

Sequential semantics for concurrent programs

Vijay Saraswat Yoav Zibin Nalini Vasudevan Olivier Tardieu Nathaniel Clinger

IBM research in TJ Watson

vsaraswa | yzibin | dcunnin | tardieu | clinger@us.ibm.com

Abstract

The safe parallelization of sequential programs is an important and challenging task. A parallelization is safe if it ensures that no new behaviors are introduced; the semantics of the parallel program is identical to that of the sequential program. In particular the parallel program must not exhibit scheduler indeterminacy or deadlock.

This is a challenging problems. Imperative memory locations are famously indeterminate under concurrent read/write operations. Further, most concurrent programming languages support synchronization mechanisms that enable a thread to wait until some appropriate condition is true, thus leading to the possibility of deadlock.

In this paper we develop a very rich fragment of the concurrent, imperative programming language X10 in which *all* programs have sequential semantics. The advantages of such a language are striking: the programmer simply cannot introduce determinacy or deadlock-related errors. All programs can be debugged as if they are sequential programs, without having to reason about all possible interleaving of threads.

The fragment includes all of X10's most powerful constructs – (clocked) *finish* and *async*, *places* and *at*. We introduce two new abstractions – *accumulators* and *clocked values* – with lightweight compiler and run-time support. Accumulators of type τ permit multiple concurrent writes, these are reduced into a single value by a user-specified reduction operator. Accumulators are designed to be determinate and deadlock-free at run-time, under all possible uses. Clocked values of type τ operate on two values of τ (the *current* and the *next*). Read operations are performed on the current value, write operations on the next. Such values are implicitly associated with a clock and the current and next values are switched (determinately) on quiescence of the clock. Clocked values capture the common “double buffering” idiom. Accumulators and clocked values are designed in such a way that they are deadlock-free and determinate even in the presence of arbitrary aliasing and patterns of concurrency.

We combine these notions with a lightweight framework for detecting disjointness of access via static effects checking.

We show all programs written in this fragment of the language are guaranteed to have sequential semantics. That is, programs in this fragment may be developed and debugged as if they are sequential, without running into the state explosion problem.

Further, many common patterns of concurrent execution can be written in this language in an “obvious” way. These patterns include histograms, all-to-all reductions, stencil computations, master/slave

execution, and most of the important HPC computing idioms. We show that there are simple statically-checkable rules that can establish for many common idioms that some run-time synchronization checks can be avoided.

The key technical problem addressed is unrestricting aliasing in the heap. We develop the idea of an *implicit ownership domain* (first introduced in the semantics of clocks in X10 in a paper in Concur 2005) that newly created objects may be registered with the current activity so that they can be operated upon only by this activity and its newly spawned children activity subtree. This permits determinacy and deadlock freedom to be established locally, independent of the context in which the code is being run.

1. Introduction

Desiderata:

- Data-structures should by design be dynamically determinate and deadlock-free.
- They should be first-class – they can be stored in data-structures/read from them, passed as arguments to procedures/returned etc with no restrictions.
- Usable – common idioms should be naturally and elegantly expressible.
- Additional static type-checking can provide extra guarantees (e.g. no concurrency related run-time exceptions) that may aid efficient implementation.

We discuss three examples

* clocks * accumulators * clocked types

Challenge Arbitrary nature of object graphs.

1. *Use activity registration as a mechanism to tame object graphs.*
2. *Focus on structured concurrency. Using scoping and block-structure to delimit regions of code that may execute in parallel and affect the data structure.*
3. *Accumulation can be defined safely by delaying. However, the delay operation is guaranteed to be deadlock-free.*
4. *Clocked types support phased computation, another common idiom particularly for stencil computations.*

Key contributions:

1. *Identification of determinate, deadlock-free data-structures.*
2. *Discussion of design alternatives which points out the difficulty of integrating these ideas in a modern OO language.*
3. *Discussion of various idioms expressible using these data-structures.*
4. *Proof of determinacy and deadlock-freedom in an abstract version of the language.*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]...\$10.00

These constructs are implemented in X10, available as open source from SVN head and will be in the next release of X10.

Semantics and theorems for an abstract version of the language.

2. X10

Discussion of X10.

2.0.1 sync

We introduce a new derivative synchronization concept, that of syncing.

We introduce `Runtime.sync()`. It returns only after all asyncs spawned by the current activity have terminated or stopped at an advance on a clock registered on the current activity.

Programs using `finish/async/at/atomic/clocks/Runtime.sync()` cannot deadlock.

3. Accumulators and Clocked Types

3.1 Accumulators

Each async (dynamically) has a set of registered @Sync accumulators and @Async accumulators.

- The registered @Async accumulators for an activity are the registered @Sync and @Async accumulators of its parent activity.
- The registered @Sync accumulators for an activity are the ones it has created.

This permits computations to be determinate even though accumulators can be stored in heaps, since no async other than the async that created the accumulator or one of its progeny can actually operate on them. One can still use the flexibility of the heap to arrange for complex data-dependent transmission pathways for the accumulator from point of creation to point of use. e.g. arrays of accumulators, hash-maps, etc. In particular, accumulators can be passed into arbitrary method invocations, returned from methods etc, with no restrictions.

The method

```
Runtime.isRegistered[T](x:Acc[T]):Int
```

returns 0 if `x` is not registered with the current activity, 1 if it is @Sync registered, and 2 if it is @Async registered.

An invocation `e.m(e1, ..., en)` of an @Sync method on an Acc is translated to:

```
{
  val x = e;
  if (Runtime.isRegistered(x) != 1)
    throw new IllegalAccAccess(x);
  Runtime.sync();
  x.m(e1, ..., en)
}
```

The @Sync methods on an Acc are ones that return its current value (@Read) and ones that reset it (@Write).

An invocation `e.m(e1, ..., en)` of an @Async method on an Acc is translated to:

```
{
  val x = e;
  if (Runtime.isRegistered(x) == 0)
    throw new IllegalAccAccess(x);
  x.m(e1, ..., en);
}
```

The only @Async method on an Acc is the one that offers an update to its value (@Write).

In many cases the compiler can statically evaluate whether `Runtime.isRegistered(x) > 0` and/or whether a call to `Runtime.sync()` will suspend.

It may then appropriately simplify the above code. e.g. in the code below

```
val x:Acc[Int] = new Acc[Int](0, Int.+);
finish for (i in 0..100000) async
  x <- i;
Console.OUT.println("x is " + x());
```

the compiler can infer that `x()` won't suspend, due to the `finish`. Hence it may eliminate the run-time suspension check. Further it can establish that `x` is @Sync registered with the current activity, hence it can eliminate the access check.

Notes:

- Accs are first-class values. There are no restrictions in storing them in data-structures, reading them, passing them as arguments to methods, returning them from methods etc.
- However, any attempt to use it will fail unless the Acc is registered with the current activity.
- The runtime checks in `Runtime.sync()` and `Runtime.registered(..)` ensure that the operations on an Acc are determinate.

Proposition 3.1. *Acc's are determinate under arbitrary usage.*

3.2 Example use of Acc

Example 3.2 (Histogram). @det

```
def histogram(N:Int,
  A:Rail[Int(0..N)]):Rail[Int](N+1) {
  val result = new Rail[Acc[Int]](N+1, (Int)=>new
    Acc[Int](0, Int.+));
  finish for (i in A.values()) async {
    result(i) <- 1;
  }
  return new Rail[Int](N+1, (i:Int)=> result(i));
}
```

Example 3.3 (Distributed word-count). // A DistHashMap is used because the input is a DistStream

```
@det
def
  wordCount(m:DistStream[Word]):DistHashMap[Word,Int](m.dist)
{
  val a = new DistHashMap[Word, Acc[Int]](m.dist,
    (w:Word)=> new Acc[Int](Int.Sum));
  finish for (p in m.dist.places()) async at (p) {
    for (word in m(p).words())
      a(word) <- 1;
  }
  return new DistHashMap[Word, Int](m.dist,
    (w:Word)=> a(w));
}
```

Accumulators can be used to implement collective operations such as all-to-all reductions in a straightforward “shared memory” style.

Here we show the single-sided, blocking version.

Example 3.4. @det

```
def reduce[T](in:DistArray[T], red:Reducible[T]):T {
  val acc = new Acc[T](red);
  val temp = new GlobalRef[Acc[T]](acc);
  finish for (dest in in.dist.places()) async
    at(dest) {
      val local = new Acc[T](red);
      for (p in in.dist | here) {
        local <- in(p);
      }
      val x = local();
      async at(origin) temp() <- x;
    }
  return acc();
}
```

An `allReduce` can be implemented by following the above operation with a broadcast:

```
@def
def
  allReduce[T] (in:DistArray[T] {self.dist==Dist.UNIQUE},
    red:Reducible[T], out:DistArray[T] (in.dist)):void {
    val x = reduce(in, red);
    finish for (dest in out.dist.places()) async at
      (dest) {
        for (p in out.dist |here)
          out(p)=x;
      }
  }
```

One can write this code using a clock (to avoid two finish nests).

The collective style requires extending clock so the advance method takes arbitrary args and performs collective operations on them, mimicking the MPI API.

Collecting finish The collecting finish construct is of the form `finish(r) S`, where `r` is an instance of `Reducible[T]` and `S` is a statement. Within the dynamic execution of `S`, any execution of the statement `offer t`; results in a value `t` being accumulated in a set. (`t` must be of type `T`.) The result of the reduction of this set of `T` values with `r` is then returned.

```
{
  val x = new Acc[T] (r);
  finish {
    S [ x <- t / offer t; ]
  }
  x()
}
```

An attractive aspect of collecting finish is that nesting is used quite naturally to reflect the relationship between the parallel computation performing the accumulation and the value returned. In particular there is no need to explicitly introduce the notion of an accumulator, or to register it with the current block, or to check that other activities in the current block have quiesced.

On the other hand, this strength is also a limitation. It is not possible to use the same idiom for clocked code, i.e. collect values being offered by multiple `clocked` asyncs in the current phase. Further, using this idiom safely across method calls requires the addition of `offers T` clauses (similar to `throws T` clauses, specifying the type of the value being offered. Otherwise at run-time a value of an incompatible type may be offered leading to a run-time exception. Finally, the lack of a name for the result being collecting means that it is difficult to use the same computation to accumulate multiple separate values without more contortions. e.g. one could set the return type to be a tuple of values, but then the `offer` statement would need to specify the index for which the `offer` was intended. Now it is not clear that the index type should be arithmetic – why should not one be able to collect into the range of a `HashMap` (so the index type could be an arbitrary type `Key`)?

We feel that these decisions are all orthogonal to the actual process of accumulating and should be dealt with by the data-structuring aspects of the language design.

3.3 Clocked types

The central idea behind clocked data-structures is that read/write conflicts are avoided using “double buffering.” Two versions of the data-structure are kept, the *current* and the *next* versions. Reads can be performed simultaneously by multiple activities – they are performed on the current version of the data-structure. Writes are performed on the next version of the data-structure. On detection of termination of the current phase – when all involved activities are quiescent – the current and the next versions are switched.

`Clocked[T]` and `ClockedAcc[T]` are distinguished in that unlike the former the latter permits accumulation operations.

Clocked objects are registered with activities, just like accumulators. This permits computations to be determinate even though objects can be stored in heaps, since no async other than a child of the async that creates the clocked object can actually operate on them.

Each async (dynamically) has a set of registered clocked values. The registered clocked values for an activity are the clocked values it has created, and the ones registered to its parent activity.

`Clocked[T]` has a constructor that takes two `T` arguments, these are used to initialize the *now* and *next* fields. These arguments should be “new” (that is, no other data-structure should have a reference to these arguments).

For `x:Clocked[T]` the following operations available to any activity on which `x` is registered:

- `x()` – this returns the value of the current field.
- `x() = t` – this sets the value of the next field. Note: write-write conflicts are possible since multiple activities may try to set the value at the same time.
- `x.finalized()` – this returns the value of the *now* field but modifies the internal state so that any subsequent attempt to use `x()=t` will result in a runtime exception.

`ClockedAcc[T]` has a constructor that takes two `T` values and a `Reducer[T]` as argument. The two `T` values are used to initialize the *current* and *next* fields. These arguments should be “new” (that is, no other data-structure should have a reference to these arguments). The reducer is used to perform accumulate operations.

Operations for `x:ClockedAcc[T]`:

- `x()` – this returns the value of the *now* field.
- `x() <- t` – this accumulates `t` into the next field. Note: No write-write conflicts are possible.
- `x() = t` – this resets the value of the next field to `t`. To avoid read/write and write/write conflicts, this operation should be invoked only by the closure argument of `Clock.advanceAll(closure)`. (See below.)
- `x.finalized()` – this returns the value of the *now* field but modifies the internal state so that any attempt to use `x()=t` or `x() <- t` will result in a runtime exception.

We add the following method on `Clock`:

```
public static def advanceAll(x:()=>void) {...}
```

If all activities registered on the clock invoke `advanceAll(f)` (for the same value `f`), then `f` is guaranteed to be invoked by some activity `A` registered on the clock at a point in time when all other activities have entered the `advanceAll(f)` call and the *current/next* swap has been performed for all registered clocked values. At this point – also called the *clock quiescent point* – it is guaranteed that none of the other activities are performing a read or write operation on user-accessible memory.

(A possible implementation of `Clocked[T]` and `ClockedAcc[T]` is that a system-synthesized closure (that performs the *current/next* swap) is run at the clock quiescent point before the user specified closure is run.)

Example 3.5. `\input {examples/stencil.x10}`

In the example above there is no need to reinitialize the value of `err()` between phases since the value will monotonically decrease. However for some other computations this may be necessary. For example, suppose the error metric was the sum of all errors. The the

above code would change as follows. We would pass in a different reduction operation `red` that sums rather than returns a max. Further, we would replace the `Clock.advanceAll()` call with

```
Clock.advanceAll( ()=>{err()=0.0D;});
```

This resets the next value of `err` to be 0.0D before the accumulations start to happen.

4. Implementation Considerations

Registration of accumulators with activities needs to be implemented efficiently. This may require the implementation of an activity stack, with registration information being looked up lazily and cached, rather than pushed eagerly. Also this information is clearly not needed in the body of `async`'s that can be statically analyzed to not contain accumulator operations.

5. Semantics

Featherweight X10 (FX10) is a formal calculus for X10 intended to complement Featherweight Java (FJ). It models imperative aspects of X10 including the concurrency constructs `finish` and `async`.

Overview of formalism

5.1 Syntax

$P ::= \bar{L}, S$ $L ::= \text{class } C \text{ extends } D \{ \bar{F}; \bar{M} \}$ $F ::= \text{var } f : C$ $M ::= \text{def } m(\bar{x} : \bar{C}) : C \{ S \}$ $p ::= l \mid x$ $e ::= p.f \mid \text{new } C \mid \text{new Acc}(r, z) \mid a()$ $B ::= (\pi) \{ S \}$ $S ::= \epsilon \mid p.f = e; \mid p.m(\bar{p}); \mid \text{val } x = e; \mid$ $\quad \text{acc } x = \text{new Acc}(r, z); \mid a \leftarrow p$ $\quad [\text{clocked}] \text{ finish } B$ $\quad [\text{clocked}] \text{ async } B \mid S \mid S$	Program. cLass declaration. Field declaration. Method declaration. Path. Expressions. Blocks. Statements.
--	--

Figure 1. FX10 Syntax. The terminals are locations (l), parameters and this (x), field name (f), method name (m), class name (B, C, D, Object), and keywords (`new`, `finish`, `async`, `val`). The program source code cannot contain locations (l), because locations are only created during execution/reduction in R-NEW of Fig. 2.

Fig. 1 shows the abstract syntax of FX10. ϵ is the empty statement. Sequencing associates to the left. The block syntax in source code is just $\{ S \}$, however at run-time the block is augmented with (the initially empty) set of locations π representing the write capabilities of the block.

Expression `val x = e; S` evaluates e , assigns it to a new variable x , and then evaluates S . The scope of x is S .

The syntax is similar to the real X10 syntax with the following difference: FX10 does not have constructors; instead, an object is initialized by assigning to its fields. X10 uses `acc` and not `val` for new accumulator declarations. These are minor simplifications intended to streamline the formal presentation.

5.2 Reduction

A *heap* H is a mapping from a given set of locations to *objects*. An object is a pair $C(F)$ where C is a class (the exact class of the object), and F is a partial map from the fields of C to locations.

The reduction relation is described in Figure 2. An *S-configuration* is of the form s, H where s is a statement and H is a heap (representing a computation which is to execute s in the heap H), or H (representing a successfully terminated computation), or \diamond representing a computation that terminated in an error. An *E-configuration* is

of the form e, H and represents the computation which is to evaluate e in the configuration H . The set of *values* is the set of locations; hence E-configurations of the form l, H are terminal.

Two transition relations \leadsto_π are defined, one over S-configurations and the other over E-configurations. Here π is a set of locations which can currently be asynchronously accessed. Thus each transition is performed in a context that knows about the current set of capabilities. For X a partial function, we use the notation $X[v \mapsto e]$ to represent the partial function which is the same as X except that it maps v to e .

The rules for termination, step, val, new, invoke, access and assign are standard. The only minor novelty is in how `async` is defined. The critical rule is the last rule in (R-STEP) – it specifies the “asynchronous” nature of `async` by permitting S to make a step even if it is preceded by `async S1`. Further, each block records its set of synchronous capabilities. When descending into the body, the block’s own capabilities are added to those obtained from the environment.

The rule (R-NEW) returns a new location that is bound to a new object that is an instance of C with none of its fields initialized. The rule (R-ACCESS) ensures that the field is initialized before it is read (f_i is contained in \bar{F}). The rule (R-ACC-N) adds the new location bound to the accumulator as a synchronous access capability to the current `async`. The rule (R-ACC-A-R) permits the accumulator to be read in a block provided that the current set of capabilities permit it (if not, an error is thrown), and provided that the only statements prior to the read are `re` is no nested `async` prior to the read are clocked `async`s that are stuck at an advance.

The rule (R-ACC-W) updates the current contents of the accumulator provided that the current set of capabilities permit asynchronous access to the accumulator (if not, an error is thrown).

The rule (R-ADVANCE) permits a clocked `finish` to advance only if all the top-level clocked `async`s in the scope of the clocked `finish` and before an advance are stuck at an advance. A clocked `finish` can also advance if all the the top-level clocked `async`s in its scope are stuck. In this case, the statement in the body of the clocked `finish` has terminated, and left behind only possibly clocked `async`. This rule corresponds to the notion that the body of a clocked `finish` deregisters itself from the clock on local termination. Note that in both these rules unlocked `async` may exist in the scope of the clocked `finish`; they do not come in the way of the clocked `finish` advancing.

5.3 Results

6. Related Work

DPJ
Phasers
CCP

7. Conclusion

We show that many determinate concurrent programs can be written in X10 using determinate, deadlock-free constructs, so that they are determinate by design.

Acknowledgments

This material is based in part on work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

- [1] V. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *Concur’05*, pages 353–367, 2005.

$$\begin{array}{c}
\frac{}{\mathbf{e}, \mathbf{H} \rightsquigarrow_{\pi} \mathbf{H}} \text{ (R-EPSILON)} \quad \frac{S, \mathbf{H} \rightsquigarrow_{\pi} S', \mathbf{H}' \mid \mathbf{H}' \mid \Diamond}{(\pi_1) \{S\}, \mathbf{H} \rightsquigarrow_{\pi_2} (\pi_1) \{S'\}, \mathbf{H}' \mid \mathbf{H}' \mid \Diamond \quad \pi = \pi_1, \pi_2} \text{ (R-TRANS)} \\
\frac{B, \mathbf{H} \rightsquigarrow_{\pi} B', \mathbf{H}' \mid \mathbf{H}' \mid \Diamond}{[\text{clocked}] [\text{finish} \mid \text{async}] B, \mathbf{H} \rightsquigarrow_{\pi} [\text{clocked}] [\text{finish} \mid \text{async}] B', \mathbf{H}' \mid \mathbf{H}' \mid \Diamond} \text{ (R-TRANS-B)} \\
\frac{S, \mathbf{H} \rightsquigarrow_{\pi} S', \mathbf{H}'}{[\text{clocked}] \text{ async } B, \mathbf{H} \rightsquigarrow_{\pi} [\text{clocked}] \text{ async } B', \mathbf{H}'} \text{ (R-ASYNC)} \\
\frac{\mathbf{e}, \mathbf{H} \rightsquigarrow 1, \mathbf{H}'}{\text{val } x = \mathbf{e}; S, \mathbf{H} \rightsquigarrow_{\pi} S[1/x], \mathbf{H}'} \text{ (R-VAL)} \quad \frac{1' \notin \text{dom}(\mathbf{H})}{\text{new } C, \mathbf{H} \rightsquigarrow_{\pi} 1', \mathbf{H}[1' \mapsto C()]} \text{ (R-NEW)} \quad \frac{\mathbf{H}(1') = C(\dots) \quad \text{mbody}(\mathbf{m}, C) = \bar{x}.S}{1'.\mathbf{m}(1), \mathbf{H} \rightsquigarrow_{\pi} S[1/\bar{x}, 1'/\text{this}], \mathbf{H}} \text{ (R-INVOKE)} \\
\frac{\mathbf{H}(1) = C(\bar{f} \mapsto 1') \quad g = f_i \mid g \notin \bar{f}}{1.g, \mathbf{H} \rightsquigarrow_{\pi} 1'_i, \mathbf{H} \mid \Diamond} \text{ (R-ACCESS)} \quad \frac{\mathbf{H}(1) = C(F)}{1.f = 1', \mathbf{H} \rightsquigarrow_{\pi} \mathbf{H}[1 \mapsto C(F[\bar{f} \mapsto 1'])]} \text{ (R-ASSIGN)} \\
\frac{1 \notin \text{dom}(\mathbf{H})}{(\pi_1) \{\text{acc } x = \text{new Acc}(r, z); S\}, \mathbf{H} \rightsquigarrow_{\pi} (\pi_1, 1) \{S[1/x]\}, \mathbf{H}[1 \mapsto \text{Acc}(r, z)]} \text{ (R-ACC-N)} \\
\frac{}{\mathbf{e} \downarrow} \text{ (R-Q-CA)} \quad \frac{B \downarrow}{\text{clocked async } B \downarrow} \text{ (R-Q-CA)} \quad \frac{S \downarrow}{(\pi) \{S\} \downarrow} \text{ (R-Q-B)} \quad \frac{S_1 \downarrow \quad S_2 \downarrow}{S_1 \quad S_2 \downarrow} \text{ (R-Q-S)} \\
\frac{\mathbf{H}(a) = \text{Acc}(r, v) \quad S \downarrow \quad a \in \pi_1 \mid a \notin \pi_1}{(\pi_1) \{S \text{ val } x = a(); S_1\}, \mathbf{H} \rightsquigarrow_{\pi} (\pi_1) \{S_1[v/x]\}, \mathbf{H}' \mid \Diamond} \text{ (R-ACC-R)} \quad \frac{\mathbf{H}(a) = \text{Acc}(r, v) \quad w = r(v, p) \quad a \in \pi \mid a \notin \pi}{a \leftarrow p, \mathbf{H} \rightsquigarrow_{\pi} \mathbf{H}[a \mapsto \text{Acc}(r, w)] \mid \Diamond} \text{ (R-ACC-W)} \\
\frac{S_1 \downarrow \quad S, \mathbf{H} \rightsquigarrow_{\pi} S', \mathbf{H}'}{S_1 \quad S, \mathbf{H} \rightsquigarrow_{\pi} S_1 \quad S', \mathbf{H}'} \text{ (R-CA)} \\
\frac{B \overset{+}{\rightsquigarrow} B'}{\text{clocked async } B \overset{+}{\rightsquigarrow} \text{clocked async } B'} \text{ (R-ADV-CA)} \quad \frac{}{(\text{async} \mid [\text{clocked}] \text{ finish}) B \overset{+}{\rightsquigarrow} (\text{async} \mid [\text{clocked}] \text{ finish}) B \quad \text{advance}; S \overset{+}{\rightsquigarrow} S} \text{ (R-ADV-A, CAF)} \\
\frac{S \overset{+}{\rightsquigarrow} S'}{(\pi) \{S\} \overset{+}{\rightsquigarrow} (\pi) \{S'\}} \text{ (R-ADV-B)} \quad \frac{S_1 \overset{+}{\rightsquigarrow} S'_1 \quad S_2 \overset{+}{\rightsquigarrow} S'_2}{S_1 \quad S_2 \overset{+}{\rightsquigarrow} S'_1 \quad S'_2} \text{ (R-ADV-S)} \quad \frac{B \overset{+}{\rightsquigarrow} B'}{\text{clocked finish } B, \mathbf{H} \rightsquigarrow_{\pi} \text{clocked finish } B', \mathbf{H}} \text{ (R-ADV)}
\end{array}$$

Figure 2. FX10 Reduction Rules ($S, \mathbf{H} \rightsquigarrow_{\pi} S', \mathbf{H}' \mid \mathbf{H}'$ and $\mathbf{e}, \mathbf{H} \rightsquigarrow_{\pi} 1, \mathbf{H}'$).