

Concurrent Clustered Programming^{*}

Vijay Saraswat^{1**} and Radha Jagadeesan^{2***}

¹ IBM T.J. Watson Research Lab

² School of CTI, DePaul University

Abstract. We present the concurrency and distribution primitives of X10, a modern, statically typed, class-based object-oriented (OO) programming language, designed for high productivity programming of scalable applications on high-end machines. The basic move in the X10 programming model is to reify locality through a notion of *place*, which hosts multiple data items and activities that operate on them. Aggregate objects (such as arrays) may be distributed across multiple places. Activities may dynamically spawn new activities in multiple places and sequence them through a *finish* operation that detects termination of activities. Atomicity is obtained through the use of *atomic blocks*. Activities may repeatedly detect quiescence of a data-dependent collection of (distributed) activities through a notion of *clocks*, generalizing SPMD barriers. Thus X10 has a handful of orthogonal constructs for *space*, *time*, *sequencing* and *atomicity*. X10 smoothly combines and generalizes the current dominant paradigms for shared memory computing and message passing.

We present a bisimulation-based operational semantics for X10 building on the formal semantics for “Middleweight Java”. We establish the central theorem of X10: programs without conditional atomic blocks do not deadlock. We prove other related results.

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, and correct compilers, transformation systems, program development methodologies and interactive refactoring tools developed.

There has always been considerable theoretical research in concurrency – CCS, CSP, process algebras, CCP, π -calculus etc. On the practical front, in imperative languages, CILK[1,2] has introduced some novel ideas such as work-stealing for symmetric multi-processors (SMPs). Titanium [3], Co-Array Fortran [4] and Unified Parallel C [5] have introduced the *Partitioned Global Address Space* (PGAS) model [6] in JAVA, Fortran and C respectively, albeit in a Single Program Multiple Data (SPMD) framework. However, the state of the art in concurrent high performance computing continues

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

^{**} Research supported in part by DARPA No. NBCH30390004.

^{***} Research supported in part by NSF 0430175.

to be library-based (e.g. OpenMP [7] for shared-memory concurrency and MPI [8] for message-passing) rather than language-based. Mainstream languages have been slow to adopt concurrency. JAVATM [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 high performance (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 languages³ 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.

A fundamental new opportunity presents itself with the development of the next generation of high performance computers (e.g. capable of $\approx 10^{15}$ 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.

Together with our colleagues, we have have designed an explicitly parallel programming language for clustered computing, X10 [19], under the aegis of the DARPA HPCS programme. 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. X10 is explicitly parallel because of our unwillingness to rely on the ability of heroic compilers to automatically extract enough parallelism to keep hundreds of thousands of nodes busy. Thus X10 is intended to support in an integrated fashion the set of problems that are today addressed by API's such as OpenMP and MPI bolted onto base programming languages such as Fortran or C.

1.1 Basic paradigm

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

³ 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.

other threads [20,21]. The simplest guarantee – sequentially consistent (SC) execution [22] – 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]).

Space. 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.) A programmer may think of a place as an MPI task or a node in a distributed Java Virtual Machine (JVM) with its own heap and collection of threads.

Each activity has an SC 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* (GALS) computation model, familiar from hardware design and embedded systems research. Unlike other PGAS languages, X10 is not SPMD – different (collections of) activities may run at each place.

All objects created by an activity reside in the place the activity is running.

Sequencing. Since X10 supports fine-grained asynchronous, parallel activities – even a remote read is an activity – a reliable mechanism is needed to detect termination. X10 provides a `finish` construct (Section 2.4). Intuitively, `finish S` executes `S` and suspends until all activities created while doing so have terminated (normally or abruptly).

Atomicity. How can multiple activities running in the same place reliably access shared data? JAVA-like languages support a notion of monitors – the programmer must write code that explicitly obtains and releases *locks* [9]. Our experience is that locks are a very low-level and error-prone synchronization mechanism, making it very easy for programmers to write erroneous code that underlocks (causing race conditions) or overlocks (causing deadlock). Instead X10 supports *atomic blocks* (cf. [17]) (Section 2.2) : a statement `atomic S` is executed as if in a single step without interruption by other activities. A conditional atomic operation, `when` is also provided.

Time. Thus an X10 computation consists of a large number of asynchronous activities scattered across space. We now introduce a notion of *time*. Many scientific computations need to progress in a sequence of *phases*. In each phase, activities (scattered across multiple places) read and write shared data (e.g. a distributed array). Once all activities have performed one phase of their calculations, each is informed of this global quiescence and computation moves to the next phase, and the process repeats. 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).

In SPMD languages this phasing is accomplished using the notion of a *barrier*. For instance, UPC provides a single barrier for all threads in a computation, accessed through `upc_notify` (signal that this thread has reached the barrier) and `upc_wait` (wait until all threads have reached this barrier).

X10 clocks (Section 2.5) are closely related but much more powerful. At the high-level, clocks provide the functionality of barriers in a dynamic context, and they are *race-free*, (hence *determinate*) and *deadlock-free*. By race-free we mean that two operations on the same clock performed at the same time by two separate activities (or by an activity and the clock) commute with each other (hence cannot conflict).

Concretely, a clock is a data-structure that may be dynamically created (`clock c = new clock()`); an activity may create as many clocks as it wishes.⁴ Conceptually each clock is associated with an integer that specifies the *current phase* of the clock; this integer is initially zero, and is incremented each time the clock *advances*. A clock *advances* to the next phase only when all activities *registered* with it have *quiesced* (see below).

The activity creating the clock is automatically registered with it. An activity *A* may at any time deregister itself from clock *c* by executing `c.drop()`; any subsequent attempt to invoke an operation on *c* results in a `ClockUseException` being thrown. *A* may indicate that it has *quiesced* on *c* (in its current phase) by executing `c.resume()`. It may suspend until *all* clocks it is registered with have moved to the next phase by executing the `next;` statement; this automatically resumes all clocks the activity is registered with. (There is no statement allowing *A* to suspend until a *given* clock it is registered with has moved to the next phase; such a statement can easily cause deadlock.)

An activity *A* may register a new activity it is spawning with clocks `c1, ..., cn` by executing `async(P) clocked (c1, ..., cn) s`. We require that *A* itself be *live* on *c* (that is, be registered with *c* and not have quiesced on it); this is called the *Live Clock Condition* (LCC). (If this condition is violated a `ClockUseException` is thrown.)

The LCC ensures that the only way an activity can be registered on a pre-existing clock *c* is through an activity that is live on *c*. While an activity is live on *c*, *c* cannot advance; hence X10 has no race conditions between registration and clock advance. (It is easy to see that permitting an activity to access a clock object through a datastructure and register itself on it could cause a race condition.) The execution of `c.resume()` (or `c.drop()`) operations by two activities commute, hence they do not constitute a race. Thus X10 clocks are race-free.

A key semantic property of a clock is that clock quiescence is stable (Theorem 5): once every activity registered on the clock has quiesced, no further action by any activity can change this fact.

X10 clocks interact with `finish`. It is easy to see that the statement `finish async clocked(c) next;` deadlocks when executed by an activity *A* registered on *c*. (*A* cannot advance till the `async` terminates; that cannot happen until *A* executes `c.resume()`.) To ensure deadlock freedom, X10 requires that an activity must not spawn a clocked `async` while executing the body of a `finish`. This can be

⁴ In particular, we remark that clocks may be used to obtain oversampling through nesting.

accomplished dynamically by throwing a `ClockUseException` in such a case (Section 2.5) or statically, with appropriate type rules.

The fundamental theorem of X10 is that these conditions are sufficient to ensure that programs without `when` are deadlock-free (Theorem 9).

To illustrate some of the power of clocks, we present:

EXAMPLE 1 (NOW). Imagine we wish to define a construct `now (c) s` intended to ensure that execution of statement `s` terminate completely in the current phase of the clock `c`. This may be accomplished by: `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`).

1.2 Rest of this paper

This completes a description of the basic concurrency and distribution primitives in X10. We briefly mention those aspects of X10 that are not covered in this abstract for reasons of space (details in [19]). X10 supports a notion of immutable datastructures called *value types* and an explicit `nullable` type annotation (to specify that the type contains the value `null`). X10 supports multi-dimensional arrays that may be distributed across multiple places, using the concept of named *regions* (set of index points), and *distributions* (mapping of these points to places). X10 also has a static place-based type system (augmented with dynamic place-casts).

The rest of this paper presents a formal operational semantics for FOUNDATIONAL X10 (FX10), 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 primary contributions of this paper are as follows. (1) We present a simple programming model for clustered computing. (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 [24]) and (sequential) imperative programming (MJ [25], Classic Java [26]), and may also be seen as a core concurrent object-oriented calculus, suitable for the study of concurrency in OO languages.

1.3 Related work

X10 is a member of the PGAS family of languages and is distinguished from them in not being based on an SPMD model, permitting multiple activities per locale or

place, supporting very general notions of clocked computations, supporting sequencing of distributed computations (through `finish`), and using atomic blocks for mutual exclusion.

The X10 `async` and `finish` operations are related to CILK's `spawn` and `sync` constructs but are not arbitrarily scoped to methods. (CILK has no notion of places, distributed arrays, clocks or atomic blocks.)

While being similar to JAVA in its sequential aspects, X10 has a completely different concurrency and distribution model. All the Java Grande Forum benchmarks [27] that use threads (crypt, lufact, moldyn, montecarlo, raytracer, series, sor) have been ported to the deadlock-free fragment of X10.

An MPI program may be represented in X10 with a place per MPI process, running a single main activity. The MPI-2 communication primitives can be directly implemented with `asyncs`.

While there has been a lot of work on formal models for concurrency, there has been less work on formal models for real concurrent languages. We have chosen to design X10 on top of a modern OO language and present the semantics as such in this paper. However the core concurrency and distribution model can also be adapted for other imperative languages such as C or Fortran.

2 The X10 Programming Model

Our presentation is built on top of the MJ calculus [25]. It includes mutable state, block structured values and basic object-oriented features. For lack of space, it does not present X10's packages, import statements, interfaces, arrays, method overloading, static state, try/catch/throws, and loops; these may be added in a routine fashion.

In MJ, the basic structure of the configurations and the semantics may be summarized as follows. A *configuration* consists of a quadruple (H, VS, s, FS) where:

- H represents the heap of objects. The heap is represented as a binding of object names to a pair of the class name and a finite function mapping field names to values (objects or basic values).
- VS , the *variable stack*, represents the block structure of the underlying programming language. The variable stack changes during reduction whenever a new scope is added or removed.
- s is the statement currently being executed.
- FS the frame stack, represents the continuation that follows the execution of s . In the case that s is an expression that evaluates to a value (say v), the head of the frame stack is an *open frame* with a hole to indicate the position at which v is to be substituted. Otherwise (s is a statement without a return value), the head of the frame stack is a closed frame without a hole.

We refer the reader to [25] for a detailed description of MJ, focussing in this extended abstract on the changes. Figure 1 describes the features we add to core MJ. The transition relation relates configurations. In a configuration (H, σ, Δ) , H is a shared heap and σ is a shared constraint store used to model clocks. In a basic configuration, Δ is an activity that includes a place designator, a frame (representing the continuation) and a

$pe ::= (\text{Place Expression})$	(FX10 Conf.)	$Xconfig ::= (H, \sigma, \Delta) \mid E$
$\text{here} \mid pe.\text{next} \mid v.\text{place}$	(Activity)	$a ::= p : (VS, CF, FS, K, S)$
$s ::= (\text{Statement})$	(Term. Activity)	$ta ::= p : (VS, :, [], [], [])$
$\text{when}(c) \ s$	(Places)	$p, q ::= int$
$\text{async}(p) \text{ clocked}(\bar{c}) \ s$	(Closed Frame)	$CF ::= s \mid \text{wait};$
$\text{finish } s$	(Open Frame)	$OF ::= \text{async}(\bullet) s$
$\text{next};$		$\mid \text{when}(\bullet) s$
$\text{clock } x \text{ is new}$	(Values)	$v ::= \text{null} \mid o \mid p$
$\text{resume } c$	(FinishStack)	$F ::= \varepsilon \mid FS \circ F$
$\text{drop } c$	(Error)	$E ::= \text{BadPlaceException}$
		$\mid \text{ClockUseException}$

Table 1. Syntax and Configurations for FX10

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*.

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

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 $1(2(3,4),5(6))$ (ignoring labels on nodes) 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 $\Delta[2 \triangleright \Delta'] \longrightarrow \Delta[8(9)]$ gives the tree $1(8(9),5(6))$ (the entire subtree at 2 is replaced).

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

We now discuss the transitions for the specific language constructs.

2.1 Places and activities

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 $p.\text{next}$ 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.⁵

The variable “`here`” evaluates to the current place.

$$\frac{\text{(HERE)}}{(H, \sigma, p : (\text{here}, S)) \longrightarrow (H, \sigma, p : (p, S))}$$

⁵ This is consistent with most MPI programs that are started with a fixed number of processes.

The heap has place information for each object: this is recoverable using `o.location`. Access of fields (other than `location`) is permitted only on objects at the same place. Access to objects located at a different place leads to `BadPlaceException`.

$$\begin{array}{c}
\text{(NEW)} \\
\begin{array}{l}
C \text{ is not clock} \\
cnBody(C) = (\bar{x}, \bar{s}), \Delta_c(C) = \bar{C}, o \notin dom(H), \\
\mathcal{F} = [f \mapsto null, f \in fields(C)], BS = [this \mapsto (o, C), \bar{x} \mapsto (\bar{v}, \bar{C})]
\end{array} \\
\hline
(H, \sigma, p : (\text{new } C(\bar{v}), (VS, FS, K, F))) \\
\longrightarrow (H[o \mapsto p : (C, \mathcal{F})], \sigma, p : (\bar{s}, ((BS \circ []) \circ VS, (return\ o;) \circ FS), K, F)) \\
\\
\text{(FIELDACCESS)} \\
\begin{array}{l}
f \text{ is not location } H(o) = p : ((C, \mathcal{F})), \mathcal{F}(f) = v, \\
(H, \sigma, p : (o.f, S)) \longrightarrow (H, \sigma, p : (v, S))
\end{array} \\
\\
\text{(FIELDACCESSPLACE)} \\
\begin{array}{l}
H(o) = q : ((C, \mathcal{F})) \\
(H, \sigma, p : (o.location, S)) \longrightarrow (H, \sigma, p : (q, S))
\end{array} \\
\\
\text{(FIELDACCESS)} \\
\begin{array}{l}
H(o) = q : ((C, \mathcal{F})), p \neq q \\
(H, \sigma, p : (o.f, S)) \longrightarrow \text{BadPlaceException}
\end{array}
\end{array}$$

2.2 Atomic blocks

X10 introduces statements of the form `when (c) s` where `s` is a statement ([17,28,29]). 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`.⁶ 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.

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 9).

EXAMPLE 2 (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;
}

```

⁶ Note that a “multi-armed” version `when (c1) s1 else (c2) s2 ... else (ck) sk` can be defined using `when` and a conditional statement.

CAS is the basis for many highly concurrent, non-blocking (lock-free, wait-free) data-structures (e.g. [30,31]). These algorithms can be expressed directly in UFX10.

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

$$\begin{array}{c}
 \text{(ATOMIC)} \\
 \frac{
 \begin{array}{l}
 (H, \sigma, p : (e, (VS, [], K, F))) \xrightarrow{*} (H_1, \sigma_1, p : (true, (VS_1, [], K, F))) \\
 (H_1, \sigma_1, p : (s, (VS_1, [], K, F))) \xrightarrow{*} (H_2, \sigma_2, p : (;, (VS_2, [], K, F))) \mid E
 \end{array}
 }{
 (H, \sigma, p : (\text{when}(e) s, (VS, FS, K, F))) \longrightarrow (H_2, \sigma_2, p : (;, (VS_2, FS, K, F))) \mid E
 }
 \end{array}$$

$$\begin{array}{c}
 \text{(ATOMIC)} \\
 \frac{
 (H, \sigma, p : (e, (VS, [], K, F))) \xrightarrow{*} E
 }{
 (H, \sigma, p : (\text{when}(e) s, (VS, FS, K, F))) \longrightarrow E
 }
 \end{array}$$

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 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`.

EXAMPLE 3 (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:⁷

<pre> 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; } } </pre>	<pre> Object force() { when (forced) { return result; } } </pre>
---	--

Async without clocks In `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

⁷ 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.

environment (the static semantics ensures only `final` variables can be accessed in VS).

$$\begin{array}{c}
 \text{(ASYNC1)} \\
 \hline
 (H, \sigma, p : (\text{async}(e) \ s, (VS, FS, K, F))) \longrightarrow (H, \sigma, p : (e, (VS, \text{async}(\bullet) \ s \circ FS, K, F))) \\
 \text{(ASYNC2)} \\
 \hline
 (H, \sigma, p : (\text{async}(q) \ s, (VS, FS, K, F))) \longrightarrow (H, \sigma, p : (;, (VS, FS, K, F)) \triangleright q : (VS, s, [], [], []))
 \end{array}$$

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.

EXAMPLE 4 (FUTURES). Consider a new expression of the form `future (p) {e}` where e is of type T . 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.

The `finish` rule creates a closed activity.

$$\begin{array}{c}
 \text{(FINISH1)} \\
 \hline
 (H, \sigma, p : (\text{finish}(s), (VS, FS, K, F))) \longrightarrow (H, \sigma, FS : p : (s, (VS, [], K, FS \circ F)))
 \end{array}$$

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

$$\begin{array}{c}
 \text{(FINISH2)} \\
 \Delta \text{ is a tree of terminated activities} \\
 \hline
 (H, \sigma, p : (;, (VS, [], K, FS \circ F)) \triangleright \Delta) \longrightarrow (H, \sigma, p : (;, (VS, FS, K, F)))
 \end{array}$$

$$\begin{array}{c}
 \text{(FINISH3)} \\
 \Delta \text{ is a tree containing an exception} \\
 \hline
 (H, \sigma, p : (;, (VS, [], K, F)) \triangleright \Delta) \longrightarrow \text{FinishException}
 \end{array}$$

2.5 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 [32,33]. This technique makes the proof of the Clock Quiescence Stability theorem

(Theorem 5) immediate. We note that this technique is used purely to specify the *semantics* of clocks. Clocks may be implemented using semaphores with counters on a shared memory machine, augmented with message passing on a multi-node machine.

In essence, the technique uses constraints to implement a *distributed stable counter* (henceforth: counter). A counter X is equipped with the following operations: (1) set to zero, (2) split and (3) check if zero. A counter r can only be split if it is not zero; two new counters are created and when both reach zero, r is set to zero. Once zero, the counter stays at zero, hence the success of the check is stable. These operations may be implemented with constraints as follows: A counter is represented by a variable X , it is set to zero by asserting $X=0$, it is split by asserting $X=Y+Z$, where Y and Z are two new variables, and it is checked by asking if $X=0$.

Clocks require a check for quiescence in each phase, hence we need a *stream* of counters, a *counterstream*.

Formally, a *constraint store* σ is a set of constraints, equipped with a function **var** which represents the set of variables over which the constraints are defined. If X does not occur in σ , then we write $\sigma + X$ to indicate a constraint store identical to σ except that **var**($\sigma + X$) = **var**(σ) \cup $\{X\}$. The relevant constraints are:

$$\begin{aligned} \text{(Term)} \quad t &::= X \mid 0 \mid t + t \mid t.\text{cdr} \mid t.\text{car} \\ \text{(Constraint Store)} \sigma &::= \text{true} \mid t = t \mid 0(t) \mid \sigma, \sigma \end{aligned}$$

with the obvious entailment relation, augmented with the axioms: $0(X), X.\text{car} = Z \vdash Z=0$ and $0(X), X.\text{cdr} = Z \vdash 0(Z)$.

A *clock-counter* is a pair of terms $\langle g, l \rangle$, where g is the *global* counterstream and l the *local* counterstream. We will arrange matters so that if the set of activities registered with a clock c is A_1, \dots, A_n , then each A_i has a clock-counter (g, l_i) , and the store has the constraint $g.\text{car} = l_1.\text{car} + \dots + l_n.\text{car}$. When activity A_i performs a *resume* it asserts the constraint $l_i.\text{car} = 0$. A_i can determine when all activities have quiesced by checking $g=0$. It can move to the next phase by progressing with the clock-counter $(g.\text{cdr}, l_i.\text{cdr})$. It can drop the counter by asserting $0(l_i)$. No separate active representation of a clock is needed.

We now present the formal rules capturing these ideas. We augment the state of each activity with a *clock map* (henceforth: map) K (a finite partial function from variables to clock-counters). We use ε to indicate the unique map with empty domain. If $K(c) = (x, y)$, we use $K_g(c)$ for x and $K_l(c)$ for y . We use $|K|$ for the domain of K ; $K[c \mapsto X_c \mid \phi]$ for K extended with the value X_c for each c satisfying ϕ (we drop K when it is ε , the empty map); and $K \setminus c$ for K with c removed from its domain.

$$\begin{array}{c} \text{(NEW CLOCK)} \\ \hline (H, \sigma, p : (\text{clock } x = \text{new clock } (); (VS, FS, K, F))) \\ \longrightarrow (H, \sigma + g, p : (, (VS, FS, K[x \mapsto (g, g)], F))) \end{array}$$

(We assume alpha renaming to ensure that x and g are new.) Note that for a newly created clock the global counterstream is the same as the local counterstream, reflecting the fact that the clock has a single activity registered with it.

Clocks may be transmitted to new activities when they are created.

(CLOCK-ASYNC)

$$\frac{c_0, \dots, c_{n-1} \in |K| \quad K' = K[c_i \mapsto (K_g(c_i), X_i) \mid i < n] \quad K'' = [c_i \mapsto (K_g(c_i), Y_i) \mid i < n]}{(H, \sigma, p : (\text{async}(q) \text{clocked}(c_0, \dots, c_{n-1}) s, (VS, FS, K, []))) \longrightarrow (H, \sigma \cup \{K_l(c_i) = X_i + Y_i \mid i < n\}, p : (:, (VS, FS, K', [])) \triangleright q : (s, (VS, [], K'', [])))}$$

(CLOCK-ASYNC-LLC)

$$\frac{\{c_0, \dots, c_{n-1}\} \not\subseteq |K|}{(H, \sigma, p : (\text{async}(q) \text{clocked}(c_0, \dots, c_{n-1}) s, (VS, FS, K, F))) \longrightarrow \text{ClockUseException}}$$

(CLOCK-ASYNC-FINISH)

$$(H, \sigma, p : (\text{async}(q) \text{clocked}(c_0, \dots, c_{n-1}) s, (VS, FS, K, FS' \circ F))) \longrightarrow \text{ClockUseException}$$

(RESUME)

$$(H, \sigma, p : (\text{resume } c; S)) \longrightarrow (H, \sigma \cup \{K_l(c).car = 0\}, p : (:, S))$$

(NEXT)

$$\frac{\sigma' = \sigma \cup \{K_l(c).car = 0 \mid c \in |K|\}}{(H, \sigma, p : (\text{next } ; S)) \longrightarrow (H, \sigma', p : (\text{wait } ; S))}$$

(WAIT)

$$\frac{\sigma \vdash K_g(c).car = 0 \quad (\forall c \in |K|) \quad K' = [c \mapsto (K_g(c).cdr, K_l(c).cdr) \mid c \in |K|]}{(H, \sigma, p : (\text{wait } ; (VS, FS, K, F))) \longrightarrow (H, \sigma, p : (:, (VS, FS, K', F)))}$$

(DROP)

$$(H, \sigma, p : (c.\text{drop}(); (VS, FS, K, F))) \longrightarrow (H, \sigma \cup \{0(K_l(c))\}, p : (:, (VS, FS, K \setminus c, F)))$$

(TERMINATE)

$$(H, \sigma, p : (:, (VS, [], K, F))) \longrightarrow (H, \sigma \cup \{0(K_l(c)) \mid c \in |K|\}, p : (:, (VS, [], [], F)))$$

THEOREM 5 (CLOCK QUIESCENCE IS STABLE). *Let configuration (H, σ, Δ) be such that $\sigma \vdash X.car = 0$ where X is the global counterstream of a clock in the clock set of some activity in Δ . Let $(H, \sigma, \Delta) \longrightarrow (H', \sigma', \Delta')$. Then $\sigma' \vdash X.car = 0$.*

The only operations performed on the constraint store are Ask and Tell operations [33]. So, the theorem follows from the monotonicity of the constraint store.

3 Properties of FX10 programs

3.1 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 $H(o) = (C, \mathcal{F}), f \in \text{dom}(\mathcal{F})$ then, the resulting heap is $\lambda H = H[o \mapsto p : (C, \mathcal{F}[f \mapsto o'])]$. 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 6. 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 6 enables us to lift the congruence property to all contexts including tree contexts.

LEMMA 7. 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 following equations hold upto bisimulation.

$$\begin{aligned}
& \text{when}(c) \text{ when}(d) \ s \equiv \text{when}(c \& \& d) \ s \\
& \text{atomic} \{ \ s1 \ \text{atomic} \ s2 \} \equiv \text{atomic} \{s1 \ s2\} \\
& \text{async}(P)\{s\}; \text{async}(Q)\{s_1\} \equiv \text{async}(Q)\{s_1\}; \text{async}(P)\{s\} \\
& \text{async}(P)\{\text{async}(Q)\{s\}\} \equiv \text{async}(Q[\text{here}/P])\{s\} \\
& \text{async}(P)\{\text{async}(Q)\{s\}s_1\} \equiv \text{async}(Q[\text{here}/P])\{s\}; \text{async}(P)\{s_1\} \\
& \text{finish} \{s \ s1\} \equiv \text{finish} \ s \ \text{finish} \ s1 \\
& \text{finish} \ \text{when}(c) \ \ s \equiv \text{when}(c) \ \text{finish} \ s \\
& \text{finish} \ \text{if}(c)\{s\} \ \text{else} \ \{s1\} \equiv \text{if} \ (c)\{\text{finish} \ s\} \ \text{else}\{\text{finish} \ s1\} \\
& \text{finish} \ x=e \equiv x=e \\
& \text{finish} \ \text{async}(p) \ \{\} \equiv \{\}
\end{aligned}$$

3.2 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 $S_{\text{Coord}} = \{\text{here}, \text{here.next}, \text{here.next.next}, \dots\}$. Let s be such that no transition sequence from $(\langle \rangle, \sigma, p : (\langle \rangle, s, \langle \rangle))$ leads to `PlaceError`.

LEMMA 8. *Let Θ be an operator on the set S_{Coord} . Let $\text{trans}(\Theta, s)$ be the result of replacing every subexpression $\text{async}(p)$ in s by $\text{async}(\Theta(e))$. Then: $(\langle \rangle, \sigma, p : (\langle \rangle, s, \langle \rangle)) \equiv (\langle \rangle, \sigma, p : (\langle \rangle, \text{trans}(\Theta, s), \langle \rangle))$.*

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.⁸

3.3 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 `finishes` to each activity spawned by s that is suspended. 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 no activity has an incoming edge in this case. Deadlock-freedom holds for a larger language that encompasses lock-free computations, CILK programs, systolic arrays and MPI computations.

⁸ Note that this property does not hold for FX10 because of clocks.

THEOREM 9. *There are no cycles in the wait-for graph for programs in the language with `atomic`, `clocks`, and `finish`.*

The proof builds on the case for clocks only and exploits the stratification on activities induced by the tree structure of the configuration.

4 Conclusion and future research

We believe that X10 offers a simple, clean but real design for high-productivity, high-performance concurrent programming for high-end computers. A reference manual for the language has been completed [19]. The language has been implemented via a translator to JAVA, developed using the Polyglot compiler framework.

Early experience with writing programs in X10 (and running them on the reference implementation) is quite encouraging [34]. Several lock-based concurrent JAVA programs, and HPCS programs have been rewritten cleanly and succinctly in X10. Work is underway to apply X10 to “hard” HPCS problems such as adaptive mesh refinement [?,?], and to develop refactorings that change concurrency and distribution patterns in the underlying program.

However, these are just the first stages of X10 development. Considerable additional work is needed to establish efficient compilers and multi-node virtual machines for X10, e.g. using LAPI [35] for messaging.

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 Interface. 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. Saraswat, V.: Report on the Experimental Language X10, v0.41. Technical report, IBM Research (2005)
20. Adve, S., Pai, V.S., P.Ranganathan: Recent Advances in Memory Consistency Models for Hardware Shared-Memory Systems. *Proceedings of the IEEE* **87** (1999) 445–455
21. Adve, S., Gharachorloo, K.: Shared Memory Consistency Models: A tutorial. Technical report, Digital Western Research Laboratory (1995)
22. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers* **28** (1979)
23. Saraswat, V., Jagadeesan, R.: Foundations of X10. Technical report, IBM Research (forthcoming)
24. 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
25. 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)
26. 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
27. : (The java grande forum benchmark suite) www.epcc.ed.ac.uk/javagrande/javag.html.
28. Hansen, P.B.: Structured multiprogramming. *CACM* **15** (1972)
29. Hoare, C.: Monitors: An operating system structuring concept. *CACM* **17** (1974) 549–557
30. Herlihy, M.: Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* **13** (1991) 124–149
31. 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
32. 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
33. Saraswat, V.: *Concurrent Constraint Programming*. Doctoral Dissertation Award and Logic Programming. MIT Press (1993)
34. 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) To appear in *OOPSLA 2005 Onwards! Track Proceedings*.
35. Center, I.I.T.S.O.P.: Technical presentation for pssp version 2.3. Technical report sg24-2080-00 (1997) <http://www.redbooks.ibm.com/redbooks/pdfs/sg242080.pdf>.