Concurrent Clustered Programming* (Extended Abstract)

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 mulitple 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 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.

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] (UPC) have introduced the *Partitioned Global Address Space* (PGAS) model [6] in JAVA, Fortran and C respectively, albeit in a Single Program Multiple Data (SPMD)

* We thank Bard Bloom, 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.
** Research supported in part by DARPA No. NBCH30390004.
*** Research supported in part by NSF 0430175.

framework. However, the state of the art in concurrent high performance computing continues 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. 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 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. The exciting new idea of *atomic blocks* [14,15,16] 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 [17], 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 heroic compilers to automatically extract enough parallelism to keep hundreds of thousands of nodes busy. For productivity, we have chosen to design X10 in the familiar statically typed, class-based, object-oriented programming mould; X10 is intended to be readily accessible to programmers in JAVA-like languages. Thus X10 is intended to support in an integrated fashion the set of problems that are today addressed by libraries such as OpenMP and MPI bolted onto base programming languages such as Fortran or C.

A reference manual for the language has been completed [17]. The language has been implemented via a translator to JAVA, developed using the Polyglot compiler framework. A number of programs have been written in X10 and preliminary productivity measures are reported in [18]. In this paper we lay out the basic semantic foundations of X10.

1.1 Basic paradigm

Space. Local vs remote memory latency and bandwidth ratios for large scale-out machines are often higher than two, perhaps three orders of mangitude. Another problem

is that current architecture research has not yet established the efficiency of sequentically consistent (SC) execution of threads. 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 dilemma 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 Virutal Machine (JVM) with its own heap and collection of threads.

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 `final` variables.

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* (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.

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.[3] Each object carries its location through a final field `location`. Access to non-final fields is permitted only for objects at the same place. Any attempt to access remote mutable data results in a `BadPlaceException` (`BPE`).

EXAMPLE 1 (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 X10 thus:[4]

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

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

[4] In X10, 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 X10 we do not discuss in this paper for lack of space.

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).

EXAMPLE 2 (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.location){L.setValue(X);}}
```

This example shows how distributed datastructures may be created in X10 (even without using distributed arrays); the field of an object may contain a reference to an object at a different place.

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. [15,19,20]) (Section 2.2). The statement `when(c) s` where `s` is 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.

`when` is the only construct for atomicity and mutual exclusion in X10: constructs such as clocks (Section 2.5) can be expressed using `when`. This power comes at the cost of potential deadlock, a risk that can be avoided by using the more restrictive clocks.

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

EXAMPLE 3 (CAS). The following class implements a *compare and swap* (CAS) operation, the basis for many highly concurrent, non-blocking (lock-free, wait-free) datastructures (e.g. [21,22]). 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;
 }
```

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 (split-phase) *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) can be thought of as obtained from split-phase barriers while (1) permitting dynamic creation, (2) permitting dynamic (de-) registration of activities, and (3) ensuring that operations are *race-free* (hence *determinate*). By race-free we mean that two operations on the same clock performed at the same time by two separate activities commute with each other (hence cannot conflict).

Concretely, a clock is a data-structure that may be dynamically created (`clock c is new`); an activity may create as many clocks as it wishes.[5] 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 is said to *advance* to the next phase 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 by *A* to invoke an operation on `c` results in a `ClockUseException` (`CUE`) 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 the *Live Clock Condition* (LCC) to hold: *A* itself be *live* on `ci` (for `i` in `1,...,n`). That is, *A* should be registered with `ci` and not have quiesced on it. A `ClockUseException` is thrown if this condition is violated.

The LCC ensures that the only way an activity can be registered on a pre-existing clock `c` is if it is created by 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 read a clock as the value of some field of some object 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.

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

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. Therefore when the *last* activity quiesces, it can trigger a clock advance.

`finish` interacts with clocks. `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 the activity executing the body of a `finish` must not spawn a clocked async while doing so. This can be 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).

EXAMPLE 4 (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 [17]). X10 supports a *rooted, synchronous, non-resumptive* exception model, with a `try/catch/finally` construct. An exception thrown by an abruptly terminating activity *A* is caught by the enclosing activity suspended on a `finish` waiting for *A* to terminate. This paper, however, permits exceptions to be raised but not caught; thus any exception raised is fatal and terminates the entire computation. 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 the concurrency and distribution features 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

– including `finish` and nested clocks – cannot deadlock. (3) We formalize a compositional operational semantics based on bisimulation. (4) 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. We refer the reader to the sister paper [18] for a discussion of how lock-free computations, CILK programs, systolic arrays and MPI computations can be expressed in this subset.

The model is formalized in the style of previous JAVA-centric calculi focusing on types ([23]) and (sequential) imperative programming (MJ [24]).

1.3 Related work

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`.

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 [25] 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`.

2 The X10 Programming Model

Our presentation is built on top of the MJ calculus [24]. It includes mutable state, block structured values and basic object-oriented features. Additional sequential constructs may be added in a routine fashion.

An MJ *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.

This structure is changed for X10 by taking a configuration to be a triple (H, σ, Δ) where H is a heap (changed from MJ to include place information with each object), σ is a constraint store used to model clocks and Δ is a tree each of whose nodes is labeled with an *activity*. An activity is of the form $p : (s, (VS, FS, K))$ where p indicates the place of the activity, *VS* and *FS* are as above and K is a *clock-map* associating object id's representing clocks with their associated data structure (clock-counters, Section 2.5). These changes are summarized in Figure 1.

The Table is to be taken in conjunction with Figure 1 and the Table in Section 2.3 of [24]. The former defines the syntactic categories programs (p), class (cd), field (fd), constructor(cnd), method (md) definitions, expressions (*MJe* below), and statements (*MJs*). The latter defines MJ's Variable Stack (*MJVS*), Closed Frame (*MJCF*), and Open Frame (*MJOF*). We refer the reader to [24] for a detailed description of MJ.

`e` ::=pe | MJe	(X10 Conf.)	Xc	::=	$(H, \sigma, \Delta) \mid E$
`s` ::=(Statement)	(Activity)	a	::=	$p : (s, (\text{VS}, \text{FS}, K)) \mid E$
`when(c) s`	(Term. Activity)	ta	::=	$p : (;, (\text{VS}, [], [])) \mid E$
`async(p)clocked(`$\bar{\text{c}}$`)s`	(Frame Stack)	FS	::=	$F \circ \text{FS} \mid []$
`finish` s	(Frame)	F	::=	CF | OF
`next;`	(Variable Stack)	VS	::=	MJVS
`clock x is new`	(Places)	p,q	::=	int
`resume` c	(Closed Frame)	CF	::=	`waitn;`
`drop` c				`waitf;` | MJCF
MJs	(Open Frame)	OF	::=	`async(`$\bullet$`)` s
pe::=(Place Expression)				| `when(`$\bullet$`)` s | MJOF
`here` | `pe.next` | `v.place`	(Values)	v	::=	`null` | o | p
	(Error)	E	::=	`BPE` | `CUE` | `FE` | `NPE` | `CCE`

Table 1. Syntax and Configurations for X10

The transition relation relates configurations. X10 specifies the top-level statement is executed implicitly in a `finish`.

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

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 $A_1(A_2(A_3, A_4), A_5(A_6))$ then an application of the

rule $\Delta[A_2] \longrightarrow \Delta[A_8(A_9)]$ gives the tree $A_1(A_8(A_9,A_3,A_4),A_5(A_6))$. An application of the rule $\Delta[A_2 \rhd \Delta'] \longrightarrow \Delta[A_8(A_9)]$ gives the tree $A_1(A_8(A_9),A_5(A_6))$ (the entire subtree at A_2 is replaced).

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

MJ *transitions.* The transition system incorporates *mutatis mutandis* all the MJ reduction and decomposition reduction rules ([24, Fig 2,3])) for the various MJ constructs, except for changes caused by the introduction of places. These changes are: the rule (E-New) is replaced by (New) below (to ensure the new object is created at the right place); the rules (E-Method), (E-MethodVoid), (E-FieldAccess) and (E-FieldWrite) are replaced by rules that check that the target object is local. We illustrate below with FieldAccess.

2.1 Places and activities

The heap has place information for each object, recoverable using the final field `location`. Access to non-final fields is permitted only for objects at the same place. Access to objects located at a different place leads to a `BPE`.

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

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

(FIELDACCESS)
$$\frac{H(o) = q:((C,\mathcal{F})), \mathcal{F}(f) = v, q = p \text{ or } f \text{ is final}}{(H,\sigma,p:(o.f,S)) \longrightarrow (H,\sigma,p:(v,S))}$$

(FIELDACCESSBPE)
$$\frac{H(o) = q:((C,\mathcal{F})), p \neq q \text{ and } f \text{ is not final}}{(H,\sigma,p:(o.f,S)) \longrightarrow \texttt{BPE}}$$

2.2 Atomic blocks

`when`(e) s completes in one step if and when e evaluates to true in the current store and without interruption s completes execution. X10 syntax rules guarantee that an atomic block cannot execute an `async` or a clock operation; hence K remains unchanged in the antecedent of Rule Atomic1.

(ATOMIC1)
$$\frac{\begin{array}{l} (H,\sigma,p:(e,(VS,[],K))) \xrightarrow{\star} (H_1,\sigma_1,p:(true,(VS_1,[],K)) \\ (H_1,\sigma_1,p:(s,(VS_1,[],K))) \xrightarrow{\star} (H_2,\sigma_2,p:(;,(VS_2,[],K))) \mid E \end{array}}{(H,\sigma,p:(\texttt{when}(e)\ s,(VS,FS,K))) \longrightarrow (H_2,\sigma_2,p:(;,(VS_2,FS,K))) \mid E}$$

(ATOMIC2)
$$\frac{(H,\sigma,p:(e,(VS,[],K))) \xrightarrow{\star} E}{(H,\sigma,p:(\texttt{when}(e)\ s,(VS,FS,K))) \longrightarrow E}$$

2.3 Asynchronous activities

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

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

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

2.4 `finish`

The finish rule creates a nested activity, with the given variable stack and clocks but no continuation. [6] On termination of this activity and its subtree the parent activity may continue, with updated VS and K. The second and third rules replace an entire subtree of terminated activities with a single node. (For the purposes of the simpler exception semantics of this paper, the last rule could have been simplified to propagate exceptions more eagerly.)

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

(FINISH2)
$$\frac{\Delta \text{ is a tree of terminated activities w/ no exceptions}}{(H,\sigma,p:(\texttt{waitf;},([],FS,[]))\rhd q:(;,(VS,[],K))\rhd\Delta) \longrightarrow (H,\sigma,p:(;,(VS,FS,K)))}$$

(FINISH3)
$$\frac{\Delta \text{ is a tree of terminated activities containing an exception}}{(H,\sigma,p:(\texttt{waitf;},([],FS,[]))\rhd\Delta) \longrightarrow \texttt{FE}}$$

In the last rule the exception could have been propagated more eagerly; we choose the above formulation because it reflects the semantics of `finish` in the richer model in which exceptions are propagated and may be caught.

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

[6] This nesting is necessary: consider `finish {s1; finish {s2;} s3;}`. `s3` cannot be initiated until all the activities spawned by `s2` have terminated; but there is no requirement that activities spawned by `s1` have terminated.

(NEW CLOCK)
$$\frac{}{(H,\sigma,p:(\texttt{clock x is new;},(VS,FS,K))) \longrightarrow (H,\sigma+g,p:(;,(VS,FS,K[x\mapsto(g,g)])))}$$

(CLOCK-ASYNC)
$$\frac{\{c_0,\ldots,c_{n-1}\}\subseteq|K|,\texttt{waitf;}\text{ not in }FS,\quad K'=K[c_i\mapsto(K_g(c_i),X_i)\mid i<n],K''=[c_i\mapsto(K_g(c_i),Y_i)\mid i<n]}{(H,\sigma,p:(\texttt{async}(q)\texttt{clocked}(c_0,\ldots,c_{n-1})\ s,(VS,FS,K))) \longrightarrow (H,\sigma\cup\{K_l(c_i)=X_i+Y_i\mid i<n\},p:(;,(VS,FS,K'))\rhd q:(s,(VS,[],K'')))}$$

(CLOCK-ASYNC-EXCEPTION)
$$\frac{\{c_0,\ldots,c_{n-1}\}\not\subseteq|K|\ \textit{or}\ \texttt{waitf;}\ \text{in}\ FS}{(H,\sigma,p:(\texttt{async}(q)\texttt{clocked}(c_0,\ldots,c_{n-1})\ s,(VS,FS,K)))\longrightarrow \texttt{CUE}}$$

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

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

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

(DROP)
$$\frac{}{(H,\sigma,p:(\texttt{drop c},(VS,FS,K)))\longrightarrow(H,\sigma\cup\{0(K_l(c))\},p:(;,(VS,FS,K\setminus c)))}$$

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

Table 2. Clock Rules

[26,27]. 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.

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 *counter-stream*.

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 $\mathbf{var}(\sigma + X) = \mathbf{var}(\sigma) \cup \{X\}$. The relevant constraints are:

$$\begin{array}{ll} \textit{(Term)} & t \ ::= \mathtt{X} \mid \mathtt{0} \mid \mathtt{t}+\mathtt{t} \mid \mathtt{t.cdr} \mid \mathtt{t.car} \\ \textit{(Constraint Store)} & \sigma ::= \mathtt{true} \mid \mathtt{t}=\mathtt{t} \mid \mathtt{0(t)} \mid \sigma,\sigma \end{array}$$

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

A *clock-counter* is a pair of terms $\langle g,l \rangle$, where g is the *global* counter-stream and l the *local* counter-stream. We will arrange matters so that if the set of activities registered with a clock c is $A_1,\ldots,A_n$, then each A_i has a clock-counter (g,l_i), and the store has the constraint `g.car= l1.car+ ...+ ln.car`. When activity A_i performs a `resume` it asserts the constraint `li.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.cdr, li.cdr)`. It can drop the counter by asserting `0(li)`. No separate active representation of a clock is needed.

In Table 2, we 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 oids 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.

In the rule for new clocks, we assume alpha renaming to ensure that x and g are new. Note that for a newly created clock the global counter-stream is the same as the local counter-stream, reflecting the fact that the clock has a single activity registered with it. Clocks may be transmitted to new activities when they are created.

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

The only operations performed on the constraint store are Ask and Tell operations [27]. So, the theorem follows from the monotonicity of the constraint store.

3 Properties of X10 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. Thus, our treatment does not validate enough equations in the sequential subset, eg. those relating to garbage collection. 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.

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 $(\lambda H_2, \sigma_2, \Delta_2) \xrightarrow{\star} (H_2', \sigma_2', \Delta_2')$ 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 $(\lambda H_1, \sigma_1, \Delta_1) \xrightarrow{\star} (H_1', \sigma_1', \Delta_1')$ such that $(H_2', \sigma_2', \Delta_2') \equiv (H_1', \sigma_1' \Delta_1')$.

Let $C[\cdot]$ be an 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 for two (promotable) expressions e_1, e_2, $e_1 \equiv e_2$ 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 $\equiv$ quantifies over all sequential contexts. The use of environment moves in Definition 6 enables us to prove a congruence property for 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.

```
        when(c) when(d) s ≡ when(c&&d) s
   atomic { s1; atomic s2} ≡ atomic {s1; s2}
async(P){s}; async(Q) {s1} ≡ async(Q){s1}; async(P){s}
  async(P){async(Q){s} s1} ≡ async(Q[here/P]){s}; async(P) {s1}
           finish{s; s1} ≡ finish{s}; finish{s1}
       finish{when(c){s}} ≡ when(c){finish{s}}
      finish async(p) {} ≡ {}
```

Additionally `finish s` is equal to `s` for `s` a `next`, `resume` or `drop` operation.

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 $S_{\texttt{Coord}} = \{\texttt{here}, \texttt{here.next}, \texttt{here.next.next} \ldots\}$. Let s be such that no transition sequence from $([], \sigma, p : ([], s, []))$ leads to an error.

LEMMA 8. *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:* $([],\sigma,p:(s,([],[],[]))) \equiv ([],\sigma,p:(\texttt{trans}(\Theta,s),([],[],[])))$.

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.

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 `finishs` 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 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.

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

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.

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.

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. 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)
15. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA. (2003) 388–403
16. Harris, T., Herlihy, M., Marlow, S., Jones, S.P.: Composable memory transaction. In: SIGPLAN Symposium on Principles and Practice of Parallel Programming. (2005)
17. Saraswat, V.: Report on the Experimental Language X10, v0.41. Technical report, IBM Research (2005)
18. 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.
19. Hansen, P.B.: Structured multiprogramming. CACM **15** (1972)
20. Hoare, C.: Monitors: An operating system structuring concept. CACM **17** (1974) 549–557
21. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems **13** (1991) 124–149
22. 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
23. 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
24. G.M. Bierman, M.J. Parkinson, Pitts, A.: MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory (2004)
25. : (The java grande forum benchmark suite) www.epcc.ed.ac.uk/javagrande/javag.html.
26. 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
27. Saraswat, V.: Concurrent Constraint Programming. Doctoral Dissertation Award and Logic Programming. MIT Press (1993)