Types for X10 Clocks

Francisco Martins
LaSIGE & University of Lisbon, Portugal
fmartins@di.fc.ul.pt

Vasco T. Vasconcelos
LaSIGE & University of Lisbon, Portugal
vv@di.fc.ul.pt

Tiago Cogumbreiro
LaSIGE & University of Lisbon, Portugal
cogumbreiro@di.fc.ul.pt

Abstract

X10 is a modern language built from the ground up to handle future parallel systems, from multicore machines to cluster configurations. We take a closer look at a pair of synchronisation mechanism: **finish** waits for the termination of parallel computations, and clocks allow multiple concurrent activities to wait for each other at certain points in time. In order to better understand these concepts we study a type system for a stripped down version of X10. The main result is a safety property for typable programs. The study will open, we hope, doors to a more flexible utilisation of the clocks constructs in the X10 language.

1 Introduction

New high-level concurrency primitives are needed more than ever, now that multicore machines lay on our desks and laps. One such primitive is *clocks*, a generalization of barriers introduced in the X10 programming language [1]. Clocks allow multiple concurrent activities to synchronise on a sequence of points in time. Another primitive is **finish** that causes an activity to block (*i.e.,* to suspend its execution) until all its sub-activities have completed.

Even though the language specification [4] provides a clear, plain English, description of the intended semantics (and properties) of the language, and a formalisation of the semantics [5] allows to prove a deadlock freedom theorem, we decided to investigate a simpler setting in which similar results could be obtained. The aim is not only to obtain a progress property for typable programs based on a simple type system, but also to hopefully provide for clock-safe extensions of the X10 language itself.

Towards this end, we have stripped X10 from most of its features, ending up with a simple concurrent language equipped with **finish** and the full functionality of X10 clocks, which we call "X10 restricted to clocks," $\mathsf{X10}\,|_{\mathsf{clocks}}$ for short. For this language we have devised a simple operational semantics with thread (or activity as called in X10) local and global views of (heap allocated) clocks. We have also crafted a simple type system, based on singleton types, drawing expertise from previous work on low-level programming languages [8]. Typable programs are exempt from clock related errors, as reported in the specification of the language [4]. We conjecture that typable programs enjoy a progress property.

There seems to be no formal account of clocks in the X10 language available on the literature. Saraswat and Jagadeesan study a bisimulation for X10 allowing to establish that programs do not deadlock, under certain conditions [5]. Lee and Palsberg present a core language for X10 (no clocks) suited for inter-procedural analysis through type inference [3].

In summary, the contributions of this work are a) a simple operational semantics for activities, **finish**, and clocks that allows to better understand these constructs, b) a simple type systems allowing to prove safety and progress properties (alternative to the constraint-based system [5]), and c) the promise of a more flexible utilization of the clock constructs. The rest of this paper presents the syntax in Section 2, the (operational) semantics and the notion of run-time errors in Section 3, the type system, examples, and main result in Section 4. We conclude, in Section 6, discussing an alternative model for the semantics and pointing directions for future work.

e ::=	*Expressions*	v ::=	*Values*
v	value	x	variable
| **async** $\vec{v}$ e	fork activity	| ()	unit
| **make**	new clock		
| **drop** v	deregister from v		
| **finish** e	wait to terminate		
| **next**	advance phase		
| **resume** v	ready to advance on v		
| **let** $x = e$ **in** e	local declaration		

Figure 1: Top-level syntax of $\mathsf{X10}|_{\mathsf{clocks}}$

2 Syntax

Object-oriented and type-safe, X10 boasts support for concurrency, parallelism, and distribution. Of particular interest to us is the synchronisation mechanism **finish** that waits for the termination of parallel computations and the *clock* primitive that allows for forcing multiple concurrent activities to wait for each other at certain points in time.

The top-level, or programmer's, language we address, $\mathsf{X10}|_{\mathsf{clocks}}$ (X10 restricted to clocks and **finish**), is a subset of the X10 language, generated by the grammar in Figure 1, and relies on a base set of *variables* ranged over by x. An $\mathsf{X10}|_{\mathsf{clocks}}$ program is an expression e that can operate on activities, clocks, or primitive values (of type **unit**). To construct a program we compose expressions through the standard let construct **let** $x = e$ **in** e', which binds variable x to the result of expression e in the scope of expression e'.

Below we present an example program with the purpose of illustrating the syntax and informally presenting the semantics of the language. The example is composed of three activity: an outermost activity a_1, defined from line 2 to line 16, an inner activity a_2, spawned at line 4, and another inner activity a_3, spawned at line 6 and lasting until line 12. Along the example we make use of the derived expression $e_1;e_2$ that abbreviates **let** $x = e_1$ **in** e_2, where x not free in e_2.

```
1   //$a_1$
2   finish
3     let x = make in (
4       finish async $e_0$; //$a_2$
5       //$a_3$
6       async x (
7         $e_1$;
8         resume x;
9         $e_2$;
10        next;
11        $e_3$;
12        drop x);
13      $e_4$;
14      next;
15      $e_5$;
16      drop x)
```

Activities can be *registered* with zero or more clocks and may share clocks with other activities. A clock can thus count with zero or more different registered activities, which are also called *participants*. When an activity a is registered with clock x, we say that x is a clock *held* by a. Activities may only register themselves with clocks via two different means: when they explicitly create a clock (line 3, **make**, creates a clock and registers activity a_1 with the clock), and when they inherit clocks from its parent activity in a spawning operation (line 6 spawns activity a_3 and registers it with the clock associated with variable x). Expression **drop** deregisters an activity from a clock (line 12, **drop** x deregisters activity a_3 from clock x, whereas line 16 deregisters activity a_1 from x). Activities are disallowed to manipulate clocks they are not registered with.

In X10 $|_{\text{clocks}}$, an activity synchronises via two different methods: by waiting for every activity spawned by expression e to terminate (line 4 only terminates when activity a_2 and all its sub-activities terminate), and by waiting for its held clocks to *advance a phase* (activity a_3 waits at line 10, whereas activity a_1 waits at line 14). X10 distinguishes between *local* and *global* termination of an expression. Local termination of an expression corresponds to concluding its evaluation (reducing to a value v). An expression terminates globally when it terminates locally and each activity spawned by the expression has also terminated globally. Expression **finish** e converts the global termination of expression e into a local termination (line 2 waits for the expression in lines 3–16 to terminate, meaning that it waits as well for the termination of activities a_2 and a_3; line 4 waits for activity a_2 (the result of the evaluation of **async** e_0) to terminate before launching the sub-activity a_3, in line 6). The second method to synchronise activities is using clocks. Groups of activities, defined by the participants of a clock, evaluate concurrently until they reach the end of a phase. When every participant of the group reaches the end of the phase, then all move to the next phase, while still executing concurrently. Phases are delimited by expression **next**; activities evaluate this expression to mark the end of a phase (activities a_1 and a_3 synchronise at lines 10 and 14).

An activity can inform all other participants of a clock x that it has completed its phase by using an expression of the form **resume** x, thus making clocks act as fuzzy barriers [2] (line 8). Expression **resume** can be viewed as an optimisation to diminish contention upon advancing a phase: it allows activities blocked on **next** to cease waiting for such activities (which can become at most one phase behind the clock's phase). In the example, expression e_2 might execute at the same time as expression e_5, since activity a_3 may trigger (at line 6) activity a_1 to advance clock x (blocked at line 12), thus evaluating expressions e_2 and e_5 in parallel. If we omit expression **resume** x from this example, then expressions e_2 and e_5 cannot evaluate in parallel.

Clocks can be thought of as data structures holding (among other information) a natural number representing its *global phase*, initially set to zero. Advancing a clock's phase amounts to incrementing its global phase when every registered activity has *quiesced*; an activity is quiescent on a clock x after performing a **resume** x. An activity resumes all its held clocks together by evaluating **next** and suspends itself until these clocks become ready to advance to the next phase.

3 Operational Semantics

Figure 2 depicts the run-time syntax of our language. The run-time system relies on one additional set, *clocks names* (also used for heap addresses), ranged over by c. A state S of an X10 $|_{\text{clocks}}$ computation comprises a shared heap H and a set of named activities A that run concurrently. Activity names, l, are taken from the set of variables introduced in Section 2. The heap stores clock values h, triples comprising a natural number i representing its global phase, a set R with the **r**egistered activities, and another set Q with the **q**uiesced activities. These sets keep track of the activities that synchronise in the clock (R) and the activities that are ready to advance the clock to the next phase (Q). Set difference $R \setminus Q$ identifies

$V ::= \{c_1 \colon i_1, \dots, c_n \colon i_n\}$	*Clocks' local view*		$e ::= \dots$	*Expressions*
$a ::= (V, e, A)$	*Activity*		$\mid$ **join** l	join activities
$A ::= \{l_1 \colon a_1, \dots, l_n \colon a_n\}$	*Sets of named activities*			
$R, Q ::= \{l_1, \dots, l_n\}$	*Sets of activity names*		$v ::= \dots$	*Values*
$h ::= \langle i, R, Q \rangle$	*Clock values*		$\mid c$	clock
$H ::= \{c_1 \colon h_1, \dots, c_n \colon h_n\}$	*Heaps*			
$S ::= H; A$	*States*			

Figure 2: Run-time syntax of X10 $|_{\text{clocks}}$

the activities yet to make progress on a clock; the clock phase only advances when all activities have quiesced (when $R = Q$). The set of registered activities R also allows to enforce that an activity is only able to operate on registered clocks.

An activity a is composed of a set of clocks' local view V, an expression e under execution, and a set of sub-activities A. Each activity has its own perception of the global phase of a clock; the clocks local view V is a map from clock names to natural numbers describing the local phase. The global phase of a clock and its activity local view may diverge in case an activity issues a **resume** on the clock and this is enough to trigger other participants to advanced the clock's phase. Only when the activity issues a **next**, the local view of the clock and the global phase become in sync. At anytime, clock's local view is at most one phase behind the global phase. Notice that an activity is itself a tree of activities, since each activity holds a set of (named) sub-activities. When evaluating an expression **finish** e, the activity starts sub-activities for evaluating expression e and all activities spawned by e. Otherwise, activities have no sub-activities.

We augment the syntax of expressions at run-time with **join** l. Expression **join** l results from evaluating **finish** e; label l identifies the activity that executes the body e of the **finish** expression and that produces the resulting value of the **finish** e expression.

We present the small step reduction rules for X10 $|_{\text{clocks}}$ in Figures 3 and 4. Reduction for activities (Figure 3), $H; a \rightarrow_l H'; A; a'$, operates on a heap H and an activity a, and produces a possible different heap H', a set A of activities spawned during the evaluation of the expression in a, and a new activity a'.

Rule R-ASYNC is the only rule that affects the set of spawned activities A. The programmer specifies a list of clocks $\vec{c}$ on which the new activity is to be registered with. The newly created activity (named l'), is added to the set R of activities registered with clocks $\vec{c}$, and stored in the heap. The result of spawning an activity is the unit value $()$. The created activity is composed of a clock view holding, for each clock c in $\vec{c}$, a copy of the global phase p, an expression e to be evaluated, and an empty set of sub-activities. The new activity inherits each clock c quiescence property, *i.e.,* if l is quiescent on clock c so is l' ($Q' =$ if $l \in Q$ then $Q \cup \{l'\}$ else Q).

Expression **make** creates a new clock in the heap with initial phase 0, with l as the only registered activity, and with no resumed activities, $\langle 0, \{l\}, \emptyset \rangle$. The activity creating the clock maintains a local clock view $\{c \colon 0\}$ stored in V. Rule R-RESUME asserts that when the l-th activity issues a **resume** c, its label is recorded in the set of resumed activities R if the clock local phase is in sync with the clock global phase ($p = V(c)$); otherwise, the effect of the expression is discarded ($p \neq V(c)$), since the clock has already advance to the next phase. An activity may only perform a **resume** operation per clock phase ($l \notin Q$).

Expression **next** blocks the activity until all clocks have been resumed (C_1) or have already advance their phases (C_2) (rule R-NEXT). Notice that when activities are waiting on a clock c, the clock can be in one of three states: (a) there are non-quiescent activities on the clock and c is neither a member of

$$\frac{\{\vec{c}\} \subseteq \operatorname{dom} V \qquad l' \text{ is fresh} \qquad Q' \triangleq \text{if } l \in Q \text{ then } Q \cup \{l'\} \text{ else } Q}{H\{c\colon \langle p,R,Q\rangle\}_{c \text{ in } \vec{c}}; (V, \mathbf{async}\ \vec{c}\ e, A) \rightarrow_l H\{c\colon \langle p, R\cup\{l'\}, Q'\rangle\}_{c \text{ in } \vec{c}}; \{l'\colon (\{c\colon p\}_{c \text{ in } \vec{c}}, e, \emptyset)\}; (V, (), A')} \quad \text{(R-ASYNC)}$$

$$\frac{c \text{ is fresh}}{H; (V, \mathbf{make}, A') \rightarrow_l H\{c\colon \langle 0, \{l\}, \emptyset\rangle\}; \emptyset; (V\{c\colon 0\}, c, A')} \quad \text{(R-MAKE)}$$

$$\frac{Q' \triangleq \text{if } p = V(c) \text{ then } Q \cup \{l\} \text{ else } Q \qquad l \notin Q}{H\{c\colon \langle p,R,Q\rangle\}; (V, \mathbf{resume}\ c, A) \rightarrow_l H\{c\colon \langle p,R,Q'\rangle\}; \emptyset; (V, (), A)} \quad \text{(R-RESUME)}$$

$$\frac{\begin{array}{c} C_1 \triangleq \{c \mid V(c) = p, H(c) = \langle p,R,R\rangle\} \\ C_2 \triangleq \{c \mid V(c) = p, H(c) = \langle p+1, \_, \_\rangle\} \qquad C_1 \cup C_2 = \operatorname{dom} V \end{array}}{H; (V, \mathbf{next}, A) \rightarrow_l H\{c\colon \langle p+1, R, \emptyset\rangle\}_{c \in C_1}; \emptyset; (\{c\colon V(c)+1\}_{c \in V}, (), A)} \quad \text{(R-NEXT)}$$

$$\frac{c \in \operatorname{dom} V \qquad H' \triangleq \text{if } R = \{l\} \text{ then } H \setminus \{c\} \text{ else } H\{c\colon \langle p, R\setminus\{l\}, Q\setminus\{l\}\rangle\}}{H\{c\colon \langle p,R,Q\rangle\}; (V, \mathbf{drop}\ c, A) \rightarrow_l H'; \emptyset; (V \setminus \{c\}, (), A)} \quad \text{(R-DROP)}$$

$$\frac{l_0 \text{ is fresh}}{H; (V, \mathbf{finish}\, e, A) \rightarrow_l H; \emptyset; (V, \mathbf{join}\ l_0, A\{l_0\colon (V, e, \emptyset)\})} \quad \text{(R-FINISH)}$$

$$H; (V, \mathbf{join}\ l_0, \{l_0\colon (\emptyset, v_0, \emptyset), \ldots, l_n\colon (\emptyset, v_n, \emptyset)\}) \rightarrow_l H; \emptyset; (\emptyset, v_0, \emptyset) \quad \text{(R-JOIN)}$$

Figure 3: Reduction rules for activities $\boxed{H; a \rightarrow_l H; A; a}$

$$H; A\{l\colon (V, \mathbf{let}\ x = v\ \mathbf{in}\ e, A')\} \rightarrow H; A\{l\colon (V, e[v/x], A')\} \quad \text{(R-LET-VAL)}$$

$$\frac{l \in \operatorname{dom} H \qquad H; (V, e, A) \rightarrow_l H'; A'''; (V', e', A')}{H; A''\{l\colon (V, \mathbf{let}\ x = e\ \mathbf{in}\ e'', A)\} \rightarrow H'; A''\{l\colon (V', \mathbf{let}\ x = e'\ \mathbf{in}\ e'', A')\}, A'''} \quad \text{(R-LET)}$$

$$\frac{H; A' \rightarrow H'; A''}{H; A\{l\colon (V, e, A')\} \rightarrow H'; A\{l\colon (V, e, A'')\}} \quad \text{(R-ACTIVITY)}$$

Figure 4: Reduction rules for states $\boxed{H; A \rightarrow H; A}$

C_1 nor of C_2; (b) all registered activities are quiescent on the clock, and so c is a member of C_1; (c) the clock has advanced to the next phase thus becoming a member of C_2. When an activity advances a clock global phase, it stops being a member of set C_1 and becomes a member of set C_2 for the remaining activities waiting on that clock. Since rule R-NEXT only updates the clock phase of those belonging to C_1 ($H\{c\colon \langle p+1, R, \emptyset\rangle\}_{c \in C_1}$) it ensures that the global clock state is updated only once. With expression **drop** c, the l-th activity cedes its control over clock c: we remove c from clock view V, and remove activity identifier l from both sets R and Q. Two consequences of dropping a clock c are: a) activities waiting on clock c are no longer blocked because of this activity; b) when executing a **next** expression, this activity no longer waits for clock c. In case l is the only activity registered with clock c, it is safe to garbage collect the clock. Expression **finish** e creates a child activity and evaluates into expression **join** l_0 (rule R-FINISH), which in turn blocks while there exists sub-activities running. When all sub-activities have reduced to a value, activity l (**join** l_0) evaluates into the value in its sub-activity l_0 and garbage collects all other sub-activities (rule R-JOIN).

The reduction for states (Figure 4), $S \rightarrow S'$, allows for non-deterministic choice of which activity l to evaluate (rule R-ACTIVITY), capturing the concurrency present in X10 computations. We evaluate the let binding from left-to-right (rule R-LET), when the left-hand-side expression becomes a value, we substitute this value for variable x in the continuation expression e (rule R-LET-VAL).

Recall the example from Section 2. Consider some loading function that sets up the initial state from a given expression, which in this case is an empty heap and an activity evaluating the code in the example under a dummy **let**.

$$S_0 = \emptyset; \{l_1 : (0, \textbf{let } z = \underbrace{\textbf{finish let } x = \textbf{make in } (\textbf{finish } (\textbf{async } e_0); e_6)}_{\text{Example from Section 2}} \textbf{ in } (), \emptyset)\}$$

where e_6 is $(\textbf{async } x\ (e_1; \textbf{resume } x; e_2; \textbf{next}; e_3; \textbf{drop } x)); e_4; \textbf{next}; e_5; \textbf{drop } x$. We perform two reduction steps to illustrate the effect of expression **finish** on the sub-activities of l_1, corresponding to the main activity.

$$\begin{aligned}
& \emptyset; \{l_1 : (\emptyset, \textbf{let } z = \textbf{finish let } x = \textbf{make in } (\textbf{finish } (\textbf{async } e_0); e_6) \textbf{ in } z, \emptyset)\} \\
(\text{R-FINISH}, \text{R-LET}) \rightarrow & \emptyset; \{l_1 : (\emptyset, \textbf{let } z = \textbf{join } l_2 \textbf{ in } (), \\
& \quad \{l_2 : (\emptyset, \textbf{let } x = \textbf{make in } (\textbf{finish } (\textbf{async } e_0); e_6), \emptyset)\})\}
\end{aligned}$$

From this state on, while **join** remains blocked, we apply rule R-ACTIVITY to evaluate the child activities of l_1. We perform three more reduction steps and observe how expression **make** updates the heap and the clock view of activity l_2.

$$\begin{aligned}
& \emptyset; \{l_1 : (\emptyset, \textbf{let } z = \textbf{join } l_2 \textbf{ in } (), \\
& \quad \{l_2 : (\emptyset, \textbf{let } x = \textbf{make in } (\textbf{finish } (\textbf{async } e_0); e_6), \emptyset)\})\} \\
(\text{R-ACTIVITY}, \text{R-LET}, \text{R-MAKE}) \rightarrow & \{c : \langle 0, \{l_2\}, \emptyset \rangle\}; \\
& \{l_1 : (\emptyset, \textbf{join } l_2, \\
& \quad \{l_2 : (\{c : 0\}, \textbf{let } x = c \textbf{ in } (\textbf{finish } (\textbf{async } e_0); e_6), \emptyset)\})\}
\end{aligned}$$

The non-determinism of our semantics allows for various different reductions. A possible reduction composed of a heap and the sub-activities of activity l_1 is

$$\begin{aligned}
{}^{\star} \rightarrow & \{c : \langle 0, \{l_2, l_3\}, \emptyset \rangle\}; \\
& \{l_1 : (\emptyset, \textbf{join } l_2, \{l_2 : (\{c : 0\}, (\textbf{next}; e_5; \textbf{drop } c), \emptyset), \\
& \quad l_3 : (\{c : 0\}, (\textbf{resume } c; e_2; \textbf{next}; e_3; \textbf{drop } c), \emptyset)\})\}
\end{aligned}$$

We now illustrate the case when activities l_2 and l_3 are evaluating expressions e_2 and e_5 concurrently.

$$\begin{aligned}
(\text{R-NEXT}, \text{R-LET}) \rightarrow & \{c : \langle 0, \{l_2, l_3\}, \{l_2\} \rangle\}; \\
& \{l_1 : (\emptyset, \textbf{join } l_2, \{l_2 : (\{c : 0\}, (\textbf{next}; e_5; \textbf{drop } c), \emptyset), \\
& \quad l_3 : (\{c : 0\}, (\textbf{resume } c; e_2; \textbf{next}; e_3; \textbf{drop } c), \emptyset)\})\} \\
(\text{R-RESUME}, \text{R-LET-VAL}) \rightarrow & \{c : \langle 0, \{l_2, l_3\}, \{l_2, l_3\} \rangle\}; \\
& \{l_1 : (\emptyset, \textbf{join } l_2, \{l_2 : (\{c : 0\}, (\textbf{next}; e_5; \textbf{drop } c), \emptyset), \\
& \quad l_3 : (\{c : 0\}, (e_2; \textbf{next}; e_3; \textbf{drop } c), \emptyset)\})\} \\
(\text{R-NEXT}, \text{R-LET}, \text{R-LET-VAL}) \rightarrow & \{c : \langle 1, \{l_2, l_3\}, \emptyset \rangle\}; \\
& \{l_1 : (\emptyset, \textbf{join } l_2, \{l_2 : (\{c : 1\}, (e_5; \textbf{drop } c), \emptyset), \\
& \quad l_3 : (\{c : 0\}, (e_2; \textbf{next}; e_3; \textbf{drop } c), \emptyset)\})\}
\end{aligned}$$

$$H; A\{l\colon (V, \textbf{let } x = \textbf{async } \vec{c}\ e \textbf{ in } e', A')\} \in \text{Error} \quad \text{if } c \not\subseteq \operatorname{dom} H \text{ or } c \notin \operatorname{dom} V \qquad \text{(E-ASYNC)}$$

$$H\{c\colon \langle \_, \_, Q\rangle\}; A\{l\colon (V, \textbf{let } x = \textbf{resume } c \textbf{ in } e, A')\} \in \text{Error} \quad \text{if } l \in Q \text{ or } c \notin \operatorname{dom} H \text{ or } c \notin \operatorname{dom} V \qquad \text{(E-RESUME)}$$

$$H; A\{l\colon (V, \textbf{let } x = \textbf{drop } c \textbf{ in } e, A')\} \in \text{Error} \quad \text{if } c \notin \operatorname{dom} H \text{ or } c \notin \operatorname{dom} V \qquad \text{(E-DROP)}$$

$$H; A\{l\colon (V, \textbf{let } x = \textbf{next in } e, A')\} \in \text{Error} \quad \text{if } V(c) = p, H(c) = (p, \_, Q), \text{ and } l \notin Q, \text{ for some } c \qquad \text{(E-NEXT)}$$

$$H; A\{l\colon (V, v, \_)\} \in \text{Error} \quad \text{if } V \neq \emptyset \qquad \text{(E-ACT)}$$

$$\frac{H; A' \in \text{Error}}{H; A\{l\colon (\_, \_, A')\} \in \text{Error}} \qquad \text{(E-ACT-SET)}$$

Figure 5: Run-time errors

$$\tau ::= \textbf{unit} \mid \textbf{clock}(\alpha) \qquad \textit{Types}$$

Figure 6: Syntax of types

After activity l_3 evaluates **resume** c, activity l_2, which is blocked evaluating **next**, progresses, thus allowing expressions e_2 and e_5 to execute in parallel. Notice that activity l_2 remains in phase 0, while activity l_3 is in phase 1.

Run-time errors is the smallest set $\in$ Error of states generated by the rules in Figure 5 and is consistent with all the conditions documented to raise exception `ClockUseException`, as discussed in the X10 language specification report [4]. The type system we present in Section 4 allow us to reject, at compile time, programs that could throw a `ClockUseException`; we assert that well typed programs manipulate solely registered clocks.

During an **async** operation, an activity cannot transmit unregistered clocks through its first argument (rule E-ASYNC). Similarly, activities can only perform **resume** or **drop** operations on clocks they are registered with (rules E-RESUME and E-DROP). In particular, it constitutes an error for an activity to drop a clock twice, or to resume a clock more than once (for the same phase) or after dropping it. We achieved a fine grained control over the clocks an activity is registered with. Specifically, we are able to devise, at compile time, whether an activity resumed or dropped all of its held clocks. Therefore, it constitutes an error when an activity evaluates a **next** expression before resuming all its clocks (rule E-NEXT), as well as when an activity evaluates to a value without dropping all its clocks (rule E-ACT). Rule E-ACT-SET allows error propagation under sub-activities.

4 Type System

For types we rely on an additional base set of *singleton types* ranged over by α. The syntax of types depicted in Figure 6 introduces of the unit value type (**unit**) and the clock type (**clock**(α)). We assign a different type (singleton type α) to each clock in order to track clock usage throughout the program.

The type system for $\mathsf{X10}|_{\mathsf{clocks}}$ programs is defined in Figures 7 and 8. A typing Γ is a map from variables (or activity labels) and clocks to types. We write $\operatorname{dom}\Gamma$ for the domain of Γ. When $x \notin \operatorname{dom}\Gamma$

$$\mathscr{R},\alpha \vdash \mathbf{clock}(\alpha) \qquad \mathscr{R} \vdash \mathbf{unit} \qquad \text{(T-WF-C, T-WF-U)}$$

$$\frac{\mathscr{R} \vdash \tau}{\Gamma,x\colon \tau;\mathscr{R} \vdash x\colon \tau} \qquad \Gamma,c\colon \mathbf{clock}(\alpha);\mathscr{R},\alpha \vdash c\colon \mathbf{clock}(\alpha) \qquad \Gamma;\mathscr{R} \vdash ()\colon \mathbf{unit}$$

$$\text{(T-VAR, T-CLOCK-REF, T-UNIT)}$$

$$\frac{\Gamma;\mathscr{R} \vdash v_1\colon \mathbf{clock}(\alpha_1) \quad \cdots \quad \Gamma;\mathscr{R} \vdash v_n\colon \mathbf{clock}(\alpha_n) \qquad \alpha_i \neq \alpha_j,\ \text{if } i \neq j \qquad \alpha_i \text{ not in } \Gamma}{\Gamma;\mathscr{R} \vdash v_1 \ldots v_n\colon \{\alpha_1,\ldots,\alpha_n\}}$$

$$\text{(T-CLOCK-SEQ)}$$

Figure 7: Typing rules for values and for well-formed types

$$\frac{\Gamma;\mathscr{R} \vdash v\colon \tau}{\Gamma;\mathscr{R};\mathscr{Q} \vdash v\colon (\tau,\mathscr{R},\mathscr{Q})} \qquad \frac{\alpha \text{ is fresh}}{\Gamma;\mathscr{R};\mathscr{Q} \vdash \mathbf{make}\colon (\mathbf{clock}(\alpha),\mathscr{R}\cup\{\alpha\},\mathscr{Q})} \qquad \text{(T-VALUE, T-MAKE)}$$

$$\frac{\Gamma;\mathscr{R} \vdash v\colon \mathbf{clock}(\alpha) \qquad \alpha \notin \mathscr{Q}}{\Gamma;\mathscr{R};\mathscr{Q} \vdash \mathbf{resume}\ v\colon (\mathbf{unit},\mathscr{R};\mathscr{Q}\cup\{\alpha\})} \qquad \frac{\Gamma;\mathscr{R} \vdash v\colon \mathbf{clock}(\alpha)}{\Gamma;\mathscr{R};\mathscr{Q} \vdash \mathbf{drop}\ v\colon (\mathbf{unit},\mathscr{R}\setminus\{\alpha\},\mathscr{Q}\setminus\{\alpha\})}$$

$$\text{(T-RESUME, T-DROP)}$$

$$\frac{\Gamma;\mathscr{R} \vdash \vec{v}\colon \mathscr{R}' \qquad \Gamma;\mathscr{R}';\mathscr{Q}\cap\mathscr{R}' \vdash e\colon (\_,\emptyset,\emptyset)}{\Gamma;\mathscr{R};\mathscr{Q} \vdash \mathbf{async}\ \vec{v}\ e\colon (\mathbf{unit},\mathscr{R};\mathscr{Q})} \qquad \Gamma;\mathscr{R};\mathscr{R} \vdash \mathbf{next}\colon (\mathbf{unit},\mathscr{R};\emptyset) \quad \text{(T-ASYNC,T-NEXT)}$$

$$\frac{\Gamma;\emptyset;\emptyset \vdash e\colon (\tau,\emptyset,\emptyset)}{\Gamma;\mathscr{R};\mathscr{Q} \vdash \mathbf{finish}\, e\colon (\tau,\mathscr{R};\mathscr{Q})} \qquad \text{(T-FINISH)}$$

$$\frac{\Gamma;\mathscr{R};\mathscr{Q} \vdash e_1\colon (\tau,\mathscr{R}',\mathscr{Q}') \qquad \Gamma,x\colon \tau;\mathscr{R}';\mathscr{Q}' \vdash e_2\colon (\tau',\mathscr{R}'',\mathscr{Q}'')}{\Gamma;\mathscr{R};\mathscr{Q} \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\colon (\tau',\mathscr{R}'',\mathscr{Q}'')} \qquad \text{(T-LET)}$$

Figure 8: Typing rules for expressions

we write $\Gamma,x\colon \tau$ for the typing Γ' such that $\operatorname{dom}\Gamma' = \operatorname{dom}\Gamma \cup \{x\}$, $\Gamma'(x) = \tau$, and $\Gamma'(y) = \Gamma(y)$ for $y \neq x$. The type system also uses sets of singleton types, ranged over by $\mathscr{R}$, for **r**egistered clocks, and $\mathscr{Q}$, for res**u**med clocks.

The typing rules for values and for well formed types (Figure 7) are simple to follow. Well-formedness for clock types (rule T-WF-C) ensures that activities only make use of clocks they are registered with. Rule T-CLOCK-SEQ ensures that clock lists (as those in the heap) have distinct singleton clock types assigned to them, a property that is crucial for establishing type safety. For typing expressions we use a type system (Figure 8) that records the changes made to the set of registered clocks, either by creating or dropping clocks, and to the set of quiescent clocks (using **resume** and **next**) of an expression. Typing judgements are of the form $\Gamma;\mathscr{R};\mathscr{Q} \vdash e\colon (\tau,\mathscr{R}',\mathscr{Q}')$ meaning that expression e is well typed assuming the types for the free identifiers in Γ, the registered clocks in $\mathscr{R}$, and the quiescent clocks in $\mathscr{Q}$. The type of an expression is a triple recording its type τ, the sets registered ($\mathscr{R}'$) and quiescent ($\mathscr{Q}'$) after execution of the expression.

Most typing rules are straightforward. When creating a clock (rule T-MAKE) we associate a new singleton type α with the clock and include it in set of clocks registered by the activity ($\mathscr{R}\cup\{\alpha\}$). Rule T-RESUME, which asserts that activities can only **resume** a clock α in the set of registered clocks $\mathscr{R}$ (*vide* rule T-VAR), marks clock α as quiescent. Notice that a clock cannot be resumed more than once for the same phase ($\alpha \notin \mathscr{R}$). A **drop** v expression removes clock v from both the sets $\mathscr{R}$ and $\mathscr{Q}$, thus the clock cannot be passed to new activities, be the target of a **resume** expression, or be dropped again.

Rule T-ASYNC asserts that when an activity spawns another activity registered on a sequence of clocks, the quiescent property of the clocks is preserved by propagating the information about the quiescent clock α ($\mathscr{Q} \cap \mathscr{R}$). Moreover, the new activity must have dropped all its clocks upon termination. Expression **next** marks the end of a phase; it checks that all clocks have been resumed and clears the quiescent clocks for the new phase (rules T-NEXT).

The **finish** construct may interfere with clocks and cause programs to deadlock. In order to avoid such situations we prevent the body of a **finish** e expression (e) from accessing any clock already defined, thus eliminating (nested) dependencies between clocks and **finish**. Rule T-FINISH also forces e to unregister from all clocks it has created, and therefore **finish** e has no effect on registered and quiescent clocks. Refer to the examples below for further discussion on the deadlock problem. When typing a **let** expression (rule T-LET), its continuation e_2 is typed taking into consideration the effects produced by expression e_1. The type of the **let** are that of e_2, as usual.

We have deliberately deviated from the standard X10 semantics in three cases: **next**, **drop**, and **resume**. The reasons for such deviation are: (a) to illustrate the power of singleton types in keeping track of clocks, (b) to simplify the (operational and static) semantics, (c) to enforce a programming discipline that may avoid potential bugs, and (d) because the compiler has enough information to suggest code fixes (*e.g.*, by enumerating the clocks that need to be dropped before a **next**; see examples below).

Next we discuss some $\mathsf{X10}|_{\mathsf{clocks}}$ programs and the semantic guarantees enforced by the type system. Our first example concerns clock aliasing. The report on X10 [4] read until recently "All clock variables are implicitly final. The initializer for a local variable declaration of type Clock must be a new clock expression. Thus X10 does not permit aliasing of clocks." Clearly a type system with linear control like the one we present allows to relieve such a restriction. The following example is not typable in X10.

```
let x = make in
let y = x in (
  async y (resume x; drop y);
  drop x)
```

In our case the code is typable, assigning the same singleton type $\mathbf{clock}(\alpha)$ to both x and y.

Our second example deals with the so called *live clock condition*. Apart from the errors in Figure 5, X10 identifies another source of problems: allowing an activity to transmit a clock that has been resumed to a spawned activity (with **async**). Our operational semantics allows the forked activity to inherit the "status" (resumed/not resumed) of the parent activity (*vide* rule R-ASYNC) and therefore preserve the quiescence property of clocks and avoid a race condition on the clock. The following example is not typable in X10 [4].

```
let x = make in (
  resume x;
  async x (resume x; next; drop x);
  drop x)
```

Below we describe a race condition triggered by clock synchronisation. Activity a_1 creates a clock x, starts a second activity a_2 registered with clock x that, in turn, resumes on x and starts a third activity a_3 also registered with x.

```
// activity $a_1$
let x = make in (
  async x ( // activity $a_2$
        $e_1$;
        resume x;
        $e_2$;
```

```
 7          async x (  // activity  a3
 8            e3 ;
 9            next;
10            drop x );
11          next;
12          drop x );
13    e4 ;
14    next;
15    drop x)
```

The race condition might occur because after a_2 resumes on x (line 5), either activity a_1 may advance clock x phase by executing **next** (line 14) or a_2 may register a new activity a_3 with x (line 7), blocking a_1 until activity a_3 executes its **next** instruction (line 9). By inheriting the resume status of clock x, activity a_3 does not block activity a_1 and the race condition disappears (*vide* rule R-ASYNC in Figure 3 and rule T-ASYNC in Figure 8).

The next example deals with resuming after resuming, a pattern accepted in X10. Rule T-RESUME rejects the program, since it is able to determine that clock x is resumed twice.

```
let  x = make in (
  resume x;
  resume x)
```

Unlike X10, we have decided to explicitly deregister activities from clocks upon activity termination. Our type system keeps track of the clocks an activity is registered with, and rejects programs with activities that finish before deregistering from all its clocks. Clocks without registered activities can be safely garbage collected (*vide* rule R-DROP). The following example fails to type check, since the launched activity does not drop clock x. Although, the compiler may suggest an appropriate fix, in this case adding a **drop** x after the **next** instruction on line 2.

```
1    let  x = make in (
2      async x  (resume x; next);
3      drop x)
```

Finally, we discuss the interplay among **finish**, **async**, and clocks, which may cause programs to deadlock. The following program deadlocks because activity a_2 is waiting on **next** (line 4) for activity a_1 to advance on x, which is planned to occur at line 6, but a_1 is waiting on **finish** (line 3) for activity a_2 to terminate, so a_1 never reaches line 6 and the program deadlocks.

```
1    //a1
2    let  x = make in (
3       finish
4         async x  (resume x; next;  drop x);  //a2
5      resume x;
6      next;
7      drop x)
```

The cause for deadlock is that activity a_2 is registered with a clock that is defined outside the enclosing **finish**: clock x is defined in line 2, whereas the **finish** expression extends from line 3 to line 4. Our type system rejects this program, because when typing a **finish** e expression we type check e in an environment with no registered clocks (*vide* rule R-FINISH).

$$\frac{c \vdash A\colon S_1 \qquad c \vdash A'\colon S_2}{c \vdash A, \{l\colon (V, \_, A')\}\colon S_1 \cup S_2 \cup \operatorname{dom} V \cap \{c\}} \qquad c \vdash \emptyset\colon \emptyset$$

(WF-ACT-CLOCK, WF-ACT-CLOCK-E)

$$\frac{Q \subseteq S \subseteq \operatorname{dom}\Gamma \qquad c \vdash A\colon S \qquad \Gamma; A \vdash H\colon \diamond}{\Gamma; A \vdash H, \{c\colon \langle \_, S, Q\rangle\}\colon \diamond} \qquad \Gamma; A \vdash \emptyset\colon \diamond \qquad \text{(WF-HEAP, WF-HEAP-E)}$$

$$\frac{H(c_i) = \langle \_, \mathscr{R}_i, \_\rangle \qquad l \in \mathscr{R}_i \qquad H \vdash A\colon \diamond \qquad H \vdash A'\colon \diamond}{H \vdash A, \{l\colon (\{c_1\colon \_, \dots, c_n\colon \_\}, \_, A')\}\colon \diamond} \qquad H \vdash \emptyset\colon \diamond$$

(WF-ACT-SET, WF-ACT-SET-E)

$$\frac{H \vdash A\colon \diamond \qquad \Gamma; A \vdash H\colon \diamond}{\Gamma \vdash H; A\colon \diamond} \qquad \text{(WF-STATE)}$$

Figure 9: Well-formed states

$$\Gamma, l\colon \tau; \mathscr{R}; \mathscr{Q} \vdash \textbf{join}\ l\colon (\tau, \mathscr{R}, \mathscr{Q}) \qquad \text{(T-JOIN)}$$

Figure 10: Typing rules for run-time expressions

5 Main results

This section is dedicated to the study of the main result of our system, namely typing preservation and type safety for typable programs.

We are only concerned with well-formed states (Figure 9). A state is well formed if for each clock, the set of registered activities with the clock contains exactly those activities that can manipulate it. State

$$\{c : \langle \_, \emptyset, \_\rangle\}; \{l\colon (\{c\colon \_\}, \textbf{let}\ \_ = \textbf{next in}\ \_, \_)\}$$

is ill formed, since activity l uses clock c and is not registered with c. The activity is able to advance c's phase without becoming quiescent on c. State

$$\{c : \langle \_, \{l, l', \dots\}, \_\rangle\}; \{l\colon (\emptyset, \_, \_), l'\colon (\{c\colon \_\}, \_, \_), \dots\}$$

is also ill formed, since activity l is mentioned as registered with clock c and is not part of l's local view (which is $\emptyset$). Any other activity registered with c (l' in the example) is bound to deadlock because l will never quiesce on c.

The typing rules for run-time expression **join** l and for machine states and activities is depicted in Figures 10 and 11. The type of a **join** l expression (rule T-JOIN) is that of activity l. Notice that **join** l is the result of evaluating a **finish** e expression (rule R-FINISH) that is, in fact, the type of e (rule T-FINISH). It is worth noticing that heap H is well typed if H each clock is assigned to a different singleton type and if the clocks allocated in the heap are exactly those of typing Γ (rule T-HEAP where C is the set of all clocks). Moreover, activities may only resume on registered clocks. An activity (V, e, A) has the type of its expression e (rule T-ACT), which must unregister from all its clocks before terminating, since after evaluating e it is expected that the set of registers clocks should be empty. Rule T-STATE incorporates the definition of well-formed states into the type system. The remaining typing rules should be easy to follow.

Lemma 1 (Weakening)**.** *Let a be a variable or a clock name.*

$$\frac{\Gamma;\mathscr{R} \vdash c_1 \ldots c_n \colon \mathscr{R}}{\Gamma \vdash \{c_1\colon \_, \ldots, c_n\colon \_\}\colon \mathscr{R}} \quad \text{(T-VIEW)}$$

$$\frac{\Gamma \vdash V\colon \mathscr{R} \qquad \mathscr{Q} \subseteq \mathscr{R} \qquad \Gamma;\mathscr{R};\mathscr{Q} \vdash e\colon (\tau, \emptyset, \emptyset) \qquad \Gamma \vdash A}{\Gamma \vdash (V, e, A)\colon \tau} \quad \text{(T-ACT)}$$

$$\frac{\Gamma \vdash a_1\colon \tau_1 \quad \cdots \quad \Gamma \vdash a_n\colon \tau_n}{\Gamma, l_1\colon \tau_1, \ldots, l_n\colon \tau_n \vdash \{l_1\colon a_1, \ldots, l_n\colon a_n\}} \quad \text{(T-ACT-SET)}$$

$$\frac{\Gamma;\mathscr{R} \vdash c_1 \ldots c_n\colon \mathscr{R} \qquad \{c_1, \ldots, c_n\} = \operatorname{dom}\Gamma|_{\mathbf{C}}}{\Gamma \vdash \{c_1\colon h_1, \ldots, c_n\colon h_n\}} \quad \text{(T-HEAP)}$$

$$\frac{\Gamma \vdash H;A\colon \diamond \qquad \Gamma \vdash H \qquad \Gamma \vdash A}{\Gamma \vdash H;A} \quad \text{(T-STATE)}$$

Figure 11: Typing rules for machine states

1. *If* $H \vdash A\colon \diamond$ *then* $H, \{c\colon h\} \vdash A\colon \diamond$.
2. *If* $\Gamma;\mathscr{R} \vdash v\colon \tau$ *then* $\Gamma, a\colon \tau' \vdash v\colon \tau$.
3. *If* $\Gamma \vdash V\colon \mathscr{R}$ *then* $\Gamma, a\colon \tau \vdash V\colon \mathscr{R}$.
4. *If* $\Gamma;\mathscr{R};\mathscr{Q} \vdash e\colon T$ *then* $\Gamma, a\colon \tau;\mathscr{R};\mathscr{Q} \vdash e\colon T$.

Proof outline.

1. By induction on the derivation of the typing rules. Case WF-ACT-SET-E is direct. Case WF-ACT-SET we use the induction hypothesis to prove that $H, \{c\colon h\} \vdash A\colon \diamond$ and $H, \{c\colon h\} \vdash A'\colon \diamond$; the remaining conditions are given by the hypotheses.
2. By inspecting the typing rules.
3. We apply rule T-VIEW to typify the clocks of the view, then we prove T-CLOCK-SEQ with (2).
4. By induction on the derivation of the typing relation. Cases T-MAKE, T-NEXT, and T-JOIN are direct. Case T-RESUME and T-DROP are proved similarly, using (2) to typify clock v. The proof for cases T-FINISH, T-ASYNC, and T-LET follow by induction hypothesis. For case T-ASYNC we also use rule T-CLOCK-SEQ and (2) to typify the clocks of the arguments.

□

Notice we do not allow heap weakening for it would introduce in the type environment clocks not present in the state.

Lemma 2 (Substitution)**.** *If* $\Gamma;\mathscr{R}' \vdash v\colon \tau$ *and* $\Gamma, x\colon \tau;\mathscr{R};\mathscr{Q} \vdash e\colon T$ *and* $\mathscr{R} \subseteq \mathscr{R}'$ *then* $\Gamma;\mathscr{R};\mathscr{Q} \vdash e[v/x]\colon T$.

Proof outline. For T-VALUE we analyse two cases: when the value is the variable being substituted, and when the it is not replaced. For the former case, we apply lemma 1 on the first hypothesis. For the latter case we use rule T-VAR. Rules T-MAKE, T-NEXT, and T-JOIN are direct. Cases T-DROP and T-RESUME follow by induction hypothesis. Rule T-ASYNC is the most complex. For the clocks being shared we use lemma 1 and the second hypothesis. For the expression being spawned we apply the induction hypothesis. Rule T-FINISH is proved similarly to T-ASYNC, but simpler since T-FINISH has no arguments and its set of clocks is empty. □

Lemma 3 (Preservation for activities)**.** *If* $\Gamma \vdash H$ *and* $\Gamma \vdash V : \mathscr{Q}$ *and* $\mathscr{Q} \subseteq \mathscr{R}$ *and* $\Gamma;\mathscr{R};\mathscr{Q} \vdash e : T$ *and* $\Gamma \vdash A$ *and* $l \in \operatorname{dom} H$ *and* $H;(V,e,A) \rightarrow_l H';A'';(V',e',A')$*, then* $\Gamma' \vdash H'$ *and* $\Gamma' \vdash V' : \mathscr{Q}'$ *and* $\mathscr{Q}' \subseteq \mathscr{R}'$ *and* $\Gamma';\mathscr{R}';\mathscr{Q}' \vdash e' : T$ *and* $\Gamma' \vdash A', A''$*, for some* $\Gamma' \supseteq \Gamma$.

Proof outline. Despite the scary look of the statement, its proof is a routine inspection of the rules in the relation $H;(V,e,A) \rightarrow_l H';A'';(V',e',A')$. □

Lemma 4 (Preservation for $\mathsf{X10}\,|_{\mathsf{clocks}}$)**.** *If* $\Gamma \vdash S$ *and* $S \rightarrow S'$ *then* $\Gamma' \vdash S'$ *and* $\Gamma \subseteq \Gamma'$.

Proof outline. By induction on the derivation of the relation $S \rightarrow S'$. In all cases we build the derivation tree for $\Gamma \vdash S$ using rules T-STATE, T-ACT-SET, and T-ACT, collect the hypotheses, use the above lemmas, and then build a tree for $\Gamma' \vdash S'$ using the same typing rules. The base cases are when the derivation ends with rules R-LET-VAL and R-LET. For R-LET-VAL we take $\Gamma' = \Gamma$ and use the substitution lemma 2. For R-LET we use the (specially crafted) preservation for activities (lemma 3), as well as the weakening (lemma 1) for the extant activity set A. The induction step is when derivation ends with rule R-ACTIVITY; in this case we use the weakening lemma. □

Theorem 5 (Type Safety)**.** *If* $\Gamma \vdash S$ *and* $S \rightarrow^* S'$*, then* $S' \notin$ Error.

Proof outline. We first establish that $\Gamma' \vdash S'$ using preservation (lemma 4). Then we proceed by contradiction. The contradiction is proved by induction on the definition of Error predicate. For the base cases of **resume**, **drop**, and **async** we build the derivation trees for the errors in Figure 5, to conclude that $\Gamma' \vdash V : \mathscr{R}$ and $\Gamma';\mathscr{R} \vdash c : \mathbf{clock}(\alpha)$. Sequent $\Gamma' \vdash V : \mathscr{R}$ is derived from rule T-VIEW, which effectively establishes a *one-to-one* correspondence between the clock names c in V and the singleton types α in $\mathscr{R}$. On the other hand, sequent $\Gamma';\mathscr{R} \vdash c : \mathbf{clock}(\alpha)$ is derived via rule T-WF-C, which says that $\alpha \in \mathscr{R}$. Since $\alpha \in \mathscr{R}$, the correspondence allows us to conclude that $c \in \operatorname{dom} V$. Establishing that $v \in \operatorname{dom} H$ is easier. Given that $\Gamma' \vdash H$, we conclude that $\operatorname{dom} H = \operatorname{dom}(\Gamma'|_{\mathbf{C}})$, and from sequent $\Gamma';\mathscr{R} \vdash c : \mathbf{clock}(\alpha)$ we know that $c \in \operatorname{dom} \Gamma'$, hence done. □

We anticipate a *progress property* for typable processes. Typability ensures that processes do not get stuck when dropping a clock that is not in its clock set anymore, or when otherwise trying to access a clock that it not allocated in the heap. The remaining case is **next** where the activity waits for set C_1 (the set of quiescent clocks the activity is registered with) to grow until becoming (together with C_2—the set of clocks that have already advance their phase) the clock set of the activity. And this is bound to happen for both **next** and **drop**, since in each activity both implicitly resume all clocks. We foresee as well that typability also rules out programs that deadlock, since **finish** expressions can only use clocks created in its body expressions.

6 Discussion and future work

We study two synchronisation constructs of X10: a primitive **finish** that waits for the termination of activities (lightweight threads), and clocks (a generalisation of barriers). To better understand the language we define an operational semantics and a type system (alternative to the constraint-based system [5]) for a subset of X10 called $\mathsf{X10}\,|_{\mathsf{clocks}}$. Our main result is the type safety of the language (Theorem 5) and we expect to have a progress property that will allow us to prove deadlock freedom.

Our semantics represents clocks in the heap as triples $\langle p,R,Q\rangle$ relying on two sets for recording the registered activities R and the quiesced activities Q on a clock. Implementing operations that work with sets is costly; for instance rule R-NEXT needs to compute sets C_1 and C_2, by checking if sets R and Q are equal, and then verify if $C_1 \cup C_2 = \operatorname{dom} V$. Should we make a real life implementation of the proposed semantics, set operations would have a significant impact on performance. We sketch a much faster approach that chooses to represent clocks as triples $\langle p,r,q\rangle$ describing the clock phase, as before, but taking r and q as the cardinal numbers of sets R and Q. With this representation we lose information about the identity of the activities registered with a clock and, in particular, we cannot determine if an activity has already resumed in the current phase (*vide* rules R-ASYNC and R-RESUME). To overcome this problem we need to enrich the clock local view with an indicator of whether an activity has resumed in the current phase. Thus, a clock local view becomes a pair $\langle p,b\rangle$ containing the current clock phase p (as before) and the resume boolean indicator b, describing when the activity has resumed. With this information it is straightforward to adapt rules R-ASYNC, R-MAKE, R-RESUME, R-NEXT, and R-DROP. For instance, rule R-RESUME only updates the clock global view ($q \leftarrow q+1$) whenever its local view indicator is false. Also, rule R-NEXT needs to set r to zero when advancing the clock global phase, and to clear the indicator b upon advancing the clock local phase. Checking that all activities registered with a clock have quiesced amounts to compare two integer values ($r = s$), instead of two sets R and Q as before. The main reasons for not adopting the semantics just sketched are that the chosen semantics needs fewer rules and is easier to read and understand.

We intend to investigate imperative features of the language, specially those related with clocks, and also other language constructs. The language report reads “X10 does not contain a register statement that would allow an activity to discover a clock in a data structure and register itself on it”; we would like to study type-safe extensions to the language that might alleviate this restriction in controlled situations. Furthermore, we expect to extend our results to $\mathsf{X10}\,|_{\mathsf{clocks}}$ equipped with recursion or some form of iteration. Futures are an example of such construct. This primitive is a form of a function that evaluates asynchronously, like an activity, but can be *forced* to finish locally to return a value. The semantics of a future, in what regards termination, is like the **finish** construct, but its use cases are different. We would also like to allow futures to register themselves with clocks, a feature missing in X10.

Phasers are a coordination construct that unifies collective and point-to-point synchronisations with performance results competitive to existing barrier implementations [6]. Phasers can be seen as an extension over clocks that allow for more fine-grained control over synchronisation modes. *Phaser accumulators* are reduction constructs for dynamic parallelism that integrate with phasers [7]. Although further investigation is needed, we believe our work can be extended to accommodate phasers and phaser accumulators, specially with regards to the operational similarities between clocks and phasers.

References

[1] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA'05*, pages 519–538. ACM, 2005.

[2] Rajiv Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. *SIGARCH Computer Architecture News*, 17(2):54–63, 1989.

[3] Jonathan K. Lee and Jens Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of PPoPP'10*, pages 25–36. ACM, 2010.

[4] Vijay Saraswat. Report on the programming language X10, version 2.01. Technical report, IBM Research, 2010.

[5] Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *Proceedings of CONCUR'05*, volume 3653 of *LNCS*, pages 353–367. Springer, 2005.

[6] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS'08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288. ACM, 2008.

[7] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phaser accumulators: A new reduction construct for dynamic parallelism. In *IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12. IEEE Computer Society, 2009.

[8] Vasco T. Vasconcelos, Francisco Martins, and Tiago Cogumbreiro. Type inference for deadlock detection in a multithreaded typed assembly language. In *Post-proceedings of PLACES'09*, volume 17 of *EPTCS*, pages 95–109, 2010.