrow (H,\sigma+h+t,p:(VS,;,FS,K'))}$$

where $x \notin dom(K)$, $h,t \notin \mathbf{var}\sigma$, and $K' = K[x\mapsto\langle h,t,h,t\rangle]$. (We assume alpha renaming to ensure that x is new.) Note that for a newly created clock the root switchstream is the same as the local switchstream. This reflects the fact that when a clock is created there is a single activity registered with the clock.

Clocks may be transmitted to new activities when they are created. The earlier `Async2` rule is altered as below to accommodate clocks.

(CLOCK-ASYNC)
$$\frac{}{(H,\sigma,p:(VS,\texttt{async}(p')\texttt{clocked}(\bar{c})\ s,FS,K)) \longrightarrow (H,\sigma',p:(VS,;,FS,K_s)\rhd p':(VS,s,[],K'))}$$

Let $\bar{c} = c_1, \ldots, c_n$. Here $\sigma' = \sigma + M_1 + \ldots + M_n$ for n new variables $M_1, \ldots, M_n$. Assume that K maps each c_i to (h_i, t_i, myH_i, myT_i). K' is a clock map over $\{c_1, \ldots, c_n\}$ mapping each c_i to (h_i, t_i, M_i, myT_i) and K_s is the same clock map as K except that each c_i is mapped to (h_i, t_i, myH_i, M_i). Thus the rule ensures that the local switchstream of each clock c_i is "split" into two switchstreams, and the old and new activities are each given one of the split switchstreams.

(RESUME)
$$\frac{}{(H, \sigma, p : (VS, \texttt{resume c;}, FS, K)) \longrightarrow (H, \sigma \cup c_f, p : (VS, \texttt{wait;}, FS, K))}$$

Here c_f is the constraint $X.\texttt{first} = Y.\texttt{first}$ where (X, Y) is the local switchstream of the clock c according to K.

(NEXT)
$$\frac{}{(H, \sigma, p : (VS, \texttt{next;}, FS, K)) \longrightarrow (H, \sigma \cup K_f, p : (VS, \texttt{wait;}, FS, K_r))}$$

Here K_f is the union of the constraints $X.\texttt{first} = Y.\texttt{first}$ where (X, Y) is the local switchstream of a clock in K. K_r is K with the local switchstream (X, Y) of each clock (in its range) replaced by $(X.\texttt{rest}, Y.\texttt{rest})$.

(WAIT)
$$\frac{\sigma \vdash K_f}{(H, \sigma, p : (VS, \texttt{wait;}, FS, K)) \longrightarrow (H, \sigma, p : (VS, ;, FS, K_t))}$$

Here K_f is the union of the constraints $X.\texttt{first} = Y.\texttt{first}$ where (X, Y) is the *root* switchstream of a clock in K. K_t is K with the root switchstream (X, Y) of each clock replaced by $(X.\texttt{rest}, Y.\texttt{rest})$. The entailment relation $\vdash$ of the constraint system ensures transitivity, symmetry and reflexivity of equality.

(DROP)
$$\frac{}{(H, \sigma, p : (VS, \texttt{drop c;}, FS, K)) \longrightarrow (H, \sigma', p : (VS, ;, FS, K'))}$$

(TERMINATE)
$$\frac{}{(H, \sigma, p : (VS, ;, [], K)) \longrightarrow (H, \sigma', p : (VS, ;, [], \{\}))}$$

In the first rule, deregistration is modeled by equating the local switchstream:σ' is σ with the constraint $X = Y$ where (X, Y) is the local switchstream for $\texttt{c}$ in K. The new clock set K' is K less c. The second rule ensures that all terminated activities deregister themselves from all clocks. In this rule, σ' is σ with the constraint $X = Y$ for each local switchstream (X, Y) of a clock in (the range of) K.

THEOREM 10 (CLOCK QUIESCENCE IS STABLE). *Let configuration (H, σ, Δ) be such that $\sigma \vdash \texttt{X.first} = \texttt{Y.first}$ where (X, Y) is the root switchstream of a clock in the clock set of some activity in Δ. Let $(H, \sigma, \Delta) \longrightarrow (H', \sigma', \Delta')$. Then $\sigma' \vdash \texttt{X.first} = \texttt{Y.first}$.*

The only operations performed on the constraint store are Ask and Tell operations [36]. So, the theorem follows from the monotonicity of the constraint store.

4 Properties of FX10 programs

4.1 Bisimulation

We define a notion of bisimulation and show that it is a congruence. Our study of bisimulation focuses on issues relating to concurrency and shared memory. In particular, our treatment does not validate enough equations in the sequential subset, eg. those relating to garbage collection. Thus, fewer processes are deemed equal in our description than could be in a more detailed treatment. However, even this weak notion of equality suffices to prove several basic laws relating the new control constructs that we have discussed in this paper.

The transition system defined so far is for closed programs. In order to get a notion of equality that is a congruence wrt shared memory concurrent programming , we need to model the transition relation for open programs. We use a notion of an *environment move* to model update of shared heap by a concurrent activity. For a heap H, an environment move $\lambda = (o, f, p, o')$ is the update of the field f in object o (if it exists) to o'. Formally, if $H(o) = (C, \mathcal{F}), f \in dom(\mathcal{F})$ then, the resulting heap is $\lambda H = H[o \mapsto p : (C, F[f \mapsto o'])]$. This notion of environment move is stronger than necessary, e.g. it does not respect the visibility constraints imposed by the underlying OO paradigm.

We now define a bisimulation notion on configurations.

DEFINITION 11. A binary relation $\equiv$ on configurations is a bisimulation if the following holds. If $(H_1, \sigma_1, \Delta_1) \equiv (H_2, \sigma_2, \Delta_2)$, then:

- $H_1 = H_2$, $\sigma_1 = \sigma_2$.
- For all environment moves $\lambda = (o, f, p, o')$, if $(\lambda H_1, \sigma_1, \Delta_1) \longrightarrow (H_1', \sigma_1', \Delta_1')$, then there exists $(\lambda H_2, \sigma_2, \Delta_2) \longrightarrow (H_2', \sigma_2', \Delta_2')$ such that $(H_1', \sigma_1', \Delta_1') \equiv (H_2', \sigma_2', \Delta_2')$.
- For all environment moves $\lambda = (o, f, p, o')$, if $(\lambda H_2, \sigma_2, \Delta_2) \longrightarrow (H_2', \sigma_2', \Delta_2')$, then there exists $(\lambda H_1, \sigma_1, \Delta_1) \longrightarrow (H_1', \sigma_1', \Delta_1')$ such that $(H_2', \sigma_2', \Delta_2') \equiv (H_1', \sigma_1' \Delta_1')$.

Let $C[\cdot]$ be a open or closed activity context with a statement hole. Two statements s_1, s_2 are bisimilar, written $s_1 \equiv s_2$ if for all $C[\cdot]$ forall heaps H and forall σ , $(H, \sigma, C[s_1]) \equiv (H, \sigma, C[s_2])$. Similarly two (promotable) expressions e_1, e_2 are bisimilar, written $e_1 \equiv e_2$ if for all $C[\cdot]$ with expression holes, forall heaps H and forall σ, $(H, \sigma, C[e_1]) \equiv (H, \sigma, C[e_2])$.

The definition of $s_1 \equiv s_2$ (and $e_1 \equiv e_2$) quantifies over all sequential contexts. The use of environment moves in definition 11 enables us to lift the congruence property to all contexts including tree contexts.

LEMMA 12. *Let $\Delta[\cdot]$ (resp. $\Delta'[\cdot]$) be a tree of open or closed activity contexts with a statement (resp. expression) hole. Then, for all heaps H and forall σ, if $s_1 \equiv s_2$, then: $(H, \sigma, \Delta[s_1]) \equiv (H, \sigma, \Delta[s_2])$ and $(H, \sigma, \Delta'[e_1]) \equiv (H, \sigma, \Delta'[e_2])$.*

The equations described in section 2 hold upto bisimulation.

THEOREM 13. *Equations 1– 13 are valid.*

4.2 Monotonicity of places

As an application of bisimulation, we show that FX10 programs are insensitive to the location of objects in the heap. For these programs, distribution may introduce efficiency but *does not affect correctness*. Let $S_{\texttt{Coord}} = \{\texttt{here}, \texttt{here.next}, \texttt{here.next.next}\ldots\}$. Let s be such that no transition sequence from $([],\sigma,p:([],s,[]))$ leads to `PlaceError`.

LEMMA 14. *Let Θ be an operator on the set $S_{\texttt{Coord}}$. Let* $\texttt{trans}(\Theta,s)$ *be the result of replacing every subexpression* `async(`p`)` *in s by* $\texttt{async}(\Theta(e))$. *Then:* $([],\sigma,p:([],s,[])) \equiv ([],\sigma,p:([],\texttt{trans}(\Theta,s),[]))$.

In particular, when Θ is the constant function, we get a class of programs can be debugged and developed in a one-place execution environment before being deployed in a multi-place execution environment for efficiency. This characterization is related to the known property of CILK programs that the sequential execution of a CILK program is *simulated* by the execution of a CILK program in a multiprocessor environment obtained by ignoring `spawn` and `sync` directives. [10]

4.3 Deadlock freedom

For any configuration, define a *wait-for* graph as follows. There is a node for each clock and each activity that is suspended on a `next;` or a `finish`. There is an edge from each clock to an activity registered on that clock that is suspended on a `finish`. There is an edge from each activity suspended on a next to a clock the activity is registered on. There is an edge from each activity suspended on a `finishs` to each activity spawned by `s` that is suspended.

PROPOSITION 15. *A configuration is stuck iff it is terminal or there is a cycle in the wait-for graph.*

Clocks (without finish) are deadlock free, since in this case, no activity has an incoming edge. Hence there are no cycles in the wait-for graph. Deadlock-freedom holds for a larger language that encompasses lock-free computations, CILK programs, systolic arrays and MPI computations.

Proof of Deadlock Freedom Stratify each activity with a level number. The level of the initial activity is 0. The level of each activity spawned by an activity at level l is $l+1$. Label each clock with the level number of its creating activity.

OBSERVATION 16. *In any configuration, if there is a wait-for edge from an activity $X(m)$ (i.e. activity X at level m) suspended on a finish to an activity $Y(n)$ then $m < n$.*

This implies that any cycle in the waits for graph must involve at least one clock. The following is the critical lemma.

[10] Note that this property does not hold for FX10 because of clocks.

LEMMA 17. *In any configuration, if there is a wait-for edge from an activity $X(m)$ suspended on a finish to an activity $Y(n)$, then the only clocks reachable from $Y(n)$ have level $> m$.*

OBSERVATION 18. *If any configuration, if there is a wait-for edge from an activity $X(m)$ to a clock $C(n)$ then $n \leq m$.*

Note that by definition a configuration is deadlocked only if every activity in it is suspended on either a finish or a next.

THEOREM 19. *There are no cycles in the wait-for graph.*

In proof, observe that an activity $Y(n)$ is either suspended on a `next;` or a `finish`.

If it is suspended on a `next;`, then it has outgoing edges only to clocks. Its only incoming edge can be from an activity $X(m)$ suspended on a `finish`. Now, $m < n$. Consider a clock $C(k)$ with an edge into $X(m)$. Then $k \leq m$. By Lemma 17, this clock is not reachable from $Y(n)$.

If $Y(n)$ is suspended on a `finish`, consider any activity $X(m)$ reachable from it which is suspended on a next. (If there are none, the configuration is not deadlocked, because there is an activity which is not blocked on finish or next.) This is the same as the previous case, with Y and X reversed.

References

1. : CILK-5.3 reference manual. Technical report, Supercomputing Technologies Group (2000)
2. Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. In: Proceedings of the 35th Annual Symposium on the Foundations of Computer Science. (1994) 356–368
3. Yelick, K.A., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P.N., Graham, S.L., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performance java dialect. Concurrency - Practice and Experience **10** (1998) 825–836
4. Numrich, R., Reid, J.: Co-array Fortran for parallel programming. Fortran Forum **17** (1998)
5. El-Ghazawi, T., Carlson, W., Draper, J.: Upc language specification v1.1.1. Technical report, George Washington University (2003)
6. Carlson, W., El-Ghazawi, T., Numrich, B., Yelick, K.: Programming in the Partitioned Global Address Space Model (2003) Presentation at SC 2003, http://www.gwu.edu/ upc/tutorials.html.
7. : (Openmp specifications) www.openmp.org/specs.
8. Skjellum, A., Lusk, E., Gropp, W.: Using MPI: Portable Parallel Programming with the Message Passing Iinterface. MIT Press (1999)
9. Gosling, J., Joy, W., Steele, G., Bracha, G.: The Java Language Specification. Addison Wesley (2000)
10. Alexandrescu, A., Boehn, H., Henney, K., Lea, D., Pugh, B.: Memory model for multi-threaded c++. Technical report, metalanguage.com (2004) JTC1/SC22/WG21 – C++, Document Number: WG21/N1680=J16/04-0120.
11. Pugh, W.: Java Memory Model and Thread Specification Revision (2004) JSR 133, http://www.jcp.org/en/jsr/detail?id=133.

12. Moreira, J.E., Midkiff, S.P., Gupta, M., Artigas, P.V., Snir, M., Lawrence, R.D.: Java programming for high-performance numerical computing. IBM Systems Journal **39** (2000) 21–
13. Moreira, J., Midkiff, S., Gupta, M.: A comparison of three approaches to language, compiler, and library support for multidimensional arrays in java computing. In: Proceedings of the ACM Java Grande - ISCOPE 2001 Conference. (2001)
14. : (The eclipse project) www.eclipse.org.
15. Fuhrer, R., Tip, F., Kiezun, A.: Advanced refactorings in eclipse. In: OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM Press (2004) 8–8
16. Flanagan, C., Freund, S.: Atomizer: A dynamically atomicity checker for multithreaded programs. In: Conference Record of POPL 04: The 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Venice, Italy, New York, NY (2004)
17. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA. (2003) 388–403
18. Harris, T., Herlihy, M., Marlow, S., Jones, S.P.: Composable memory transaction. In: SIGPLAN Symposium on Principles and Practice of Parallel Programming. (2005)
19. Charles, P., Grothoff, C., Donawa, C., Ebcioglu, K., Kielstra, A., von Praun, C., Saraswat, V., Sarkar, V.: X10: An object-oriented approach to non-uniform cluster computing. Technical report, IBM Research (2005)
20. Saraswat, V.: Report on the Experimental Language X10, v0.41. Technical report, IBM Research (2005)
21. Saraswat, V., Jagadeesan, R.: Foundations of X10. Technical report, IBM Research (forthcoming)
22. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: a minimal core calculus for java and gj. ACM Trans. Program. Lang. Syst. **23** (2001) 396–450
23. G.M. Bierman, M.P., Pitts, A.: Mj: An imperative core calculus for java and java with effects. Technical Report 563, University of Cambridge Computer Laboratory (2003)
24. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, New York, NY (1998) 171–183
25. Adve, S., Pai, V.S., P.Ranganthan: Recent Advances in Memory Consistency Models for Hardware Shared-Memory Systems. Proceedings of the IEEE **87** (1999) 445–455
26. Adve, S., Gharachorloo, K.: Shared Memory Consistency Models: A tutorial. Technical report, Digital Western Research Laboratory (1995)
27. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers **28** (1979)
28. Chamberlain, B.L., Choi, S.E., Deitz, S.J., Snyder, L.: The high-level parallel language ZPL improves productivity and performance. In: Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing. (2004)
29. Hansen, P.B.: Structured multiprogramming. CACM **15** (1972)
30. Hoare, C.: Monitors: An operating system structuring concept. CACM **17** (1974) 549–557
31. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems **13** (1991) 124–149
32. Michael, M., Scott, M.: Simple, Fast and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In: Proceedings of the 15th ACM Annual Symposium on Principles of Distributed Computing. (1996) 267–275
33. Saraswat, V., Kahn, K., Shapiro, U., Weinbaum, D.: Detecting stable properties of networks in concurrent logic programming languages. In: Seventh Annual ACM Symposium on Principles of Distributed Computing. (1988) 210–222
34. : (Specjbb2000 (java business benchmark)) www.spec.org/jbb2000.

35. : (The java grande forum benchmark suite) www.epcc.ed.ac.uk/javagrande/javag.html.
36. Saraswat, V.: Concurrent Constraint Programming. Doctoral Dissertation Award and Logic Programming. MIT Press (1993)

A MJ semantics

(Configuration)	config	::=	*(*H, σ,*p:(* VS, CF, FS*))*
(Frame Stack)	FS	::=	F ∘ FS | []
(Frame)	F	::=	CF | OF
(Closed Frame)	CF	::=	$\bar{s}$ | `return` e; | {} | e | `super`$(\bar{e})$
(Open Frame)	OF	::=	`if`$(\bullet == e)\{\bar{s_1}\}$`else`$\{\bar{s_2}\}$;
			`if`$(v == \bullet)\{\bar{s_1}\}$`else`$\{\bar{s_2}\}$;
			$\bullet.f$ | $\bullet.f = e$; | $v.f = \bullet$; | $(C)\bullet$
			$v.m(v_1,\ldots,v_{i-1},\bullet,e_{i+1},\ldots,e_n)$
			`new`$C(v_1,\ldots,v_{i-1},\bullet,e_{i+1},\ldots,e_n)$
			`super`$(v_1,\ldots,v_{i-1},\bullet,e_{i+1},\ldots,e_n)$
			$x = \bullet$; | `return`$\bullet$; | $\bullet.m(\bar{e})$
(Values)	v	::=	`null` | o | p
(Variable Stack)	VS	::=	MS ∘ VS | []
(Method Scope)	MS	::=	BS ∘ MS | []
(Block Scope)	MS	::=	*finite pf from variables to pairs* (C, v)
(Heap)	H	::=	*finite pf from oids to heap objects*
(Heap Objects)	ho	::=	$(C, \mathcal{F})$
	F	::=	*finite pf from field names to values*

Table 6. Configurations for MJ

$eval(MS;x)$, evaluates a variable, x, in a method scope, MS. This partial function is defined only if the variable name is in the scope. $update(MS, x \mapsto v)$, that updates a method scope MS with the value v for the variable x, is also a partial function that is undefined if the variable is not in the scope.

Reduction1: MJ EVALUATION $(H,\sigma,p:(VS,CF,FS)) \longrightarrow (H',\sigma,p:(VS',CF',FS'))$

(EVAR-ACCESS)

$$\frac{}{(H,\sigma,p:(MS\circ VS,x,FS)) \longrightarrow (H,\sigma,p:(MS\circ VS,v,FS))}\ eval(MS,x)=(v,C)$$

(EVARWRITE)

$$\frac{}{(H,\sigma,p:(MS\circ VS,x=v;,FS)) \longrightarrow (H,\sigma,p:((update(MS,x\mapsto v))\circ VS,;,FS))}\ eval(MS,x)\downarrow$$

(EVARINTRO)

$$\frac{}{(H,\sigma,p:((BS\circ MS)\circ VS,Cx;,FS)) \longrightarrow (H,\sigma,p:((BS'\circ MS)\circ VS,;,FS))}\ BS'=BS[x\mapsto(null,C)], x\notin dom(BS\circ MS)$$

(BLOCKINTRO)

$$\frac{}{(H,\sigma,p:(MS\circ VS,\{\bar{s}\},FS)) \longrightarrow (H,\sigma,p:((\{\}\circ MS)\circ VS,\bar{s},(\{\})\circ FS))}$$

(BLOCKELIM)

$$\frac{}{(H,\sigma,p:((BS\circ MS)\circ VS,\{\},FS)) \longrightarrow (H,\sigma,p:(MS\circ VS,;,FS))}$$

(RETURN)

$$\frac{}{(H,\sigma,p:(MS\circ VS,return\ v;,FS)) \longrightarrow (H,\sigma,p:(VS,v,FS))}$$

(IFELSE1)

$$\frac{}{(H,\sigma,p:(VS,if\ v_1==v_2\ \{\bar{s}_1\}\ else\ \{\bar{s}_2\};,FS)) \longrightarrow (H,\sigma,p:(VS,\{\bar{s}_1\},FS))}\ v_1=v_2$$

(IFELSE2)

$$\frac{}{(H,\sigma,p:(VS,if\ v_1==v_2\ \{\bar{s}_1\}\ else\ \{\bar{s}_2\};,FS)) \longrightarrow (H,\sigma,p:(VS,\{\bar{s}_2\},FS))}\ v_1\neq v_2$$

(FIELDACCESS)

$$\frac{}{(H,\sigma,p:(VS,o.f,FS)) \longrightarrow (H,\sigma,p:(VS,v,FS))}\ H(o)=p:((C,\mathcal{F})),\mathcal{F}(f)=v,$$

(FIELDACCESS)

$$\frac{}{(H,\sigma,p:(VS,o.f,FS)) \longrightarrow PlaceError}\ H(o)=p':((C,\mathcal{F})),p\neq p'$$

(FIELDWRITE)

$$\frac{}{(H,\sigma,VS,p:(o.f=v;,FS)) \longrightarrow (H',\sigma,p:(VS,;,FS))}\ H(o)=p:(C,\mathcal{F}),f\in dom(\mathcal{F})),H'(o)=H[o\mapsto p:(C,\mathcal{F}[f\mapsto v])]$$

(FIELDWRITE)

$$\frac{}{(H,\sigma,p:(VS,o.f=v;,FS)) \longrightarrow PlaceError}\ H(o)=p':(C,\mathcal{F}),p'\neq p$$

(HERE)

$$\frac{}{(H,\sigma,p:(VS,\texttt{here},FS)) \longrightarrow (H,\sigma,p:(VS,p,FS))}$$

(NEXT)

$$\frac{}{(H,\sigma,p:(VS,\texttt{p'.next},FS)) \longrightarrow (H,\sigma,p:(VS,p'+1,FS))}$$

Reduction2: MJ EVALUATION $(H,\Delta) \longrightarrow (H',\Delta')$

(CAST)

$$\frac{}{(H,\sigma,p:(VS,((C')o),FS)) \longrightarrow (H,\sigma,p:(VS,o,FS))}\ H(o) = p:(C,\mathcal{F}), C \preceq C'$$

(CAST)

$$\frac{}{(H,\sigma,p:(VS,((C')o),FS)) \longrightarrow PlaceError}\ H(o) = p':(C,\mathcal{F}), p' \neq p$$

(NULLCAST)

$$\frac{}{(H,\sigma,p:(VS,((C)null),FS)) \longrightarrow (H,\sigma,p:(VS,null,FS))}$$

(NEW)

$$\frac{cnBody(C) = (\bar{x},\bar{s}), \Delta_c(C) = \bar{C}, o \notin dom(H), \mathcal{F} = [f \mapsto null, f \in fields(C)], BS = [this \mapsto (o,C), \bar{x} \mapsto (\bar{v},\bar{C})]}{(H,\sigma,p:(VS,\texttt{new}\ C(\bar{v}),FS)) \longrightarrow (H[o \mapsto p:(C,\mathcal{F}),\sigma,p:((BS \circ []) \circ VS,\bar{s},(return\ o;) \circ FS))}$$

(SUPER)

$$\frac{MS(this) = C, C \preceq_1 C', cnBody(C') = (\bar{x},\bar{s}), \Delta_c(C') = \bar{C}, BS' = [this \mapsto p:(o,C'), \bar{x} \mapsto (\bar{v},\bar{C})]}{(H,\sigma,p:(MS \circ VS, super(\bar{v}),FS)) \longrightarrow (H,\sigma,p:((BS' \circ []) \circ (MS \circ VS),\bar{s},(return\ o;) \circ FS))}$$

(METHOD)

$$\frac{H(o) = p:(C,\mathcal{F}), mBody(C,m) = (\bar{x},\bar{s}), \Delta_m(C)(m) = \bar{C} \to C', BS' = [this \mapsto (o,C), \bar{x} \mapsto (\bar{v},\bar{C})]}{(H,\sigma,p:(VS,o.m(\bar{v}),FS)) \longrightarrow (H,\sigma,p:((BS' \circ []) \circ VS,\bar{s},FS))}$$

(METHOD)

$$\frac{H(o) = p':(C,\mathcal{F}), p' \neq p}{(H,\sigma,p:(VS,o.m(\bar{v}),FS)) \longrightarrow PlaceError}$$

(METHODVOID)

$$\frac{H(o) = p:(C,\mathcal{F}), mBody(C,m) = (\bar{x},\bar{s}), \Delta_m(C)(m) = \bar{C} \to\ void, BS' = [this \mapsto (o,C), \bar{x} \mapsto (\bar{v},\bar{C})]}{(H,\sigma,p:(VS,o.m(\bar{v}),FS)) \longrightarrow (H,\sigma,p:((BS' \circ []) \circ VS,\bar{s},(return\ o;) \circ FS))}$$

(METHODVOID)

$$\frac{H(o) = p':(C,\mathcal{F}), p' \neq p}{(H,\sigma,p:(VS,o.m(\bar{v}),FS)) \longrightarrow PlaceError}$$

(SKIP)

$$\frac{}{(H,\sigma,p:(VS,;,F \circ FS)) \longrightarrow (H,\sigma,p:(VS,F,FS))}$$

(SUB)

$$\frac{}{(H,\sigma,p:(VS,v,F \circ FS)) \longrightarrow (H,\sigma,p:(VS,;,F[v] \circ FS))}$$

Decomposition Reduction1:

(SEQ)

$$\frac{}{(H,\sigma,p:(VS,s_1s_2\ldots s_n,FS)) \longrightarrow (H,\sigma,p:(VS,s_1,(s_2\ldots s_n)\circ FS))}$$

(RET)

$$\frac{}{(H,\sigma,p:(MS\circ VS,returne;,FS)) \longrightarrow (H,\sigma,p:(MS\circ VS,e,(return\ \bullet;)\circ FS))}$$

(EXPSTATE)

$$\frac{}{(H,\sigma,p:(MS\circ VS,e';,FS)) \longrightarrow (H,\sigma,p:(MS\circ VS,e',FS))}$$

(IF1)

$$\frac{}{(H,\sigma,p:(VS,if\ e_1 == e_2\ \{\bar{s}_1\}\ else\ \{\bar{s}_2\};,FS)) \longrightarrow (H,\sigma,p:(VS,e_1,(if\ \bullet == e_2\ \{\bar{s}_1\}\ else\ \{\bar{s}_2\};)\circ FS))}$$

(IF2)

$$\frac{}{(H,\sigma,p:(VS,if\ v_1 == e_2\ \{\bar{s}_1\}\ else\ \{\bar{s}_2\};,FS)) \longrightarrow (H,\sigma,p:(VS,e_2,(if\ v_1 == \bullet\ \{\bar{s}_1\}\ else\ \{\bar{s}_2\};)\circ FS))}$$

(FIELDACCESS)

$$\frac{}{(H,\sigma,p:(VS,e.f;,FS)) \longrightarrow (H,\sigma,p:(VS,e,(\bullet.f)\circ FS))}$$

(FIELDWRITE1)

$$\frac{}{(H,\sigma,p:(VS,e.f = e';,FS)) \longrightarrow (H,\sigma,p:(VS,e,(\bullet.f = e';)\circ FS))}$$

(FIELDWRITE2)

$$\frac{}{(H,\sigma,p:(VS,v.f = e;,FS)) \longrightarrow (H,\sigma,p:(VS,e,(v.f = \bullet;)\circ FS))}$$

(VARWRITE)

$$\frac{}{(H,\sigma,p:(VS,x = e;,FS)) \longrightarrow (H,\sigma,p:(VS,e,(x = \bullet;)\circ FS))}$$

(CAST)

$$\frac{}{(H,\sigma,p:(VS,(C)e,FS)) \longrightarrow (H,\sigma,p:(VS,e,((C)\bullet)\circ FS))}$$

Decomposition Reduction2:

(NEW-I)

$$\frac{}{(H,\sigma,p:(VS,new\ C(v_1,\ldots,v_{i-1},e_i,\ldots,e_n),FS)) \longrightarrow (H,\sigma,p:(VS,e_i,(new\ C(v_1,\ldots,v_{i-1},\bullet,\ldots,e_n))\circ FS))}$$

(SUPER-I)

$$\frac{}{(H,\sigma,p:(VS,super.(v_1,\ldots,v_{i-1},e_i,\ldots,e_n),FS)) \longrightarrow (H,\sigma,p:(VS,e_i,(super.(v_1,\ldots,v_{i-1},\bullet,\ldots,e_n))\circ FS))}$$

(METHOD1)

$$\frac{}{(H,\sigma,p:(VS,e.m(e_1,\ldots,e_n),FS)) \longrightarrow (H,\sigma,p:(VS,e,(\bullet.m(e_1,\ldots,e_n))\circ FS))}$$

(METHOD2)

$$\frac{}{(H,\sigma,p:(VS,v.m(v_1,\ldots,v_{i-1},e_i,\ldots,e_n),FS)) \longrightarrow (H,\sigma,p:(VS,e_i,(v.m(v_1,\ldots,v_{i-1},\bullet,\ldots,e_n))\circ FS))}$$