Semantics of (Resilient) X10

Silvia Crafa[1], David Cunningham[2], Vijay Saraswat[3], Avraham Shinnar[3], Olivier Tardieu[3]

[1] University of Padova, Italy. crafa@math.unipd.it
[2] Google Inc., USA. sparkprime@gmail.com
[3] IBM TJ Watson Research Center, USA.
{vsaraswa,shinnar,tardieu}@us.ibm.com

Abstract. We present a formal small-step structural operational semantics for a large fragment of X10, unifying past work. The fragment covers multiple places, mutable objects on the heap, sequencing, `try/catch`, `async`, `finish`, and `at` constructs. This model accurately captures the behavior of a large class of concurrent, multi-place X10 programs. Further, we introduce a formal model of resilience in X10. During execution of an X10 program, a place may fail for many reasons. Resilient X10 permits the program to continue executing, losing the data at the failed place, and most of the control state, and repairing the global control state in such a way that key semantic principles hold, the Happens Before Invariance Principle, and the Exception Masking Principle. These principles permit an X10 programmer to write clean code that continues to work in the presence of place failure. The given semantics have additionally been mechanized in Coq.

1 Introduction

The need for scale-out programming languages is now well-established, because of high performance computing applications on supercomputers, and analytic computations on big data. Such languages – based for example on a partitioned global address space ([21,9], [10]) – permit programmers to write a single program that runs on a collection of places on a cluster of computers, can create global data-structures spanning multiple places, can spawn tasks at remote places, detect termination of an arbitrary tree of spawned tasks etc. The power of such languages is shown by programs such as M3R, which implement a high-performance, main-memory version of Hadoop Map Reduce [22] in a few thousand lines of code. Other high performance multi-place libraries have been developed for graph computations [12] and sparse matrix computations [23].

At the same time, the practical realities of running large-scale computations on clusters of commodity computers in commercial data centers are that nodes may fail (or may be brought down, e.g. for maintenance) during program executions. This is why multi-place application frameworks such as Hadoop [13], Resilient Data Sets [25], Pregel [18] and MillWheel [2] support resilient computations out of the box. In case of node failure, relevant portions of the user computation are restarted.

A new direction has been proposed recently in [11]: extending a general purpose object-oriented, scale-out programming language (X10) to support resilience. The hypothesis is that application frameworks such as the ones discussed above can in fact be

programmed in a much simpler and more direct fashion in an object-oriented language (powerful enough to build parallel, distributed libraries) that already supports resilience. It is feasible to extend X10 in this way since is based on a few, orthogonal constructs organized around the idea of *places* and *asynchrony*. A place (typically realized as a process) is simply a collection of objects together with the threads that operate on them. A single computation may have tens of thousands of places. The statement `async S` supports asynchronous execution of `S` in a separate task. `finish S` executes `S`, and waits for all tasks spawned by `S` to terminate. Memory locations in one place can contain references (*global refs*) to locations at other places. To use a global ref, the `at (p) S` statement must be used. It permits the current task to change its place of execution to `p`, execute `S` at `p` and return, leaving behind tasks that may have been spawned during the execution of `S`. The termination of these tasks is detected by the `finish` within which the `at` statement is executing. The values of variables used in `S` but defined outside `S` are serialized, transmitted to `p`, de-serialized to reconstruct a binding environment in which `S` is executed. Constructs are provided for unconditional (`atomic S`) and conditional (`when (c) S`) atomic execution. Finally, Java-style non-resumptive exceptions (`throw`, `try`/`catch`) are supported. If an exception is not caught in an `async`, it is propagated to the enclosing `finish` statement. Since there may be many such exceptions, they appear wrapped in a `MultipleExceptions` exception.

[11] shows that this programming model may be extended to support resilience in a surprisingly straightforward way. A place `p` may fail at any time with the loss of its heap and tasks. Any executing (or subsequent) tasks on that place throw a `DeadPlaceException` (`DPE`). Global refs pointing to locations hosted at `p` now "dangle"; however they can only be dereferenced via an `at (p) S`, and this will throw a `DPE` exception. If a task at a failed place has started a task T at another place, this task is not aborted. Instead Resilient X10 posits a high-level principle, the Happens Before Invariance (HBI) principle: failure of a place should not alter the happens before relationship between statement instances at remaining places. [11] shows that many interesting styles of resilient programming can be expressed in Resilient X10. The language is implemented at a fairly modest cost.

In this paper we formalize the semantics of Resilient X10. Our fundamental motivation is to provide a mechanized, formal semantics for a core fragment of Resilient X10 that is separate from the implementation and can be used as a basis for reasoning about properties of programs and for establishing that principles such as HBI actually hold.

We proceed as follows. Our first task is to formalize a large portion of X10, called TX10. We build on the small-step, transition system for X10 presented in [24] which deals with `finish`, `async` and `for` loops. We extend it to handle multiple places and `at`, exceptions and try/catch statements, necessary to express place failure. (In the spirit of [24] we omit formalization of any of the object-oriented features of X10 since it is fairly routine). Configurations are just pairs $\langle s, g\rangle$ representing a statement s (the program to be executed) and a global heap g, a partial map from the set of places to heaps. Transitions are (potentially) labeled with exceptions, tagged with whether they were generated from a synchronous or asynchronous context. We establish desirable properties of the transition system (absence of stuck states, invariance of place-local heaps). We establish a bisimulation based semantics that is consistent with the intuitions

underlying the "gap based" trace set semantics of Brookes [8]. We establish a set of equational laws for this semantics.

On this foundation we show that the semantics of Resilient X10 can be formalized with just three kinds of changes. (1) A place failure transition models the failure of a place p by simply removing p from the domain of g. This cleanly models loss of all data at p. Next, the transition rules for various language constructs are modified to reflect what happens when those constructs are "executed" at a failed place. (2) An attempt to activate any statement at a failed place results in a `DeadPlaceException`. (3) Consistent with the design of Resilient X10, any synchronous exception thrown by (the dynamic version of) an `at(q) s` at a failed place `q` are masked by a `DPE`. These are the only changes needed.

We show that the main properties of TX10 carry over to Resilient TX10. We also show important resilience-related properties: our main theorem establishes that in fact Resilient TX10 satisfies Happens Before Invariance. We also present a set of equational laws and discuss differences with the laws for TX10.

We have encoded a mechanized version of the syntax and semantics of both TX10 and Resilient TX10 in Coq, an interactive theorem prover [4]. In doing so we addressed the challenge of formalizing the copy operation on heaps and establishing termination (even in the presence of cycles in the object graph). We mechanize the proof that there are no stuck configurations, and furthermore prove that the relation is computable, yielding a verified interpreter for TX10 and Resilient TX10.

Related work. Our work is related to three broad streams of work. The first is formalization of X10 and Java with RMI. The first formalization of X10 was in [21]. This paper adapts the framework of Middleweight Java [5] to represent a configuration as a collection of stacks and heaps. This choice led to a rather complex formalization. [17] presents an operational semantics for the X10 `finish`/`async` fragment, but again with a complex representation of control. We build on the work of [24] which for the first time represents the control state as a statement, and presents a very simple definition of the Happens Before relation. We extend that work to handle exceptions (necessary for the formalization of resilience), and place-shifting `at`, and formally treat resilience. [1] presents a semantics for Java with remote method invocation; hence they also deal with multiple places and communication across places. In particular they formalize a relational definition of copying an object graph, although they do not formalize or mechanize an implementation of this specification. Their formalization does not deal with place failure, since Java RMI does not deal with it.

The second stream is the work on formalization of the semantics of concurrent imperative languages [7,6,8]. Our work can be seen as adding block-structured concurrency constructs (`finish`, `async`), exceptions, and, of course, dealing with multiple places, and place failure.

The third stream is the work on distributed process algebras that deal with failure [14,16,15,3,19]. [3] introduces an extension of the π-calculus with located actions, in the context of a higher-order, distributed programming language, Facile. [14] introduces locations in the distributed join calculus, mobility and the possibility of location failure, similar to our place failure. The failure of a location can be detected, allowing failure recovery. In the context of Dπ [16], an extension of the π-calculus with multiple places

and mobility, [15] gives a treatment of node- and link-failure. In relationship with all these works, this work differs in dealing with resilience in the context of distributed state, global references, mobile tasks with distributed termination detection (finish), and exceptions, and formalizing the HBI principle. Our work is motivated by formalizing a real resilient programming language, rather than working with abstract calculi.

Summary of Contributions. The contributions of this paper are:

- We present a formal operational semantics for TX10, a significant fragment of X10, including multiple places, mutable heap, `try/catch` statements, `throws`, `async`, `finish` and `at` statements. The semantics is defined in terms of a labeled transition relation over configurations in which the control state is represented merely as a statement, and the data state as a mapping from places to heaps.
- We present a set of equational laws for operational congruence.
- We extend the formal operational semantics to Resilient TX10, showing that it enjoys Happens Before Invariance and Exception Masking Principles.
- We present equational laws for Resilient X10.
- We mechanize proofs of various propositions in Coq. More precisely, all the the proofs in the paper have been mechanized but for Theorem 4.8, Theorem 4.10, Theorem 3.3 and the equational laws, which have been proved manually. In particular, the mechanization of the proof that no configurations are stuck yields a verified executable version of the semantics.

Rest of this paper. Section 2 introduces TX10, informally describing the basic constructs and a small-step operational semantics of TX10. Section 3 presents laws for equality for a semantics built on congruence over bisimulation. The second half of the paper presents a semantic treatment of resilience. Section 4 discusses the design of Resilient TX10. formalizes this semantics using the idea of rewriting the control state to represent place failure. Finally, presents equational laws for congruence, and Section 5 concludes.

2 TX10

We describe in this section the syntax and the semantics of TX10, the formal subset of the X10 language [20] we consider in this work. We have also encoded a mechanized version in Coq, which will be discussed in Section 2.2.

The syntax of TX10 is defined in Table 1. We assume an infinite set of values Val, ranged over by v, w, an infinite set of variables ranged over by x, y, and an infinite set of field names ranged over by f. We also let p, q range over a finite set of integers $\mathsf{Pl} = 0...(n-1)$, which represent available computation *places*. A source program is defined as a static statement s activated at place 0 under a governing `finish` construct. The syntax then includes dynamic statements and dynamic values that can only appear at runtime. Programs operate over objects, either local or global, that are handled through object identifiers (object ids). We assume an infinite set of object ids, ObjId (with a given bijection with the natural numbers, the "enumeration order"); objects are in a one to one correspondence with object ids. Given the distributed nature of the language and

(Values) v, w	::=
o	*(Runtime only.) Object ids*
$o\$p$	*(Runtime only.) Global Object ids*
$\mathsf{E}, \mathsf{BF}, \mathsf{BG}, \mathsf{DP}$	*Exceptions*

(Programs) pr	::=
`finish at` (0) s	*activation*

(Expressions) d, e	::=
v	*Values*
x	*Variable access*
$e.f$	*Field selection*
$\{f{:}e, \ldots, f{:}e\}$	*Object construction*
`globalref` e	*GlobalRef construction*
`valof` e	*Global ref deconstruction*

(Statements) s, t	::=
`skip;`	*Skip – do nothing*
`throw` v;	*Throw an exception*
`val` $x = e$ s	*Let bind e to x in s*
$e.f = e$;	*Assign to field*
$\{s\ t\}$	*Run s then t*
`at`(p)`(val` $x = e)$ s	*Run s at p with x bound to e*
`async` s	*Spawn s in a different task*
`finish` s	*Run s and wait for termination*
`try` s `catch` t	*Try s, on failure execute t*
z	*Runtime versions*

(Dynamic Stmts) z	::=
$\overline{\texttt{at}}\ (p)\ s$	*Runtime only*
$\overline{\texttt{async}}\ s$	*Runtime only*
$\texttt{finish}_\mu\ s$	*Run s, recording exceptions in* μ

Table 1. Syntax of TX10

to model X10's global references, we assume that each object lives in a specific (home) place, and we distinguish between local and global references, denoted by o and $o\$q$. More precisely, we use the following notation:

- $\mathsf{p} : \mathsf{ObjId} \rightarrow \mathsf{Pl}$ maps each object id to the place where it lives;
- $\mathsf{ObjId}_q = \{o \in \mathsf{ObjId} \mid \mathsf{p}(o) = q\}$ and $\mathsf{grObjId} = \{o\$p \mid o \in \mathsf{ObjId}_p \ \wedge\ p \in \mathsf{Pl}\}$

Then given $o \in \mathsf{ObjId}_q$, we say that o is a local reference (to a local object) while $o\$q$ is a global reference (to an object located at q).

The expression $\{f_1 : e_1, \ldots, f_n : e_n\}$ (for $n \geq 0$) creates a new local object and returns its fresh id. The object is initialized by setting, in turn, the fields f_i to the value obtained by evaluating e_i. Local objects support field selection: the expression $e.f$ evaluates to the value of the field with name f in the object whose id is obtained by evaluating e. Similarly, the syntax of statements allows field update. X10 relies on a type system to ensure that any selection/update operation occurring at runtime is performed on an object that actually contains the selected/updated field. Since TX10 has no corresponding static semantic rules, we shall specify that $o.f$ throws a *BadFieldSelection* BF exception when the object o does not have field f.

The expression `globalref e` creates a new global reference for the reference returned by the evaluation of `e`. Whenever `e` evaluates to a global reference, the expression `valof e` returns the local object pointed by `e`. Errors in dealing with global

references are modelled by throwing a *BadGlobalRef* exception BG. (see Section 2.1 for a detailed explanation of the semantics of global references).

TX10 deals with exception handling in a standard way: the statement $\texttt{throw}\, v$ throws an exception value v that can be caught with a $\texttt{try}\, s\, \texttt{catch}\, t$ statement. For simplicity, exception values are constants: besides BF and BG described above, we add E to represent a generic exception. The exception DP stands for *DeadPlaceException*, and will only appear in the semantics of the resilient calculus in Section 4. Variable declaration $\texttt{val}\, x = e\ s$ declares a new local variable x, binds it to the value of the expression e and continues as s. The value assigned to x cannot be changed during the computation. We shall assume that the only free variable of s is x and that s does not contain a sub-statement that declares the same variable x.This statement is a variant of the variable declaration available in X10. In X10 the scope s is not marked explicitly; rather all statements in the rest of the current block are in scope of the declaration. We have chosen this "let" variant to simplify the formal presentation.

The construct $\texttt{async}\, s$ spawns an independent lightweight thread, called *activity*, to execute s. The new activity running in parallel is represented by the dynamic statement $\overline{\texttt{async}}\, s$. The statement $\texttt{finish}\ s$ executes s and waits for the termination of all the activities (recursively) spawned during this execution. Activities may terminate either normally or abruptly, *i.e.* by throwing an exception. If one or more activities terminated abruptly, $\texttt{finish}\ s$ will itself throw an exception that encapsulates all exceptions. In TX10, we use the parameter μ in $\texttt{finish}_\mu$ s to record the exception values thrown by activities in s. μ is a possibly empty set of values; we simply write $\texttt{finish}\ s$ instead of $\texttt{finish}_\emptyset\ s$.

The sequence statement $\{s\, t\}$ executes t after executing s. Note that if s is an async, its execution will simply spawn an activity $\overline{\texttt{async}}\, s$, and then activates t. Therefore, $\{\overline{\texttt{async}}\, s\ \ t\}$ will actually represent s and t executing in parallel. We say that sequencing in X10 has *shallow* $\texttt{finish}$ *semantics*.

Finally, $\texttt{at}(p)(\texttt{val}\, x = e)\, s$ is the place-shifting statement. We assume that the only free variable in s is x. This statement first evaluates e to a value v, then copies the object graph rooted at v to place p to obtain a value v', and finally executes s synchronously at p with x bound to v'. Running s at p synchronously means that in $\{\texttt{at}(p)(\texttt{val}\, x = e)\, s\ \ t\}$, t will be enabled precisely when the at statement has only asynchronous sub-statements left (if any). Thus at also has shallow finish semantics, just like sequential composition. In the cases when the programmer does not need to transmit values from the calling environment to s, the variant $\texttt{at}\ (p)\ s$ may be used instead. As an example, the program $\texttt{finish}\, \texttt{at}(0)\, \{\texttt{at}(1)\, \texttt{async}\, s\ \ \texttt{at}(2)\, \texttt{async}\, s\}$ evolves to a state where two copies of s run in parallel at places 1 and 2. The entire program terminates whenever both remote computations end.

Currently, X10 supports a variant of these at constructs. The programmer writes $\texttt{at}\ (p)\ s$ and the compiler figures out the set of variables used in s and declared outside s. A copy is made of the object reference graph with the values of these variables as roots, and s is executed with these roots bound to this copied graph. Moreover X10, of course, permits mutually recursive procedure (method) definitions. We leave the treatment of recursion as future work.

2.1 Operational Semantics

We build on the semantics for X10 presented in [24]. In this semantics, the data state is maintained in a shared global heap (one heap per place), but the control state is represented in a block structured manner – it is simply a statement.

$$\textit{Heap}\ \ h\ ::=\ \emptyset \mid h \cdot [o \mapsto r] \qquad \textit{Global heap}\ \ g\ ::=\ \emptyset \mid g \cdot [p \mapsto h]$$

The local heap at a place p is a partial map that associates object ids to objects represented by partial maps r from field names to object ids. The global heap g is a partial map form the set of places Pl to local heaps. Heaps are inductively defined with the operator $\cdot$ used to append a new entry. We let $\emptyset$ denote the unique partial map with empty domain, and for any partial map f by $f[p \to v]$ we mean the map f' that is the same as f except that it takes on the value v at p. Moreover, in the following we write $s[{}^{v}/_{x}]$ for variable substitution.

X10 is designed so that at run-time heaps satisfy the *place-locality invariant* formalized below. Intuitively, the domain of any local heap only contains local object references, moreover any object graph (rooted at a local object) only contains references to either (well defined) local objects or global references.

Let h be a local heap and $o \in dom(h)$ an object identifier. We let $h{\downarrow_o}$ denote *the object graph rooted at o*, that is the graph with vertexes the values reachable from o via the fields of o or of one or more intermediaries. In other terms, it is the graph where an f-labelled edge (v, f, v') connects the vertices v and v' whenever v is an object with a field f whose value is v'. We also denote by h_o the set of all object values that are reachable from o, that is the set of all vertices in the object graph $h{\downarrow_o}$.

Definition 2.1 (Place-local heap). *A global heap g is* place-local *whenever for every $q \in dom(g)$, and $h = g(q)$*

- $dom(h) \subseteq \mathsf{ObjId}_q$ *and* $\forall o \in dom(h).\ \ h_o \subseteq (\mathsf{ObjId}_q \cap dom(h)) \cup \mathsf{grObjId}$

The semantics is given in terms of a transition relation between *configurations*, which are either a pair $\langle s, g\rangle$ (representing the statement s to be executed in global heap g) or a singleton g, representing a computation that has terminated in g. Let k range over configurations. The transition relation $k \xrightarrow{\lambda}_p k'$ is defined as a labeled binary relation on configurations, where $\lambda \in \Lambda = \{\epsilon, v\times, v\otimes\}$, and p ranges over the set of places. The transition $k \xrightarrow{\lambda}_p k'$ is to be understood as: the configuration k executing at p can in one step evolve to k', with $\lambda = \epsilon$ indicating a normal transition, and $\lambda = v\otimes$, resp. $v\times$, indicating that an exception has thrown a value v in a synchronous, resp. asynchronous, subcontext. Note that failure is not fatal; a failed transition may be followed by any number of failed or normal transitions. We shall write $\xrightarrow{\epsilon}_p$ as $\longrightarrow_p$, and we let $\xrightarrow{\star}$ represent the reflexive, transitive closure of $\xrightarrow{\lambda}_0$.

Definition 2.2 (Semantics). *The operational semantics $\mathcal{O}[\![s]\!]$ of a statement s is the relation* $\mathcal{O}[\![s]\!] \stackrel{def}{=} \{(g, g') \mid \langle \texttt{finish}\ \overline{\texttt{at}}\,(0)\ s, g\rangle \xrightarrow{\star} g'\}$.

$$\vdash \mathsf{isAsync}\ \overline{\texttt{async}}\, s \qquad \frac{\vdash \mathsf{isAsync}\ s}{\begin{array}{l}\vdash \mathsf{isAsync}\ \overline{\texttt{at}}\,(p)\, s \\ \vdash \mathsf{isAsync}\ \texttt{try}\, s\, \texttt{catch}\, t\end{array}} \qquad \frac{\vdash \mathsf{isAsync}\ s \qquad \vdash \mathsf{isAsync}\ t}{\vdash \mathsf{isAsync}\ \{s\ t\}}$$

$$\vdash \mathsf{isSync}\ s^* \quad \text{with } s^* \in \left\{ \begin{array}{l} \texttt{skip},\ \texttt{val x=e s},\ e.f = e, \\ \texttt{at}(p)(\texttt{val}\, x = e)\, s,\ \texttt{async}\, s, \\ \texttt{finish}_{\,\mu}\, s, \texttt{throw}\, v \end{array} \right\} \qquad \frac{\vdash \mathsf{isSync}\ s}{\begin{array}{l}\vdash \mathsf{isSync}\ \{s\ t\} \\ \vdash \mathsf{isSync}\ \{t\ s\} \\ \vdash \mathsf{isSync}\ \overline{\texttt{at}}\,(p)\, s \\ \vdash \mathsf{isSync}\ \texttt{try}\, s\, \texttt{catch}\, t\end{array}}$$

Table 2. Synchronous and Asynchronous Statements

In order to present rules compactly, we use the "matrix" convention exemplified below, where we write the left-most rule to compactly denote the four rules obtained from the right-most rule with $i = 0, 1, j = 0, 1$.

$$\frac{\gamma \xrightarrow{\lambda} \gamma_0 \mid \gamma_1}{\begin{array}{l} cond_0 \ \ \delta^0 \xrightarrow{\lambda_0} \delta^0_0 \mid \delta^0_1 \\ cond_1 \ \ \delta^1 \xrightarrow{\lambda_1} \delta^1_0 \mid \delta^1_1 \end{array}} \qquad \frac{\gamma \xrightarrow{\lambda} \gamma_i \quad cond_j}{\delta^j \xrightarrow{\lambda_j} \delta^j_i} \qquad i = 0, 1\ j = 0, 1$$

We also introduce in Table 2 two auxiliary predicates to distinguish between *asynchronous* and *synchronous* statements. A statement is asynchronous if it is an $\overline{\texttt{async}}\, s$, or a sequential composition of asynchronous statements (possibly running at other places). The following proposition is easily established by structural induction.

Proposition 2.3. *For any statement s, either* $\vdash \mathsf{isAsync}\ s$ *xor* $\vdash \mathsf{isSync}\ s$.

In order to define the transition between configurations, we first define the evaluation relation for expressions by the rules in Table 3. Transitions of the form $\langle e, h\rangle \longrightarrow_p \langle e', h'\rangle$ state that the expression e at place p with local heap h correctly evaluates to e' with heap h'. On the other hand an error in the evaluation of e is modeled by the transition $\langle e, h\rangle \xrightarrow{v\otimes}_p h$. An object creation expression is evaluated from left to right, according to rule (EXP CTX). When all expressions are evaluated, rule (NEW OBJ) states that a new local object id is created and its fields set appropriately. Rule (NEW GLOBAL REF) shows that a new global reference is built from an object id o by means of the expression `globalref` o. A global reference $o\$p$ can be dereferenced by means of the `valof` expression. Notice that rule (VALOF), according to X10's semantics, shows that the actual object can only be accessed from its home place, i.e. $\mathsf{p}(o) = p$. Any attempt to select a non-existing field from an object results in the BF exception by rule (SELECT BAD), while any attempt to access a global object that is not locally defined results in a BG error by rule (VALOF BAD). In X10, the static semantics guarantees that objects and global references are correctly created and that any attempt to select a field is type

(NEW OBJ)

$$\frac{o \in \mathsf{ObjId}_p \backslash dom(h) \qquad n \geq 0}{\langle \{f_1{:}v_1, ..., f_n{:}v_n\}, h\rangle \longrightarrow_p \langle o, h \cdot [o \mapsto \emptyset[f_1 \mapsto v_1] \ldots [f_n \mapsto v_n]]\rangle}$$

(SELECT)

$$\frac{h(o){=}r[f \mapsto v]}{\langle o.f, h\rangle \longrightarrow_p \langle v, h\rangle}$$

(SELECT BAD)

$$\frac{v{\neq}o \vee (v{=}o \wedge f{\notin}dom(h(o)))}{\langle v.f, h\rangle \xrightarrow{\mathsf{BF}\otimes}_p h}$$

(NEW GLOBAL REF)

$$\frac{}{\langle \texttt{globalref}\ o, h\rangle \longrightarrow_p \langle o\$p, h\rangle}$$

(VALOF)

$$\frac{}{\langle \texttt{valof}\ o\$p, h\rangle \longrightarrow_p \langle o, h\rangle}$$

(BAD GLOBALREF)

$$\frac{}{\begin{array}{l} v \neq o\$p\ \langle \texttt{valof}\ v, h\rangle \xrightarrow{\mathsf{BG}\otimes}_p h \\ v \neq o \quad \langle \texttt{globalref}\ v, h\rangle \xrightarrow{\mathsf{BG}\otimes}_p h \end{array}}$$

(EXP CTX)

$$\frac{\langle e, h\rangle \xrightarrow{\lambda}_p \langle e', h'\rangle \mid h}{\begin{array}{c} \langle e.f, h\rangle \xrightarrow{\lambda}_p \langle e'.f, h'\rangle \mid h \\ \langle \texttt{globalref}\ e, h\rangle \xrightarrow{\lambda}_p \langle \texttt{globalref}\ e', h'\rangle \mid h \\ \langle \texttt{valof}\ e, h\rangle \xrightarrow{\lambda}_p \langle \texttt{valof}\ e', h'\rangle \mid h \\ \langle \{f_1{:}v_1, ..., f_i{:}v_i, f_{i+1}{:}e, ...\}, h\rangle \xrightarrow{\lambda}_p \langle \{f_1{:}v_1, ..., f_i{:}v_i, f_{i+1}{:}e', ...\}, h'\rangle \mid h \end{array}}$$

Table 3. Expression Evaluation

safe, hence well typed X10 programs do not occur in BF and BG exceptions, however we introduce rules (SELECT BAD), (VALOF BAD) and (BAD FIELD UPDATE) so that the operational semantics of TX10 enjoys the property that there are no stuck states, i.e. Proposition 2.10 in Section 2.3.

The following proposition shows that the heap modifications performed by rules (NEW OBJ) and (NEW GLOBAL REF) respect the place-locality invariant.

Proposition 2.4. *Let g be a place-local heap, $p \in dom(g)$ and $h = g(p)$. We say that $\langle e, h\rangle$ is place-local whenever for any local object id o occurring in e it holds $o \in dom(h)$. If $\langle e, h\rangle$ is place-local and $\langle e, h\rangle \longrightarrow_p \langle e', h'\rangle$, then $g \cdot [p \mapsto h']$ is place-local, and $\langle e', h'\rangle$ is place-local.*

Now we turn to the axiomatization of the transition relation between configurations. Table 4 collects a first set of rules dealing with basic statements. These rules use the condition $p \in dom(g)$, which is always true in TX10 since places do not fail. We include this condition to let Table 4 to be reused when we consider place failure in Section 4. Most of these rules are straightforward. Rule (EXCEPTION) shows that throwing an exception is recorded as a synchronous failure. Moreover, rule (BAD FIELD UPDATE) throws a BF exception whenever f is not one of its fields.

The rest of operational rules are collected in Table 5. These rules, besides defining the behavior of the major X10 constructs, also illustrate how the exceptions are prop-

$$\text{(SKIP)}\ \frac{p \in dom(g)}{\langle \texttt{skip}, g\rangle \longrightarrow_p g} \qquad \text{(EXCEPTION)}\ \frac{p \in dom(g)}{\langle \texttt{throw}\, v, g\rangle \xrightarrow{v\otimes}_p g} \qquad \text{(FIELD UPDATE)}\ \frac{p \in dom(g) \quad f \in dom(g(p)(o))}{\langle o.f{=}v, g\rangle \longrightarrow_p g[p \to g(p)[o \to g(p)(o)[f{\mapsto}v]]]}$$

$$\text{(DECLARE VAL)}\ \frac{p \in dom(g) \quad \langle s[^{v}/_{x}], g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\langle \texttt{val}\, x = v\ s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'} \qquad \text{(BAD FIELD UPDATE)}\ \frac{p{\in}dom(g) \quad (v{\neq}o \vee (v{=}o \wedge f{\notin}dom(g(p)(o))))}{\langle v.f = v', g\rangle \xrightarrow{\mathsf{BF}\otimes}_p g}$$

$$\text{(CTX)}\ \frac{p \in dom(g) \quad \langle e, g(p)\rangle \xrightarrow{\lambda}_p \langle e', h'\rangle \mid h' \quad g' = g[p \mapsto h']}{\begin{array}{c}\langle \texttt{val}\, x = e\ s, g\rangle \xrightarrow{\lambda}_p \langle \texttt{val}\, x = e'\ s,\ g'\rangle \mid g' \\ \langle e.f = e_1, g\rangle \xrightarrow{\lambda}_p \langle e'.f = e_1,\ g']\rangle \mid g' \\ \langle o.f = e, g\rangle \xrightarrow{\lambda}_p \langle o.f = e',\ g'\rangle \mid g' \\ \langle \texttt{at}(p)(\texttt{val}\, x = e)\, s, g\rangle \xrightarrow{\lambda}_p \langle \texttt{at}(p)(\texttt{val}\, x = e')\, s,\ g'\rangle \mid g'\end{array}}$$

Table 4. Basic Statements

agated through the system and possibly caught. In words, *synchronous failures* arise from synchronous statements, and lead to the failure of any synchronous continuation, while leaving (possibly remote) asynchronous activities that are running in parallel free to correctly terminate (cf. Proposition 2.11). On the other hand, *asynchronous failures* arise when an exception is raised in a parallel thread. In this case the exception is confined within that thread, and it is caught by the closest `finish` construct that is waiting for the termination of this thread. On termination of all spawned activities, since one (or more) asynchronous exception were caught, the `finish` constructs re-throws a synchronous failure (cf. Proposition 2.12).

Let us precisely discuss the rules in Table 5. The `async` construct takes one step to spawn the new activity by means of the rule (SPAWN). Moreover, according to rule (ASYNC), an exception (either synchronous or asynchronous) in the execution of s is *masked* by an asynchronous exception in $\overline{\texttt{async}}\, s$. We let $\mathsf{MskAs}(\lambda)$ be the label λ where we highlight the fact that an exception masking has occurred. The `finish` s statement waits for the termination of any (possibly remote) asynchronous (and synchronous as well) activities spawned by s. Any exception thrown during the evaluation of s is absorbed and recorded into the state of the governing `finish`. Indeed, consider rule (FINISH) where we let be $\mu \cup \lambda{=}\mu$ if $\lambda{=}\epsilon$ and $\mu \cup \lambda{=}\{v\} \cup \mu$ if $\lambda{=}v\times$ or $\lambda{=}v\otimes$. Then this rule shows that the consequence has a correct transition $\longrightarrow_p$ even when $\lambda \neq \epsilon$: i.e., the exception in s has been absorbed and recorded into the state of `finish`. Moreover, the rule (END OF FINISH) shows that `finish` terminates with a generic synchronous exception whenever at least one of the activities its governs threw an exception (in X10 it throws a `MutipleExceptions` containing the list of exceptions collected by `finish`).Two rules describe the semantics of sequential com-

(SPAWN)

$$\frac{}{\langle \texttt{async}\ s, g\rangle \longrightarrow_p \langle \overline{\texttt{async}}\ s, g\rangle}$$

(ASYNC)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\begin{array}{ll} \lambda{=}\epsilon & \langle \overline{\texttt{async}}\ s, g\rangle \longrightarrow_p \langle \overline{\texttt{async}}\ s', g'\rangle \mid g' \\ \lambda{=}v\times, v\otimes & \langle \overline{\texttt{async}}\ s, g\rangle \xrightarrow{\mathsf{MskAs}(v\times)}_p \langle \overline{\texttt{async}}\ s', g'\rangle \mid g' \end{array}}$$

(FINISH)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle}{\langle \texttt{finish}_\mu\, s, g\rangle \longrightarrow_p \langle \texttt{finish}_{\mu\cup\lambda}\, s', g'\rangle}$$

(END OF FINISH)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p g' \qquad \lambda' = \begin{cases} \epsilon & \text{if } \lambda\cup\mu{=}\emptyset \\ \mathsf{MskAs}(\mathsf{E}\otimes) & \text{if } \lambda\cup\mu{\neq}\emptyset \end{cases}}{\langle \texttt{finish}_\mu\, s, g\rangle \xrightarrow{\lambda'}_p g'}$$

(SEQ)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\begin{array}{ll} \lambda = \epsilon, v\times & \langle \{s\ t\}, g\rangle \xrightarrow{\lambda}_p \langle \{s'\ t\}, g'\rangle \mid \langle t, g'\rangle \\ \lambda = v\otimes & \langle \{s\ t\}, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g' \end{array}}$$

(PAR)

$$\frac{\vdash \mathsf{isAsync}\ t \quad \langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\langle \{t\ s\}, g\rangle \xrightarrow{\lambda}_p \langle \{t\ s'\}, g'\rangle \mid \langle t, g'\rangle}$$

(PLACE SHIFT)

$$\frac{(v', g') = \mathsf{copy}(v, q, g)}{\langle \texttt{at}(q)(\texttt{val}\ x{=}v)\, s, g\rangle \longrightarrow_p \langle \overline{\texttt{at}}\,(q)\,\{s[^{v'}/_x]\ \texttt{skip}\}, g'\rangle}$$

(AT)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_q \langle s', g'\rangle \mid g'}{\langle \overline{\texttt{at}}\,(q)\, s, g\rangle \xrightarrow{\lambda}_p \langle \overline{\texttt{at}}\,(q)\, s', g'\rangle \mid g'}$$

(TRY)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\begin{array}{ll} \lambda = \epsilon, v\times & \langle \texttt{try}\, s\, \texttt{catch}\, t, g\rangle \xrightarrow{\lambda}_p \langle \texttt{try}\, s'\, \texttt{catch}\, t, g'\rangle \mid g' \\ \lambda = v\otimes & \langle \texttt{try}\, s\, \texttt{catch}\, t, g\rangle \longrightarrow_p \langle \{s'\ t\}, g'\rangle \mid \langle t, g'\rangle \end{array}}$$

Table 5. Statements Semantics

position. When executing $\{s\ t\}$, rule (SEQ) shows that the continuation t is activated whenever s terminates normally or with an asynchronous exception. On the other hand, when the execution of s throws a synchronous exception (possibly leaving behind residual statements s') the continuation t is discarded. Rule (PAR) captures the essence of asynchronous execution allowing reductions to occur in parallel components.

The rule (PLACE SHIFT) activates a remote computation; it uses a *copy* operation on object graphs, $\mathsf{copy}(o, q, g)$, that creates at place q a copy of the object graph rooted at o, respecting global references. In X10 place shift is implemented by recursively serializing the object reference graph G rooted at o into a byte array. In this process, when it is encountered a global object reference $o\$p$, the fields of this object are not followed; instead the unique identifier $o\$p$ is serialized. The byte array is then transported to q, and de-serialized at q to create a copy G' of G with root object a fresh identifier $o' \in \mathsf{ObjId}_q$. All the objects in G' are new. G' is isomorphic to G and has the additional

property that if z is a global ref that is reachable from o then it is also reachable (through the same path) from o'.

Definition 2.5 (The copy operation.). *Let g be a global heap, q a place with $h = g(q)$. Let be $o \in \mathsf{ObjId}$ such that $\mathsf{p}(o) \in dom(g)$, then $\mathsf{copy}(o, q, g)$ stands for the (unique) tuple $\langle o', g[q \to h'] \rangle$ satisfying the following properties, where $N = dom(h') \setminus dom(h)$.*

- *N is the next $|N|$ elements of ObjId_q.*
- *$o' \in N$*
- *There is an isomorphism ι between the object graph $g(\mathsf{p}(o))\downarrow_o$ rooted at o and the object graph $h'\downarrow_{o'}$ rooted at o'. Further, $\iota(v) = v$ for $v \in \mathsf{grObjId}$*
- *$h'_{o'} \subseteq N \cup \mathsf{grObjId}$.*
- *$h' = h \cdot [o' \mapsto r]$ where r is the root object of the graph $h'\downarrow_{o'}$*

We extend this definition to arbitrary values, that is $\mathsf{copy}(v, q, g)$ is defined to be v unless v is an object id, in which case it is defined as above.

Proposition 2.6. *Let g be a place-local heap. Let $p, q \in dom(g)$ be two (not necessarily distinct) places, and let $o \in \mathsf{ObjId}_p$. Let $\mathsf{copy}(o, q, g) = \langle o', g' \rangle$. Then g' is place-local.*

Place-shift takes a step to activate. Moreover, in the conclusion of the rule (PLACE SHIFT) the target statement contains a final `skip` in order to model the fact that the remote control has to come back at the local place after executing the remote code $s[^{v'}/_{x'}]$. As an example, consider $\{\overline{\texttt{at}}\,(p)\,\{\overline{\texttt{async}}\,s\;\texttt{skip}\}\;\;t\}$ and $\{\overline{\texttt{at}}\,(p)\,\{\overline{\texttt{async}}\,s\}\;\;t\}$. The local code t is already active only in the second statement while in the first one it is waiting for the termination of the synchronous remote statement. Accordingly, the second program models the situation where the control has come back locally after installing the remote asynchronous computation. Modeling this additional step is actually relevant just in the resilient calculus, where we need to model the case where the remote place precisely fails after executing s but before the control has come back. Indeed, consider $\{\overline{\texttt{at}}\,(p)\,\{\overline{\texttt{async}}\,s\;\texttt{skip}\}\;\;t\}$ and $\{\overline{\texttt{at}}\,(p)\,\{\overline{\texttt{async}}\,s\}\;\;t\}$. The local code t is already active only in the second statement while in the first one it is waiting for the termination of the synchronous remote statement. Accordingly, the second statement models the situation where the control has come back locally after installing the remote asynchronous computation.

As for error propagation, by rule (AT) we have that any exception, either synchronous or asynchronous, that occurred remotely at place p is homomorphically reported locally at place r. As an example, consider $\overline{\texttt{at}}\,(r)\,\{\overline{\texttt{at}}\,(p)\,\texttt{throw}\,\mathsf{E}\;\;t\}$, then the exception at p terminates the remote computation and is reported at r as a synchronous error so that to also discard the local continuation t, whose execution depends on the completion of the remote code. In order to recover from remote exceptions, we can use the try-catch mechanism and write $\overline{\texttt{at}}\,(r)\,\{\texttt{try}\;(\overline{\texttt{at}}\,(p)\,\texttt{throw}\,\mathsf{E})\;\texttt{catch}\,t'\quad t\}$ so that the synchronous exception is caught at r according to the rule (TRY). More precisely, the $\texttt{try}\,s\,\texttt{catch}\,t$ statement immediately activates s. Moreover, the rule (TRY) shows that asynchronous exceptions are passed through, since they are only caught by `finish`. On the other hand, synchronous exceptions are absorbed into a correct transition and the `catch`-clause is activated, together with the (asynchronous) statements s' left behind by the failed s.

Example 2.7. Consider the two programs $s_1 = \overline{\texttt{at}}\,(p)\,\texttt{finish}\,\overline{\texttt{at}}\,(q)\,\overline{\texttt{async}}\,s$ and $s_2 = \texttt{finish}\,\overline{\texttt{at}}\,(p)\,\{\overline{\texttt{at}}\,(q)\,\overline{\texttt{async}}\,s\}$. In both programs the termination of s is detected by the `finish` construct, that is, at place p in s_1 and at place 0 in s_2. Moreover, if the execution of s at q throws an exception, we have that the asynchronous exception is also caught by the `finish` construct, that is it is caught at place p for s_1 and at place 0 for s_2. Such a difference is not observable in TX10, indeed we will provide in Section 3 an equational law (cf. law (24)) showing that s_1 and s_2 are observationally equivalent. On the other hand, we will see that in Resilient TX10 the two statements behave differently when places p and q are subject to failure. As a further example consider the programs $s_1' = \{s_1\ s'\}$ and $s_2' = \texttt{finish}\,\{\overline{\texttt{at}}\,(p)\,\{\overline{\texttt{at}}\,(q)\,\overline{\texttt{async}}\,s\}\ \ s'\}$. In s_1' we have that s' is executed at place 0 after the termination of s, while in s_2' we have that s' is executed at place 0 in parallel with s running at q. Moreover, let s throw an exception, then in s_1' we have that the `finish` at p re-throws a (masked) synchronous exception that discards the continuation s', while in s_2' we have that s' correctly terminates since the asynchronous exception is captured by the outer `finish`.

2.2 Mechanization in Coq

We have encoded the syntax and semantics of TX10 in Coq, an interactive theorem prover. Encoding the syntax and semantics are mostly straightforward, and closely follows the paper presentation. However, the mechanized formalism has a richer notion of exception propagation, which was omitted from the paper for compactness. Labels can carry a list of exceptions, allowing multiple exceptions to be propagated by `finish` (instead of using a single generic exception). Additionally, labels / exceptions can be any value type. This complicates the rules, since the (AT) rule needs to copy any values stored in the labels from the target heap to the caller's heap. This is done by the actual X10 language, and correctly modeled by our mechanized semantics.

The most challenging part of encoding the semantics is encoding the copy operation given in Definition 2.5, which copies an object graph from one heap to another.

Mechanizing the Copy Operation Definition 2.5 provides a declarative specification of the copy operation, asserting the existence of a satisfying function. The mechanization explicitly constructs this function. In particular, it provides a pure (provably terminating and side-effect free) function with the given specification.

We first encode definitions of (local) reachability and graph isomorphism, proving key theorems relating them. We also define what it means for a value to be *well-formed* in a given heap: all objects (locally) reachable from that value must be in the heap. In other words, the object graph rooted at the value may not contain dangling pointers.

The tricky part of implementing this algorithm in Coq is proving termination. This is not obvious, since there can be cycles in the object graph that we are copying. To prevent looping on such cycles, the implementation carefully maintains and uses the set of existing mappings from the source to the destination heap. To prove termination for a non-structurally recursive function, we define a well-founded measure that provably decreases on every recursive call. We omit details for lack of space.

As well as proving that the implementation is total, we also prove that it has the required specification. Moreover, if copy fails, there must exist some object id reachable

from the root that is not contained in the heap. This last part of the specification in turn enables us to prove that copy will always succeed if the initial value is well formed.

2.3 Properties of the transition relation

TX10 satisfies a number of useful properties, given below. We have mechanized all these proofs in Coq, using our encoding of TX10. This provides a high level of assurance in these proofs, and fills in the details of the various well-formedness conditions, such as place-locality, needed to ensure that the properties hold.

Definition 2.8 (Place-local Configuration). *Given a place-local heap g, we say that a configuration $\langle s, g\rangle$ is place-local if*

- *for any local object id o occurring in s under $\texttt{at}(p)$ or $\overline{\texttt{at}}\,(p)$, we have that $o \in dom(g(p))$ (hence $o \in \mathsf{ObjId}_p$ by place-locality of g), and*
- *for any global reference $o\$q$ occurring in s, we have that $o \in dom(g(q))$.*

Proposition 2.9 (Place-locality). *If $\langle s, g\rangle$ is a place-local configuration and $\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'$, then $\langle s', g'\rangle$ is a place-local configuration, resp. g' is a place-local heap.*

Proposition 2.10 (Absence of stuck states). *If a configuration k is terminal then k is of the form g.*

The mechanized proof of Proposition 2.10 additionally proves that the evaluation relation is computable: if the configuration is not terminal, we can always compute a next step. This is of course not the only step, since the relation is non-deterministic. Similarly, we prove that the transitive closure of the evaluation relation does not get stuck and is computable. This proof can be "run", yielding a simple interpreter for TX10.

The following propositions deal with error propagation. Proposition 2.11 shows that *synchronous failures* arise from synchronous statements; they entail the discard of any synchronous continuation, while leaving (possibly remote) asynchronous activities running in parallel free to correctly terminate. On the other hand, Proposition 2.12 shows that *asynchronous failures* are caught by the closest `finish` construct that is waiting for the termination of the thread where the failure arose. We rely on the following definition of *Evaluation Contexts*, that is contexts under which a reduction step is possible:

$$E ::= [\,] \mid \{E\; t\} \mid \{t\; E\}\ \mathsf{with}\ \vdash \mathsf{isAsync}\ t \mid \overline{\texttt{at}}\,(p)\, E$$
$$\mid \overline{\texttt{async}}\, E \mid \texttt{finish}_\mu\ E \mid \texttt{try}\, E\, \texttt{catch}\, t$$

Proposition 2.11 (Synchronous Failures). *If $\langle s, g\rangle \xrightarrow{v\otimes}_p k$ then $\vdash \mathsf{isSync}\ s$. Moreover, if $k \equiv \langle s', g'\rangle$, then $\vdash \mathsf{isAsync}\ s'$.*

Proposition 2.12 (Asynchronous Failures).

- *If $\langle s, g\rangle \xrightarrow{v\times}_p k$ then there exists an evaluation context $E[\,]$ such that $s = E[s_1]$ with $\langle s_1, g\rangle \xrightarrow{v\times}_p k'$ and $\vdash \mathsf{isAsync}\ s_1$.*

– *If* $\langle \texttt{finish}_\mu\ s, g\rangle \xrightarrow{\lambda_1}_p \ldots \xrightarrow{\lambda_n}_p g$ *because of* $\langle s, g\rangle \xrightarrow{\lambda'_1}_p \ldots \xrightarrow{\lambda'_n}_p g$*, then*
  1. $\lambda_i = \epsilon$ *for* $i = 1, \ldots, n-1$*, and*
  2. *either* $\lambda_n = \mathsf{E}\otimes$ *or* $\lambda_n = \epsilon$ *and* $\forall j = 1, \ldots, n\ \ \lambda'_j = \epsilon$*.*

The proofs of the propositions above easily follow by induction on the derivation of $\langle s, g\rangle \xrightarrow{v\otimes}_p k$, resp. $\langle s, g\rangle \xrightarrow{v\times}_p k$, and an inspection of the rules for `finish`.

Proposition 2.13. *Let be* $\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle$*, then if* $\vdash \mathsf{isAsync}\ s$ *then* $\vdash \mathsf{isAsync}\ s'$*, or equivalently, if* $\vdash \mathsf{isSync}\ s'$ *then* $\vdash \mathsf{isSync}\ s$*.*

3 Equivalence and Equational Laws

In this section we define a notion of equivalence for TX10 programs along the lines of [21]. We consider weak bisimulation defined on both normal transitions and transitions that throw an exception. Moreover, the bisimulation encodes the observation power of the concurrent context in two ways: (i) it preserves the $\mathsf{isSync}/\mathsf{isAsync}$ predicate and (ii) takes into account concurrent modification of shared memory. As a result, the equivalence turns out to be a congruence (cf. Theorem 3.3).

We use a notion of *environment move* to model the update of a shared heap by a concurrent activity. The store can be updated by updating a field of an existing object, by creating a new (local) object, or by means of a serialization triggered by a place shift.

Definition 3.1 (Environment move). *An environment move* Φ *is a map on global heaps satisfying:*

1. *if* g *is place-local, then* $\Phi(g)$ *is place-local,*
2. $dom(\Phi(g)) = dom(g)$*, and* $\forall p \in dom(g)\ dom(g(p)) \subseteq dom(\Phi(g)(p))$*.*

Let $(\longrightarrow_p)^*$ denote the reflexive and transitive closure of $\xrightarrow{\epsilon}_p$, that is any number (possibly zero) of ϵ-steps. Then we let $\overset{\lambda}{\Longrightarrow}_p$ stand for $(\longrightarrow_p)^* \xrightarrow{\lambda}_p (\longrightarrow_p)^*$ when $\lambda \neq \epsilon$, and $(\longrightarrow_p)^*$ if $\lambda = \epsilon$.

Definition 3.2 (Weak Bisimulation). *A binary relation* $\mathcal{R}$ *on closed configurations is a* weak bisimulation *if whenever*

1. $g\ \mathcal{R}\ k$ *then* $k = g$*,*
2. $\langle s, g\rangle\ \mathcal{R}\ k$ *then* $k = \langle t, g\rangle$ *for some* t*, and*
   – $\vdash \mathsf{isSync}\ s$ *if and only if* $\vdash \mathsf{isSync}\ t$ *and*
   – *for every environment move* Φ*, and for every place* p *it is the case that*
     (a) *if* $\langle s, \Phi(g)\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle$ *then for some* t'*,* $\langle t, \Phi(g)\rangle \overset{\lambda}{\Longrightarrow}_p \langle t', g'\rangle$ *and* $\langle s', g'\rangle\ \mathcal{R}\ \langle t', g'\rangle$*, and vice versa.*
     (b) *if* $\langle s, \Phi(g)\rangle \xrightarrow{\lambda}_p g'$ *then* $\langle t, \Phi(g)\rangle \overset{\lambda}{\Longrightarrow}_p g'$ *and vice versa.*

Two configurations are weak bisimilar, written $\langle s, g\rangle \equiv \langle t, g'\rangle$*, whenever there exists a weak bisimulation relating them. The weak bisimilarity is the largest weak bisimulation between configurations.*

Theorem 3.3. *Weak bisimilarity is a congruence.*

The theorem comes by a standard argument showing that the smallest congruence containing weak bisimilarity is a weak bisimulation. We illustrate the equivalence by means of a number of equational laws dealing with the main constructs of TX10. To ease the notation we write laws between statements rather than configurations. We start with laws for sequencing and asynchronus actvities:

$$\vdash \mathsf{isSync}\ s \qquad \{\texttt{skip};\ s\} \equiv s \qquad \{s\ \texttt{skip};\} \equiv s \tag{1}$$

$$\{\texttt{throw}\, v\ \ s\} \equiv \texttt{throw}\, v \tag{2}$$

$$\{\{s\, t\}\, u\} \equiv \{s\, \{t\, u\}\} \tag{3}$$

$$\vdash \mathsf{isAsync}\ s, \vdash \mathsf{isAsync}\ t \qquad \{s\, t\} \equiv \{t\, s\} \tag{4}$$

$$\texttt{async}\ \texttt{async}\ s \equiv \texttt{async}\ s \tag{5}$$

$$\overline{\texttt{async}}\, \texttt{skip} \not\equiv \texttt{skip} \qquad \overline{\texttt{async}}\, \texttt{throw}\, v \not\equiv \texttt{throw}\, v \tag{6}$$

$$\{\overline{\texttt{async}}\, \texttt{throw}\, v\ \ \overline{\texttt{async}}\, \texttt{throw}\, v\} \not\equiv \overline{\texttt{async}}\, \texttt{throw}\, v \tag{7}$$

Observe that (1) only hold for synchronous statements since both $\{\texttt{skip}\ s\}$ and $\{s\ \texttt{skip}\}$ are synchronous statements irrespective of s, hence the equivalence only holds when also the r.h.s. is synchronous. Laws (6) do not hold since only the l.h.s. are asynchronous. Law (7) does not hold since weak bisimilarity counts the number of (asynchronous) exceptions, and the l.h.s. throws two asynchronous $\mathsf{E}\times$ while the r.h.s. just one. Notice that by law (2) we have instead $\{\texttt{throw}\, v\ \texttt{throw}\, v\} \equiv \texttt{throw}\, v$, which is correct because the l.h.s. throws a single $\mathsf{E}\otimes$ since synchronous errors discard the continuation. The following set of laws deals with the try/catch construct:

$$\texttt{try}\, \texttt{skip}\, \texttt{catch}\, t \equiv \texttt{skip} \tag{8}$$

$$\vdash \mathsf{isSync}\ s \qquad \texttt{try}\, \texttt{throw}\, v\, \texttt{catch}\, s \equiv s \tag{9}$$

$$\texttt{try}\, s\, \texttt{catch}\, \texttt{throw}\, v \equiv s \tag{10}$$

$$\vdash \mathsf{isAsync}\ s \qquad \texttt{try}\, s\, \texttt{catch}\, u \equiv s \tag{11}$$

$$\vdash \mathsf{isAsync}\ s \qquad \texttt{try}\, \{s\, t\}\, \texttt{catch}\, u \equiv \{\texttt{try}\, s\, \texttt{catch}\, u\ \ \texttt{try}\, t\, \texttt{catch}\, u\} \tag{12}$$

$$\texttt{try}\, (\texttt{try}\, s\, \texttt{catch}\, t)\, \texttt{catch}\, u \equiv \texttt{try}\, s\, \texttt{catch}\, (\texttt{try}\, t\, \texttt{catch}\, u) \tag{13}$$

Notice that law (12) does not hold if s is a synchronous statement. Indeed, a synchronous error in s implies that in the l.h.s. the continuation t is discarded, while the execution of the r.h.s. might activate two copies of u when both s and t fail in sequence.

$$\texttt{at}\ (p)\, \texttt{skip} \equiv \texttt{skip} \tag{14}$$

$$\texttt{at}\ (p)\, \texttt{throw}\, v \equiv \texttt{throw}\, v \tag{15}$$

$$\texttt{at}\ (p)\, \{s\, t\} \equiv\ \{\texttt{at}\ (p)\, s\ \ \texttt{at}\ (p)\, t\} \tag{16}$$

$$\texttt{at}\ (p)\, \texttt{at}\ (q)\, s \equiv \texttt{at}\ (q)\, s \tag{17}$$

$$\texttt{async}\ \texttt{at}\ (p)\, s \equiv \texttt{at}\ (p)\, \texttt{async}\ s \tag{18}$$

$$\texttt{at}\ (p)\, (\texttt{try}\, s\, \texttt{catch}\, t) \equiv \texttt{try}\, (\texttt{at}\ (p)\, s)\, \texttt{catch}\, (\texttt{at}\ (p)\, t) \tag{19}$$

All the laws above for place shift also hold for the dynamic version of `at`. Finally, the following set of laws deal with the finish construct:

$$\texttt{finish skip} \equiv \texttt{skip} \tag{20}$$
$$\texttt{finish}\,\{s\ t\} \equiv \texttt{finish}\ s\ \ \texttt{finish}\ t \tag{21}$$
$$\texttt{finish async}\ s \equiv \texttt{finish}\ s \tag{22}$$
$$\texttt{finish}\,\{s\ \texttt{async}\ t\} \equiv \texttt{finish}\,\{s\ t\} \tag{23}$$
$$\texttt{finish at}\ (p)\ s \equiv \texttt{at}\ (p)\ \texttt{finish}\ s \tag{24}$$
$$\texttt{finish finish}\ s \equiv \texttt{finish}\ s \tag{25}$$

Notice that law (23) comes form (21) and (22). We conclude with a set of inequalities, where we write $\vdash$ noAsync s if s has no sub-term of the form `async` s' for some s', i.e., if s cannot evolve to an asynchronous statement.

$$\texttt{finish throw}\ v \not\equiv \texttt{throw}\ v \tag{26}$$
$$\texttt{finish}\,\{s\ \texttt{throw}\ v\} \not\equiv \{\texttt{finish}\ s\ \texttt{throw}\ v\} \tag{27}$$
$$(\vdash \mathsf{noAsync}\ s)\ \texttt{finish}\ s \not\equiv s \tag{28}$$
$$(\vdash \mathsf{noAsync}\ s)\ \texttt{finish try}\ s\ \texttt{catch}\ t \not\equiv \texttt{try}\ s\ \texttt{catch finish}\ t \tag{29}$$

All these laws do not hold because of the exception masking mechanism performed by the finish construct. For instance, in law (26) the exception $v\otimes$ thrown by `throw` v is masked in the l.h.s. by $\mathsf{E}\otimes$ by the finish construct.

4 Resilient TX10

The resilient calculus has the same syntax of TX10. We now assume that any place $p \in \mathsf{Pl}\backslash\{0\}$ can fail at any moment during the program computation. Place 0 has a special role: programs start at place zero, then this place is used to communicate the result to the user, so we assume it can never fail (if it does fail, the whole execution is torn down). In order to define the semantics, we now let global heaps g to be partial (rather than total) maps from places to local heaps. Intuitively, $dom(g)$ is the set of non-failed places. The semantics of Resilient TX10 is given by the rules in Table 3 and Table 4 from Section 2 plus the rules in Tables 6, 7 and 8 given in this section. More precisely, the resilient calculus inherits from TX10 the rules for expression evaluation (i.e., Table 3) and those in Table 4 which correspond to basic statement executed at non-failed place p, i.e. $p \in dom(g)$. The rules for TX10's main constructs, i.e. those in Table 5, hold also in the resilient calculus when $p \in dom(g)$, but they must be integrated with additional rules dealing with the case where the local place p has failed. Therefore, in order to improve the presentation, rather than inheriting Table 5, we collect here all the operational rules for the main constructs, compacting them in Tables 6, 7 and 8.

The place failure may occur at anytime, and it is modelled by the rule (PLACE FAILURE), which removes the failed place from the global heap. The semantics of TX10 is then extended according to the behaviour of Resilient X10 ([11]), that is so to ensure that after the failure of a place p:

1. any attempt to execute a statement at p results in a DP exception (Theorem 4.8);

(PLACE FAILURE)
$$\frac{p \in dom(g)}{\langle s, g\rangle \longrightarrow_p \langle s, g \setminus \{(p, g(p))\}\rangle}$$

(SPAWN)
$$\frac{}{\begin{array}{l} p \in dom(g)\ \langle \texttt{async}\ s, g\rangle \longrightarrow_p \langle \overline{\texttt{async}}\, s, g\rangle \\ p \notin dom(g)\ \langle \texttt{async}\ s, g\rangle \xrightarrow{\mathsf{DP}\otimes}_p g \end{array}}$$

(LOCAL FAILURE)
$$\frac{p \notin dom(g)}{\begin{array}{rl} \langle \texttt{skip}, g\rangle & \xrightarrow{\mathsf{DP}\otimes}_p g \\ \langle \texttt{throw}\, v, g\rangle & \xrightarrow{\mathsf{DP}\otimes}_p g \\ \langle \texttt{val}\, x = e\ s, g\rangle & \xrightarrow{\mathsf{DP}\otimes}_p g \\ \langle e_1.f = e_2, g\rangle & \xrightarrow{\mathsf{DP}\otimes}_p g \end{array}}$$

(ASYNC)
$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\begin{array}{ll} \lambda = \epsilon & \langle \overline{\texttt{async}}\, s, g\rangle \longrightarrow_p \langle \overline{\texttt{async}}\, s', g'\rangle \mid g' \\ \lambda = v\times, v\otimes & \langle \overline{\texttt{async}}\, s, g\rangle \xrightarrow{\mathsf{MskAs}(v\times)}_p \langle \overline{\texttt{async}}\, s', g'\rangle \mid g' \end{array}}$$

(FINISH)
$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle}{\langle \texttt{finish}_\mu\, s, g\rangle \longrightarrow_p \langle \texttt{finish}_{\mu\cup\lambda}\, s', g'\rangle}$$

(END OF FINISH)
$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p g' \quad \lambda' = \begin{cases} \epsilon & \text{if } \lambda\cup\mu = \emptyset \\ \mathsf{E}\otimes & \text{if } \lambda\cup\mu \neq \emptyset, p \in dom(g) \\ \mathsf{DP}\otimes & \text{if } \lambda\cup\mu \neq \emptyset, p \notin dom(g) \end{cases}}{\langle \texttt{finish}_\mu\ s, g\rangle \xrightarrow{\mathsf{MskAs}(\lambda')}_p g'}$$

Table 6. Resilient Semantics I

2. place shifts cannot be initiated form p nor launched to the failed p (rule (PLACE SHIFT));
3. any remote code that has been launched from p before its failure is not affected and it is free to correctly terminate its remote computation. If a synchronous exception escapes from this remote code and flows back to the failed place, then this exception is masked by a DP (Proposition 4.7) which is thrown back to a parent `finish` construct waiting at a non-failed place.

More precisely, we will show that the operational semantics of Resilient TX10 enforces the following three design principles:

1. ***Happens Before Invariance Principle:*** failure of a place q should not alter the happens before relationship between statement instances at places other than q.
2. ***Exception Masking Principle:*** failure of a place q will cause synchronous exceptions thrown by $\overline{\texttt{at}}\ (q)\ s$ statements to be masked by DP exceptions.
3. ***Failed Place Principle:*** at a failed place, activating any statement or evaluating any expression should result in a DP exception.

We now precisely describe the rules for the main constructs. The rule (LOCAL FAILURE) shows that no expression is evaluated at a failed place; any attempt to execute a basic statement at the failed place results in a synchronous DP exception. Similarly, rule (SPAWN) shows that new activities can only be spawned at non-failed places. On the other hand, rule (ASYNC) is independent form the failure of p, so that any remote computation contained in s proceeds not affected by the local failure. The semantics

(SEQ)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle}{\begin{array}{l}\lambda = \epsilon, v\times \ \langle \{s\ t\}, g\rangle \xrightarrow{\lambda}_p \langle \{s'\ t\}, g'\rangle \\ \lambda = v\otimes \quad \langle \{s\ t\}, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle\end{array}}$$

(PAR)

$$\frac{\vdash \mathsf{isAsync}\ t \quad \langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\langle \{t\ s\}, g\rangle \xrightarrow{\lambda}_p \langle \{t\ s'\}, g'\rangle \mid \langle t, g'\rangle}$$

(SEQ TERM)

$$\frac{p \in dom(g) \quad \langle s, g\rangle \xrightarrow{\lambda}_p g'}{\begin{array}{l}\lambda = \epsilon, v\times \ \langle \{s\ t\}, g\rangle \xrightarrow{\lambda}_p \langle t, g'\rangle \\ \lambda = v\otimes \quad \langle \{s\ t\}, g\rangle \xrightarrow{\lambda}_p g'\end{array}}$$

(SEQ FAILED TERM)

$$\frac{p \notin dom(g) \quad \langle s, g\rangle \xrightarrow{\lambda}_p g'}{\begin{array}{l}\vdash \mathsf{isSync}\ s \quad \langle \{s\ t\}, g\rangle \xrightarrow{\mathsf{DP}\otimes}_p g' \\ \vdash \mathsf{isAsync}\ s\ \langle \{s\ t\}, g\rangle \xrightarrow{\mathsf{DP}\times}_p \langle t, g'\rangle\end{array}}$$

Table 7. Resilient Semantics II

of `finish` is the same as in Section 2, but for the rule (END OF FINISH), which now ensures that when $p \notin dom(g)$ a DP$\otimes$ (rather than E$\otimes$) exception is thrown whenever one of the governing activities (either local or remote) threw an exception.

The rules for sequences are collected in Table 7. Rules (SEQ) and (PAR) are the same as in the basic calculus, allowing remote computation under sequential or parallel composition to evolve irrespective of local place failure. The failure of p plays a role only in rule (SEQ FAILED TERM): in this case the termination of the first component s in the sequence $\{s\ t\}$ always results in a DP exception. Moreover, the continuation t is discarded when s is a synchronous statement. On the other hand, when s is an asynchronous statement, t might be an already active remote statement, hence the rule gives to t the chance to terminate correctly.

Rule (PLACE SHIFT) allows the activation of a place-shift only when both the source and the target of the migration are non-failed places. Rule (AT) behaves like in TX10 except that it masks any remote synchronous exception with a DP exception. As an example consider $\overline{\texttt{at}}\,(p)\,\{\overline{\texttt{at}}\,(q)\,s\ \overline{\texttt{at}}\,(r)\,t\}$; if p fails while s and t are (remotely) executing, it is important not to terminate the program upon completion of just s (or just t). Then with rule (AT) we have that a remote computation silently ends even if the control comes back at a failed home. As another example, consider $\overline{\texttt{at}}\,(r)\,\{\overline{\texttt{at}}\,(p)\,\texttt{skip}\ t\}$ with $p \notin dom(g)$, then the failure of `skip` at p must be reported at r as a synchronous error so that the continuation t is discarded.

Example 4.1. Consider the following program, where the code s_q is expected to be executed at q after the termination of any remote activities recursively spawned at p:

$$\overline{\texttt{at}}\,(q)\,\{\texttt{finish}\ \overline{\texttt{async}}\ \overline{\texttt{at}}\,(p)\,\{\texttt{finish}\ s\ \ s_p\}\quad s_q\}$$

Let us also assume that s spawns new remote activities running in a third place r. Now, assume that both p and r fail while s is (remotely) executing. We have that s throws an exception that should be detected by the inner `finish`, however since p is a failed place, termination and error detection in s must be delegated to the outer `finish` waiting at non failed place q: that is indeed performed by rule (END OF FINISH). Hence

(PLACE SHIFT)

$$\frac{(v', g') = \mathsf{copy}(v, q, g)}{\begin{array}{l} p, q \in dom(g)\ \langle \mathtt{at}(q)(\mathtt{val}\, x{=}v)\, s, g\rangle \longrightarrow_p \langle \overline{\mathtt{at}}\,(q)\,\{s[^{v'}/_x]\ \mathtt{skip}\}, g'\rangle \\ q \notin dom(g) \quad \langle \mathtt{at}(q)(\mathtt{val}\, x = v)\, s, g\rangle \xrightarrow{\mathsf{DP}\otimes}_p g \\ p \notin dom(g) \quad \langle \mathtt{at}(q)(\mathtt{val}\, x = e)\, s, g\rangle \xrightarrow{\mathsf{DP}\otimes}_p g \end{array}}$$

(AT)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_q \langle s', g'\rangle \mid g' \quad \lambda' = \begin{cases} \mathsf{MskAs}(\mathsf{DP}\otimes) & \text{if } \lambda{=}v\otimes, p{\notin} dom(g) \\ \lambda & \text{otherwise} \end{cases}}{\langle \overline{\mathtt{at}}\,(q)\, s, g\rangle \xrightarrow{\lambda'}_p \langle \overline{\mathtt{at}}\,(q)\, s', g'\rangle \mid g'}$$

(TRY)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\begin{array}{l} \lambda = \epsilon, v\times \qquad \langle \mathtt{try}\, s\, \mathtt{catch}\, t, g\rangle \xrightarrow{\lambda}_p \langle \mathtt{try}\, s'\, \mathtt{catch}\, t, g'\rangle \mid g' \\ p \in dom(g), \lambda = v\otimes\ \langle \mathtt{try}\, s\, \mathtt{catch}\, t, g\rangle \longrightarrow_p \langle \{s'\ t\}, g'\rangle \mid \langle t, g'\rangle \\ p \notin dom(g), \lambda = v\otimes\ \langle \mathtt{try}\, s\, \mathtt{catch}\, t, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g' \end{array}}$$

Table 8. Resilient Semantics III

we have that the `finish` at q throws a synchronous error and the continuation s_q is discarded. Notice that enclosing the inner `finish` within a try-catch construct is only useful when p is a non failed place. Indeed, consider the program

$$\overline{\mathtt{at}}\,(q)\,\{\mathtt{finish}\,\overline{\mathtt{async}}\,\overline{\mathtt{at}}\,(p)\,\{\mathtt{try}\,(\mathtt{finish}\,\overline{\mathtt{async}}\,\overline{\mathtt{at}}\,(p)\,\{\mathtt{finish}\, s\ \ s_p\})\,\mathtt{catch}\, t\ \ s_p\}\quad s_q\}$$

then by the rule (TRY) for exception handling we have that when p is a failed place the clause is never executed, hence the two programs above have the same semantics. On the other hand, we can recover from an exception in s by installing a try/catch at the non failed place q: $\overline{\mathtt{at}}\,(q)\,\{\mathtt{try}\,(\mathtt{finish}\,\overline{\mathtt{async}}\,\overline{\mathtt{at}}\,(p)\,\{\mathtt{finish}\, s\ \ s_p\})\,\mathtt{catch}\, t\quad s_q\}$.

Example 4.2. Let review Example 2.7 in the context of Resilient TX10. Let be $s'_1 = \{\overline{\mathtt{at}}\,(p)\,\mathtt{finish}\,\overline{\mathtt{at}}\,(q)\,\overline{\mathtt{async}}\, s\ \ s'\}$ and $s'_2{=}\mathtt{finish}\,\{\overline{\mathtt{at}}\,(p)\,\{\overline{\mathtt{at}}\,(q)\,\overline{\mathtt{async}}\, s\}\ \ s'\}$. Assume that place p fails during the remote execution of s at q. Despite such a failure, the behaviour of the two programs is the same as in Example 2.7, according to the Happens Before Invariant Principle That is s' is executed at place 0 after the completion of s in the case of program s'_1 while in s'_2 we have that s' runs in parallel with s. Moreover, let s throw an exception; since the asynchronous remote exception is caught by the closest `finish` construct, in s'_2 we have that the asynchronous exception flows at place 0 while s' correctly continues its execution. On the other hand, the rule (END OF FINISH) ensures that in s'_1 a $\mathsf{DP}\otimes$ exception is thrown and the continuation s' is discarded, according to the Exception Masking Principle.

4.1 Properties of the transition relation

The main properties of the operational semantics of TX10 scale to Resilient TX10. We have encoded the syntax and semantics of Resilient X10 in Coq, as we did for TX10 (see Section 2.2). Using this encoding, we have mechanized the analogous proofs for Resilient X10. First of all, the definition of place-locality must be generalized to the case of partially defined heaps. More precisely, given a configuration $\langle s, g\rangle$, any local object id o in s must be locally defined, while a global reference $o\$p$ might now be a dangling reference since the global object's home place p might have failed.

Definition 4.3 (Place-local Resilient Configuration). *Given a place-local heap g, we say that a configuration $\langle s, g\rangle$ is place-local if $\forall p \in dom(g)$*

- *for any local object id o occurring in s under $\texttt{at}(p)$ or $\overline{\texttt{at}}\,(p)$, we have that $o \in dom(g(p))$ (hence $o \in \mathsf{ObjId}_p$ by place-locality of g).*

Given the definition above, we can prove that resilient semantics preserves place-locality of resilient configurations and that the semantics has no stuck states.

Proposition 4.4 (Place-locality). *If $\langle s, g\rangle$ is a place-local resilient configuration and $\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'$, then $\langle s', g'\rangle$ is a place-local resilient configuration, resp. g' is a place-local heap.*

Proposition 4.5 (Absence of stuck states). *If a configuration k is terminal then k is of the form g.*

Proposition 2.11 and 2.12 dealing with error propagation hold also in Resilient TX10, with a minor modification: in the second clause of Proposition 2.12 the final error thrown by a `finish` construct might be either $\mathsf{E}\otimes$ or $\mathsf{DP}\otimes$.

The main results of this section are the three principles stated above. We start with the Exception Masking Principle, formalized by Theorem 4.6, showing that no synchronous exception other than DP can arise form a failed place.

Theorem 4.6 (Exception Masking Principle). *Let be $p \notin dom(g)$ and $\langle s, g\rangle \xrightarrow{\lambda}_p k$. If $\lambda = v\otimes$, then $v = \mathsf{DP}$.*

The following proposition states that remote computation at a non-failed place proceeds irrespective of local place failure, but for the exception masking effect. Then Theorem 4.8 formalizes the Failed Place Principle, showing that if s performs a correct step at a failed place p, then either (i) s contains a substatement that remotely computed a correct step at a non failed place, or (ii) a local activity ended at p with a DP that has been absorbed by a governing finish. We introduce the following notation: we write $\vdash \mathsf{isLocal}\ s$ whenever s does not contain active remote computation, that is s has no substatements of the form $\overline{\texttt{at}}\,(q)\,s'$. We write $\vdash \mathsf{isRemote}_p\ s$ when any basic statement in s occurs under a $\overline{\texttt{at}}\,(q)$ construct for some place q with $q \neq p$.

Proposition 4.7 (Remote computation). *Let be $\vdash \mathsf{isRemote}_p\ s$. If $\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'$ with $p \in dom(g)$, then $\langle s, g\rangle \longrightarrow_p \langle s, g \setminus \{(p, g(p))\}\rangle \xrightarrow{\lambda'}_p \langle s', g'_*\rangle \mid g'_*$ where $g'_* = g' \setminus \{(p, g'(p))\}$ and $\lambda' = \lambda$ if $\lambda = \epsilon, v\times$ while $\lambda' = \mathsf{DP}\otimes$, if $\lambda = v\otimes$. Moreover $\vdash \mathsf{isRemote}_p\ s'$.*

Theorem 4.8 (Failed Place Principle). *If* $\langle s, g\rangle \longrightarrow_p \langle s', g'\rangle \mid g'$ *with* $p \notin dom(g)$, *then either*

- $s = E[s_1]$ *with* $\vdash \mathsf{isRemote}_p\ s_1$, $\langle s_1, g\rangle \longrightarrow_p \langle s_1', g'\rangle \mid g'$ *and* $s' = E[s_1']$, *or*
- $s = E[\mathtt{finish}_\mu\ t]$, $s' = E[\mathtt{finish}_{\mathsf{DP}}\ t']$ *and* $\langle t, g\rangle \xrightarrow{\mathsf{DP}\otimes \text{ or } \mathsf{DP}\times}_p \langle t', g'\rangle$.

We refer to [24] for a precise definition of the happens before relation in terms of paths that identify occurrences of static statements. We rely here on a much simpler definition in terms of the operational semantics. Intuitively, given a program s with two substatements s_1, s_2, we say that s_1 happens before s_2 whenever in any program execution s_1 is activated, i.e. it appears under an evaluation context, before s_2. This definition is weaker than that in [24] since it captures the idea of "is enabled before" rather than "happens before". However, we think that the core of the Happens Before Invariance is already carried over by Theorem 4.10, and we think that its proof scales to a standard "happens-before" relation at the price of labelling substatements and transitions along the lines of [24].

We denote by $\vec{k}$ a trace $\langle s_0, g_0\rangle \xrightarrow{\lambda_1}_0 \langle s_1, g_1\rangle \xrightarrow{\lambda_2}_0 \ldots \xrightarrow{\lambda_n}_0 \langle s_n, g_n\rangle$. Moreover we write $|\vec{k}|$ for the length n of such a trace, and k_i to indicate the i-th configuration $\langle s_i, g_i\rangle$, $i = 0, ..., n$.

Definition 4.9 (Happens Before). *Let* s_0 *be a program and let* s_1, s_2 *be two substatements of* s_0, *i.e.* $s_0 = E_1[s_1]$ *and* $s_0 = E_2[s_2]$ *for some evaluation contexts* E_1, E_2. *Then we say that* s_1 happens before s_2, *written* $s_1 < s_2$, *whenever for any trace* $\vec{k}$ *such that* $k_0 = \langle s_0, g_0\rangle$ *and* $k_{|\vec{k}|} = \langle E[s_2\rho], g\rangle$ *for some* g, *some evaluation context* E *and some variable substitution* ρ, *there exists* $i \in 0, ..., |\vec{k}|$ *such that* $k_i = \langle E'[s_1\rho'], g'\rangle$ *for some* g', E', ρ'.

Notice that the definition of the Happens Before relation is parametric on a transition relation. Let write $s_1 < s_2$ when we restrict to (traces in) TX10 semantics, and $s_1 <_R s_2$ when considering (traces in) the resilient semantics.

Theorem 4.10 (Happens Before Invariance). *Let* s_0 *be a program and let* s_1, s_2 *be two substatements of* s_0. *Then* $s_1 < s_2$ *if and only if* $s_1 <_R s_2$.

4.2 Equational laws

The equational theory of TX10 can be smoothly generalized to the resilient calculus. In order to scale the notion of weak bisimilarity to Resilient TX10 we have to consider generalized environment moves that take into account the failure of a number of places.

Definition 4.11 (Resilient Environment move). *An environment move* Φ *is a map on global heaps satisfying:*

1. *if* g *is place-local, then* $\Phi(g)$ *is place-local,*
2. $dom(\Phi(g)) \subseteq dom(g)$, *and* $\forall p \in dom(\Phi(g))\ dom(g(p)) \subseteq dom(\Phi(g)(p))$.

The weak bisimilarity for Resilient TX10 is then defined as in Definition 3.2, where we rely on resilient environment moves and the operational steps used in the bisimulation game are those defined in this section. In particular, this means that also place failures occurring at any time must be simulated by equivalent configurations. We discuss in the following which of the equational laws of Section 3 are still valid in the resilient calculus.

$$\vdash \mathsf{noAsync}\ s \quad \vdash \mathsf{isLocal}\ s \quad \{\mathtt{skip};\ s\} \equiv s \qquad \text{(1a R)}$$

$$\vdash \mathsf{isSync}\ s \quad \{s\ \mathtt{skip};\} \not\equiv s \qquad \text{(1b R)}$$

$$\vdash \mathsf{noAsync}\ s \ \vdash \mathsf{isLocal}\ s \quad \mathtt{try}\,\mathtt{throw}\,v\,\mathtt{catch}\,s \equiv s \qquad \text{(9 R)}$$

The law (1) of Section 2 is not valid anymore, as illustrated by (1a R) and (1b R) above. The problem is that now the place where the commands are executed may fail at any time. Hence, in order for the law to be valid also at a failed place, law (1a R) requires a stronger constraint for s so to ensure that also the r.h.s throws a $\mathsf{DP}\otimes$. On the other hand law (1b R) never holds since the failure of the local place can happen after the completion of s but before the execution of `skip`, thus only the l.h.s. would throw a $\mathsf{DP}\otimes$. Similarly, law (9) of Section 2 is replace here by the stricter law (9 R) to ensure that s throws a synchronous $\mathsf{DP}\otimes$ error whenever the local place is failed. All the other laws of TX10 are still valid in Resilient TX10, but for those involving place shifting, summarized below:

$$\mathtt{at}\,(p)\,\mathtt{skip} \not\equiv \mathtt{skip} \qquad \text{(14 R)}$$

$$\mathtt{at}\,(p)\,\mathtt{throw}\,v \not\equiv \mathtt{throw}\,v \qquad \text{(15 R)}$$

$$\mathtt{at}\,(p)\,\{s\,t\} \not\equiv \{\mathtt{at}\,(p)\,s\ \ \mathtt{at}\,(p)\,t\} \qquad \text{(16 R)}$$

$$\overline{\mathtt{at}}\,(p)\,\{s\,t\} \equiv \{\overline{\mathtt{at}}\,(p)\,s\ \ \overline{\mathtt{at}}\,(p)\,t\} \qquad \text{(16 dyn R)}$$

$$\mathtt{at}\,(p)\,\mathtt{at}\,(q)\,s \not\equiv \mathtt{at}\,(q)\,s \qquad \text{(17 R)}$$

$$\mathtt{async}\,\mathtt{at}\,(p)\,s \not\equiv \mathtt{at}\,(p)\,\mathtt{async}\,s \qquad \text{(18 R)}$$

$$\mathtt{at}\,(p)\,(\mathtt{try}\,s\,\mathtt{catch}\,t) \not\equiv \mathtt{try}\,(\mathtt{at}\,(p)\,s)\,\mathtt{catch}\,(\mathtt{at}\,(p)\,t) \qquad \text{(19 R)}$$

$$\mathtt{finish}\,\mathtt{at}\,(p)\,s \not\equiv \mathtt{at}\,(p)\,\mathtt{finish}\,s \qquad \text{(24 R)}$$

The laws (14) and (15) for place shift does not hold in the resilient calculus since they involve two terms that run in different places that might fail at different moments. Rule (16) does not hold anymore since the local place can fail after the completion of s but before the place shift of t. On the other hand its dynamic version, i.e., law (16 dyn R) is still valid since both terms already run at the same place p and the failure of local place does not affect remote computation. The law (17) does not hold since p may fail before the place-shift at q. Note that also the dynamic version of rule (17) does not hold, i.e. $\overline{\mathtt{at}}\,(p)\,\overline{\mathtt{at}}\,(q)\,s \not\equiv \overline{\mathtt{at}}\,(q)\,s$ since the failure of p would mask any exception thrown at q. Law (18 R) (as well as its dynamic version) does not hold anymore because of the exception masking effect. Indeed, if s remotely throws a synchronous exception $v\otimes$ and the home place is failed, we have that the r.h.s. throws a $v\times$ exception while the l.h.s. throws $\mathsf{DP}\times$ by means of masking. Law (19) does not hold anymore since p may fail after s has thrown an exception but before the activation of the handling t. Finally, in law (24 R) a difference appears between the two terms when the remote place p fails after the remote code has been activated. In this case s throws a DP exception at the

failed place, but in the l.h.s. the local (non failed) `finish` masks this exception as a generic E, while in the r.h.s. the exception reported locally is still DP.

5 Conclusions and Future work

We have studied a formal small-step structural operational semantics for TX10, that is a large fragment of the X10 language covering multiple places, shared mutable objects, sequences, `async`, `finish`, `at` and `try/catch` constructs. We have then shown that this framework smoothly extends to the case where places dynamically fail. Failure is exposed through exceptions thrown by any attempt to execute a statement at the failed place. The error propagation mechanism in Resilient TX10 extends that of TX10 (i) by discarding exception handling at failed places, i.e. no `catch` clause is ever executed at failed places, and (ii) by masking with a *DeadPlaceException* any remote exception flowing back at the failed place. Moreover, we established a Happens Before Invariance Principle showing that the failure of a place p does not alter the happens before relationship between statements at places other than p.

As an example of formal methods that can be developed on top of the given operational semantics, we studied a bisimulation based observation equivalence. We showed that it correctly encodes the observation power of the concurrent context by proving that it is a congruence. We illustrated this equivalence by means of a number of laws dealing with the main constructs of the language, discussing which of these equivalences are invariant under place failures. The axiomatization of the given equivalence is left for future work. We think that the resilient equational theory opens the way to the development of laws that can be used in the X10 compiler to optimize programs, e.g. using polyhedral analysis [24]. We also plan for future work the extension of the framework we presented to cover the `atomic` and `when` constructs from X10. We also plan to develop a denotational semantics for TX10 based on a pomset model that naturally allows the definition of the happens before relation. Another promising approach seems to be the study of full abstraction by extending to this setting the trace set model of S. Brookes [7].

References

1. Alexander Ahern and Nobuko Yoshida. Formalising java rmi with explicit code mobility. In *OOPSLA '05*, pages 403–422, New York, NY, USA, 2005. ACM.
2. Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *Very Large Data Bases*, pages 734–746, 2013.
3. Roberto M. Amadio. An asynchronous model of locality, failure, and process mobility. In *Coordination Languages and Models*, LNCS, pages 374–391. Springer, 1997.
4. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Comp. Sci. Springer, 2004.
5. G.M. Bierman, M.J. Parkinson, and A. M. Pitts. Mj: An imperative core calculus for java and java with effects. Technical report, University of Cambridge Computer Laboratory, 2003.
6. Frank S. de Boer, Joost N. Kok, Catuscia Palamidessi, and Jan J. M. M. Rutten. The failure of failures in a paradigm for asynchronous communication. In *CONCUR '91*, pages 111–126, London, UK, UK, 1991. Springer-Verlag.

7. Stephen Brookes. Full abstraction for a shared variable parallel language. In *In Proceedings, 8th Annual IEEE Symposium on Logic in Computer Science*, pages 98–109. IEEE Computer Society Press, 1993.
8. Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, April 2007.
9. 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 *OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM.
10. UPC Consortium et al. UPC language specifications. *Lawrence Berkeley National Lab Tech Report LBNL–59208*, 2005.
11. David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Vijay Saraswat, Olivier Tardieu, Kiyokuni Kawachiya, Hiroki Murata, and Mikio Takeuchi. Resilien X10: Efficient failure-aware programming. In *PPoPP '14*, pages 67–80, New York, NY, USA, 2014. ACM.
12. Miyuru Dayarathna, Charuwat Houngkaew, and Toyotaro Suzumura. Introducing Scalegraph: an X10 library for billion scale graph analytics. In *X10 '12*, pages 6:1–6:9, New York, NY, USA, 2012. ACM.
13. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
14. Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *CONCUR '96*, pages 406–421, London, UK, UK, 1996. Springer-Verlag.
15. Adrian Francalanza and Matthew Hennessy. A theory of system behaviour in the presence of node and link failure. *Inf. Comput.*, 206(6):711–759, 2008.
16. Matthew Hennessy. *A Distributed Pi-Calculus*. Cambridge University Press, New York, NY, USA, 2007.
17. Jonathan K. Lee and Jens Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *PPoPP '10*, pages 25–36, New York, NY, USA, 2010. ACM.
18. Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.
19. James Riely and Matthew Hennessy. Distributed processes and location failures. *Theor. Comput. Sci.*, 266(1-2):693–735, September 2001.
20. Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification version 2.2, March 2012. `x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`.
21. Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *CONCUR 2005 - Concurrency Theory*, pages 353–367, London, UK, 2005. Springer-Verlag.
22. Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3R: increased performance for in-memory Hadoop jobs. *Proc. VLDB Endow.*, 5(12):1736–1747, August 2012.
23. X10 Global Matrix Library. `https://x10.svn.sourceforge.net/svnroot/x10/trunk/x10.gml`, October 2011.
24. Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadhye, and Vijay Saraswat. Array dataflow analysis for polyhedral x10 programs. In *PPoPP '13*, pages 23–34, New York, NY, USA, 2013. ACM.
25. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *HotCloud'10*, pages 10–10, 2010.

A Artifact Description

Authors of the artifact. Avraham Shinnar.

Summary. The artifact is a mechanization of the semantics for TX10 and Resilient X10 in Coq. The mechanization verifies key properties of both language(s), bringing an additional level of assurance to the paper versions. These properties include the totality and computability of both languages. The latter proofs additionally serve as interpreters for the languages. An important part of the mechanization effort is the implementation of a total heap copy algorithm. This algorithm is shown to have the properties states in the accompanying paper. In particular, the result is a heap isomorphism of the original.

Content. The artifact package includes:

- An html page (index.html) describing the structure of the development, and an overview of the content of each file.
- The actual mechanization, presented as a series of Coq source (*.v) files.
- A Makefile that can be used to automate building (verifying) the development.

Getting the artifact. The artifact endorsed by the Artifact Evaluation Committee is available free of charge as supplementary material of this paper on SpringerLink.

Tested platforms. This artifact should compile on any platform that supports Coq 8.4pl3 (`http://coq.inria.fr/download`). A few Gigabytes of RAM are required for the compilation process. Compiling the artifact (in particular CopyObj) takes multiple hours.

License. EPL-1.0 (http://www.eclipse.org/legal/epl-v10.html)

MD5 sum of the artifact. 52acdbfd95ad5a7f48b959a253a286a9

Size of the artifact. 90K