1 Changes, Version 1 (Oct 4 2010)

This document summarizes the main changes between X10 2.0.6 and X10 2.1. The descriptions are intended to be suggestive rather than definitive; see the language specification – when it is finished – for full details.

1.1 Object Model

1. Objects are now local rather than global.

   (a) The `home` property is gone.

   (b) `at(P)S` produces deep copies of all objects located `here` when it executes `S`. (**Warning:** They are copied even in `at(here)S`.)

2. The `GlobalRef[T]` struct is the only way to produce or manipulate cross-place references.

   (a) `GlobalRef`'s have a `home` property.

   (b) Use `GlobalRef[Foo](foo)` to make a new global reference.

   (c) Use `myGlobalRef()` to access the object referenced; this requires `here == myGlobalRef.home`.

3. All those cursed `!`s in types are gone.

4. `global` modifiers are now gone:

   (a) `global` methods in *interfaces* are now the default.

(b) `global` *fields* are gone. In some cases object copying will produce the same effect as global fields. In other cases code must be rewritten. It may be desirable to mark nonglobal fields `transient` in many cases.

(c) `global` *methods* are now marked `@Global` instead. Methods intended to be non-global may be marked `@Pinned`.

1.2 Constructors

1. `proto` types are gone.
2. Constructors and the methods they call must satisfy a number of static checks.
   (a) Constructors can only invoke `private` or `final` methods, or methods annotated `@NonEscaping("v1,v2")`.
   (b) Methods invoked by constructors cannot read fields before they are written.
   (c) The compiler ensures this with a detailed protocol.
3. It is still impossible for X10 constructors to leak references to `this` or observe uninitialized fields of an object. Now, however, the mechanisms enforcing this are less obtrusive than in 2.0.6; the burden is largely on the compiler, not the programmer.

1.3 Implicit clocks for each finish

Most clock operations can be accomplished using the new implicit clocks.

1. A `finish` may be qualified with `clocked`, which gives it a clock.
2. An `async` in a `clocked finish` may be marked `clocked`. This registers it on the same clock as the enclosing `finish`.
3. `clocked async S` and `clocked finish S` may use `next` in the body of `S` to advance the clock.

4. When the body of a `clocked finish` completes, the `clocked finish` is dropped form the clock. It will still wait for spawned asyncs to terminate, but such asyncs need to wait for it.

1.4 Asynchronous initialization of val

`val`s can be initialized asynchronously. As always with `val`s, they can only be read after is guaranteed that they have been initialized. For example, both of the `print`s below are good. However, the commented-out `print` in the `async` is bad, since it is possible that it will be executed before the initialization of `a`.

```
val a:Int;
finish {
  async {
     a = 1;
     print("a=" + a);
  }
  async {
     // WRONG: print("a=" + a);
  }
}
print("a=" + a);
```

1.5 Main Method

The signature for the `main` method is now:

```
def main(Array[String](1)) {..}
```

or, if the arguments are actually used,

```
def main(argv: Array[String](1)) {..}
```

1.6 Assorted Changes

1. The syntax for destructuring a point now uses brackets rather than braces: `for( [i] in 1..10 )`, rather than the prior `(i)`.

1.7 Safety of atomic and when blocks

1. Static effect annotations (`safe`, `sequential`, `nonblocking`, `pinned`) are no longer used. They have been replaced by dynamic checks.

2. Using an inappropriate operation in the scope of an `atomic` or `when` construct will throw `IllegalOperationException`. The following are inappropriate:

   - `when`
   - `resume()` or `next` on clocks
   - async
   - `Future.make()`, or `Future.force()`.
   - `at`

1.8 Removed Topics

The following are gone:

1. `foreach` is gone.

2. All `var`s are effectively `shared`, so `shared` is gone.

3. The place clause on `async` is gone. `async (P) S` should be written `at(P) async S`.

4. Checked exceptions are gone.

5. `future` is gone.

6. `await ... or ...` is gone.

7. `const` is gone.

1.9 Deprecated

The following constructs are still available, but are likely to be replaced in a future version:

1. `ValRail`.