ActorX10: An Actor Library for X10

Sascha Roloff[‡], Alexander Pöppl[†], Tobias Schwarzer[‡], Stefan Wildermann[‡],
Michael Bader[†], Michael Glaß[‡], Frank Hannig[‡], and Jürgen Teich[‡]

[‡]Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany
{sascha.roloff, tobias.schwarzer, stefan.wildermann, michael.glass, frank.hannig, juergen.teich}@fau.de
[†]Technical University of Munich (TUM), Germany
{poeppl, bader}@in.tum.de

Abstract

The APGAS programming model is a powerful computing paradigm for multi-core and massively parallel computer architectures. It allows for the dynamic creation and distribution of thousands of threads amongst hundreds of nodes in a cluster computer within a single application. For programs of such a complexity, appropriate higher level abstractions on computation and communication are necessary for performance analysis and optimization. In this work, we present actorX10, an X10 library of a formally specified actor model based on the APGAS principles. The realized actor model explicitly exposes communication paths and decouples these from the control flow of the concurrently executed application components. Our approach provides the right abstraction for a wide range of applications. Its capabilities and advantages are introduced and demonstrated for two applications from the embedded system and HPC domain, i. e., an object detection chain and a proxy application for the simulation of tsunami events.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages, Data-flow languages; F.1.1 [*Computation by Abstract Devices*]: Models of Computation

General Terms Design, Theory, Languages

Keywords Parallel programming, actor-based programming, high performance computing, stream processing

1. Introduction

The APGAS (asynchronously partitioned global address space) model, as implemented in the X10 programming language, is a powerful programming model for massively parallel distributed memory systems. However, that complexity poses a significant challenge for the application developer, as the communication pattern between different threads running concurrently and on different places is not always obvious. This becomes especially apparent when considering the implicit capture of object graphs when using lambda functions and the `at` construct. However, a significant number of applications have a relatively stable communication structure—e. g., pipelining or communication with a fixed number of other components—once initialization has concluded.

The actor model, as first described by Hewitt et al. (1973) and later formalized by Agha (1985), can help to lessen the burden on the developer by making the communication between different parts of the program explicit and by separating the control flow aspects from the computational aspects of a program. *Actors* naturally describe concurrently executable parts of computations and communicate by sending and receiving data using *ports* and *channels*. Furthermore, their activation is data-driven which enables to execute data-dependent code as soon as data is available in a self-scheduled manner. Moreover, in contrast to the closure-based place shift which implicitly captures, serializes and deserializes data, in the actor model it is always directly recognizable what will be sent. Finally, if developers formalize their computation in this way, they also gain the possibility to perform further analyses upon the program. One technique popular in the embedded world is the so-called *design space exploration*, a methodology where different run configurations are evaluated in order to find a set of Pareto-efficient mappings, e. g., of X10 places to physical resources, that produce a Pareto-efficient set of quality numbers (Weichslgartner et al. 2014).

In this paper, we present an X10 library that maps the properties of the actor model onto the APGAS paradigm. The library uses the features provided by the APGAS environment to implement its functionality as well as to enable features such as migrating actors between different places. Our contributions may be summarized as follows:

- Definition of a formalism that incorporates the PGAS paradigm into an actor model
- Implementation of the formalism by an actor library in X10
- Demonstration of the viability of combining the actor model with X10 using examples from two different application areas, namely a simulation of the shallow water equations from the domain of HPC and a computer vision application from the area of embedded systems.

2. Actor Model

In this section, we formally define the notions of actors and actor graphs. This general model is the basic formal underpinning of the proposed X10 actor library and—as we will discuss later—can be specialized to adhere to different *Models-of-Computation* (MoCs).

DEFINITION 1 (Actor graph). *An actor graph is a directed graph $G_a = (A, C)$ containing a set of actors A and a set of channels $C \subseteq A.O \times A.I$ connecting actor output ports $A.O$ with actor input ports $A.I$. Each channel has a buffer size determined by*

X10'16, June 14, 2016, Santa Barbara, CA, USA

http://dx.doi.org/10.1145/2931028.2931033

$n : C \to \mathbb{N}^+$, and a possibly empty sequence of initial tokens denoted by $d : C \to D^*$ where D^* denotes the set of all possible finite sequences of tokens.

The communication between actors through channels connected by ports occurs in FIFO order. This is natural for many applications in, e. g., stream processing where all data produced must be consumed in the same order. As indicated by the formal definition, actors are only permitted to communicate with each other via channels and there is no *implicit* communication, e. g., via shared data structures, allowed. An actor is thereby constrained to only consume data items also called *tokens* from channels connected to its input ports and to produce tokens on channels connected to its output ports. It is also possible to place initial tokens in FIFO channels, e.g., $d(c) = \langle 1, 2 \rangle$ to denote that two initial tokens with the values 1 and 2 are on the channel c.

In actor-based modeling, rules must be specified how and when actors may be activated to perform a computation. Thereby, it must also be specified, how many tokens are read and consumed from input ports, and how many tokens will be produced on which output ports during such a transition also called *firing*. In our model of an actor, the communication and firing behavior is strictly separated from the functions that are performed on the data at input ports. Moreover, this firing rules are formally described by a finite state machine as follows.

DEFINITION 2 (Actor). *An* actor *is a tuple $a = (I, O, F, R)$ containing actor ports partitioned into a set of actor input ports I and a set of actor output ports O, a set of functions F, and a finite state machine R called* firing FSM. *The functions encapsulated in an actor are partitioned into so-called* actions *$F_{actions} \subseteq F$ and* guards *$F_{guards} \subseteq F$. Functions are activated during a so-called* firing transition *of the FSM R, which unambiguously formalizes the communication behavior of the actor (i. e., the number of tokens consumed and produced in each actor firing). Actions may produce results in the form of output tokens residing in the FIFO channels connected to the actor output ports. Using guards, more complex models of computation may be modeled. In particular, the activation of actors is based not only on the availability of a minimal number of tokens on the input ports, but also on their values. Guards return a Boolean value and may be assigned to each transition of the FSM of an actor.*

DEFINITION 3 (Actor (Firing) FSM). *The* firing FSM *of an actor $a \in A$ is a tuple $R = (Q, q_0, T)$ containing a finite set of states Q, an initial state $q_0 \in Q$, and a finite set of transitions T. Moreover, a* transition *of an FSM is a tuple $t = (q, k, f, q') \in T$ contains the current state $q \in Q$, an activation pattern k, the respective action $f \in a.F_{actions}$, and the next firing state $q' \in Q$. The activation pattern k is a Boolean function which decides if transition t can be taken (*true*) or not (*false*) based on: (1) a minimum number of available tokens on the input ports $a.I$, (2) a set of guard functions $F' \subset F_{guards}$, and (3) a minimum number of free places in the channels connected to respective output ports.*

An example of a simple actor for computing the square root of a number and its firing FSM is shown in Figure 1. In the initial state S_1 of the firing FSM, the actor waits for at least one token on its input port in_1 ($\#in_1 \geq 1$) as well as for free space of at least one token on its output port out_1 ($\#out_1 \geq 1$). If the value of the first arriving token is greater than zero (guard: $in_1 \geq 0$), then the square root result is computed (action: $\sqrt{in_1}$) and the result token produced directly on output port out_1 ($out_1 := \sqrt{in_1}$). However, if the number is negative, its absolute value is calculated first and produced as a token on output port out_2 which is fed back over a channel to input port in_2. In state S_2, the actor waits for its activation by at least one token on this input port in_2 as well as for free space of one token on the output port out_1 and then advances back to the initial state S_1 once having computed the square root of this token on input in_2.

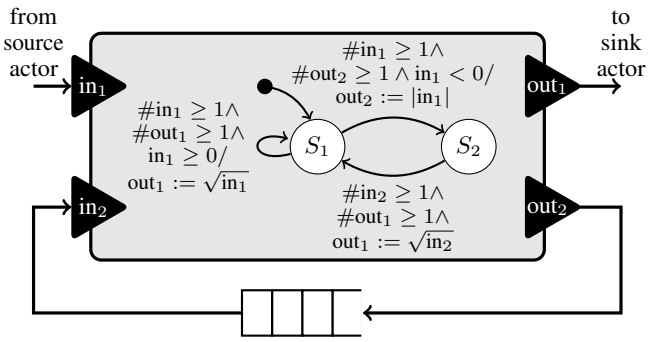

Figure 1. Example of an actor with two input and two output ports calculating the square root of a number. In the initial state S_1 of the firing FSM, the actor waits for one token on its input port in_1 and for free space of one token on its output port out_1. If the number is positive, the square root is computed and the result token is produced on output port out_1. However, if the number is negative, its absolute value is calculated and produced on output port out_2 which is fed back over a channel to input port in_2. In state S_2, the actor waits and consumes a token on input port in_2 and then advances back to the initial state S_1.

The concept of actor FSM is an extension of the state machines of the FunState model of computation (see Strehl et al. (2001)) standing for functions driven by state machines, by additionally providing mechanisms to check the availability of sufficient space on output channels before a transition can be taken. The state transitions of an actor FSM are annotated each with a so-called *activation pattern* which is a Boolean expression.

2.1 Mapping Actor Model onto PGAS Model

The aforementioned formal model clearly separates control flow and data flow. It encapsulates data processing in side-effect functions called *actions*. Similar to the simple task model proposed by Roloff et al. (2014), our formal actor model may be seamlessly mapped now to the PGAS programming model as data processing is performed solely on locally available data, whereas all data transmissions between actors are explicitly modeled by channels.

The X10 PGAS model introduces the notion of places and giving us now the opportunity to distribute an actor graph among the available places in order to exploit processing power of the underlying cores and for optimal workload distribution. Informally, each actor and each channel must be assigned a certain place on which they reside during the execution of the actor graph. For static actor graphs, an optimal actor and channel distribution with respect to different objectives, such as performance, power efficiency, or temperature distribution, can be found or at least approximated by applying techniques of design space exploration. For this paper, we consider a manual distribution of actors to places. In case of more adaptive scenarios, the dynamic migration of actors and channels to other places at runtime has to be considered. Techniques for static design space exploration and dynamic place assignment are ongoing work.

In this paper, we present a first step into that direction by introducing an X10 actor library. In the following, we will show how the formal actor model is implemented by means of available X10 language constructs. Moreover, different MoCs that can be realized by means of specialization of the actor library are presented using two concrete examples from the image-processing and the high-performance computing domains.

```
1 public class SquareRootActor extends Actor {
2   val in1 = newInPort[double]("in1");
3   val in2 = newInPort[double]("in2");
4   val out1 = newOutPort[double]("out1");
5   val out2 = newOutPort[double]("out2");
6   var state:int = 1;
7   public def this() { super("SquareRoot"); }
8   public def act() {
9     switch (state) {
10      case 1: // S1
11        if (in1(1) && out1(1) && in1.peek() >= 0) {
12          val sqrtNumber = Math.sqrt(in1.read());
13          out1.write(sqrtNumber);
14          state = 1;
15        } else if (in1(1) && out2(1) &&
16            in1.peek() < 0) {
17          val absNumber = Math.abs(in1.read());
18          out2.write(absNumber);
19          state = 2;
20        }
21        break;
22      case 2: // S2
23        if (in2(1) && out1(1)) {
24          val sqrtNumber = Math.sqrt(in2.read());
25          out1.write(sqrtNumber);
26          state = 1;
27        }
28        break;
29    }
30  }
31 }
```

Figure 2. Square root actor from Figure 1 using actorX10.

3. X10 Implementation

In this section, we present the basic data structures and primitives of our library-based implementation of the previously introduced actor model and semantics in X10.

3.1 Actor

According to the definition of an actor (see Section 2), we introduce the fundamental abstract class `Actor`. Each `Actor` consists of two lists, one for its input (*I*) and output (*O*) ports, and provides factory methods for creating ports. `Actor` defines an abstract method `act`, which has to be implemented by its subclasses and contains the (firing) state machine (*R*) as well as all functions (*F*) of the actor. Each actor and port possesses a unique name.

An actor is defined by creating a subclass of the `Actor` class and implementing the method `act`. The new subclass may define its own constructors. Ports are created using the factory methods of the class `Actor` and are used within the `act` method to send tokens to and receive tokens from other actors.

3.2 Port

The input and output ports of an actor define its logical interface to the outside world. From these, the application developer may deduce the type of data that is produced on, respectively consumed on a port of an actor. Accordingly, we introduce the generic classes `InPort[T]` and `OutPort[T]`. The type parameter `T` allows for a type-safe communication between actors due to the fact that only ports with the same type can be connected by a channel. An actor implementation may `read` data from an `InPort` and `write` data to an `OutPort`[1]. These communication primitives fail if no data is available or no space is left at the corresponding channel. Methods are provided that allow to check these activation patterns non-destructively (e. g., `in1(1)` checks for one input token on the input port `in1`). Internally,

[1] In a certain state of a firing FSM, the activation patterns are evaluated. Yet, tokens on an input port are only destructively consumed by `read` once an activated transition fires that required a certain number of tokens on that port for activation. Once a transition is selected to fire, the respective number of tokens are consumed. Whereas, the transition completes by producing output tokens only once the actions as evoked during the chosen transition have finished.

```
1 public static def main(args:Rail[String]) {
2   val source = new SourceActor(100);
3   val sink = new SinkActor();
4   val squareRoot = new SquareRootActor();
5   val ag = new ActorGraph();
6   ag.addActor(source);
7   ag.addActor(sink);
8   ag.addActor(squareRoot);
9   ag.connectPorts(source.outPort, squareRoot.in1);
10  ag.connectPorts(squareRoot.out1, sink.inPort);
11  ag.connectPorts(squareRoot.out2, squareRoot.in2);
12  ag.moveActor(squareRoot, here.next());
13  ag.start();
14 }
```

Figure 3. Actor graph generation and execution of the square root example. The three actors `SourceActor`, `SinkActor`, and `SquareRootActor` are connected with each other properly. All actors are mapped to the place `here` except of the actor `SquareRootActor` itself which is moved to the place `here.next()` according to the above code. The method `start` then initializes all actors to their respective initial state.

a port has a global reference to a `Channel` and forwards all read and write requests to the corresponding X10 object on its respective home place, which may include a place shift. These channel references are set in the port mapping phase during the creation of an actor graph.

Example of an X10 Actor

An implementation of the square root example actor from Figure 1 is depicted in Figure 2. The class `SquareRootActor` is derived from the `Actor` class and two input and two output ports of type `double` are created. A state variable—initialized with state S_1—indicates the current state of the actor. The implementation of the state machine is described by the body of the `act` method. In state S_1, the two activation conditions emanating state S_1 are checked. According to line 11, at least one token must be present on input port in_1. The guard method `peek` in line 12 tests the value of the first token in the FIFO channel connected to port in_1 non-destructively[2].

3.3 Channel

A channel is implemented by the generic class `Channel[T]`. It provides a FIFO implementation of limited capacity. Similar to the port classes, it has a type parameter `T` that matches the type of its writing and reading port. Data of type `T` is stored in a thread-safe queue which allows concurrent read and write accesses. Furthermore, `peek` and `poke` methods allow the caller to check the presence or absence of tokens in the queue. As this communication pattern may involve several places, each token must be *deeply copied* from the output port place via the channel place to the input port place, if these places are not the same.

3.4 Actor Graph

The class `ActorGraph` holds lists of global references to `Actors` and `Channels`, according to the definition of actor graphs. Constructing an actor graph involves adding actors (`addActor`) and connecting input and output ports (`connectPorts`) of these actors via channels. A connection between two ports is established by creating a `Channel` objects and exchanging references between port and channel on both sides. Actors can be freely migrated to other places by calling the `moveActor` method prior to the start of the actor graph's execution. During migration, the state of the actor is copied to the new place and all affected channel and port references are updated. The actor graph may be started by calling the `start` method, which internally

[2] Note that whereas our formal actor model introduced in Section 2 assumes a non-deterministic choice of transition in case multiple transitions should become simultaneously activated, our X10 implementation implements a priority given by the order in which the code checks the activation conditions.

calls the `start` method of all its actors on their respective place. This method only returns once all actors are finished.

Example of an Actor Graph Creation

An example of an actor graph creation using our library is depicted in Figure 3. First, all relevant actors are instantiated using their individual constructors (line 2-4). In this case, the source actor generates a list of 100 random numbers. After instantiating the actor graph in line 5, all actors are added to it and their ports are properly connected. All actors are mapped to the place `here` except of the actor `SquareRootActor` itself which is moved to the place `here.next()`. Finally, the whole actor graph is started in line 13.

3.5 Actor Execution

In order to best exploit the natural concurrency of actors, we heavily made use of the concurrent constructs X10 provides. In particular, for each port of an actor, a separate activity is spawned that waits at the channel either for an element production in case of an `InPort` or an element consumption in case of an `OutPort`. Once an event occurs, the respective port is marked either readable or writable and the actor activity is notified by calling its `act` method.

4. Case Studies

In this section, we showcase two complex real-world X10 applications that were adapted to use our actor library.

4.1 Object Detection

Object detection is a typical computer vision task, for example as used in the robotics domain. Here, a previously trained object is detected in a stream of images. There are several approaches to solving this problem, most of which involve chaining together multiple algorithms. Our implementation uses a scale-invariant feature transform (SIFT) based approach and comprises of the following sub-algorithms:

1. Harris operator to detect corners in an image. These act as a basis for the features of the image, which are later used to compare against the object features.
2. SIFT description to transform the corners of the image into 128-dimensional feature descriptors. The descriptors are scale and rotation invariant, which allow detection of an object, even if it is rotated and/or scaled.
3. SIFT matching to compare the current image features with the object features. Here, the goal is to find for each image feature the nearest neighbor out of the object features in the 128-dimensional space. Specialized data structures such as k-d trees are used to handle this highly-dimensional search space.
4. Random sample consensus (RANSAC) algorithm to register the matched features to the shape of the actual object. This algorithm eliminates outliers and detects the objects.

All actors are executed periodically and data is exchanged from one stage to the next. This algorithm chain intuitively maps to the actor model where each algorithm is encapsulated into an actor. The actorized view of the full chain is depicted in Figure 4 and an implementation of the SIFT matching actor using our actor library is shown in Figure 5.

The actors only exchange one token with their neighbors. This triggers the firing of the next actor. In this context, tokens can be: whole images, derivatives from images, lists of corners and descriptors as well as coordinates.

Due to data dependencies, all steps in the computation of one frame have to be executed in sequence. However, actorized modeling allows for the exploitation of pipeline parallelism and hence for the concurrent processing of several stages of the algorithm chain. Using the actor library, the throughput of this application for a

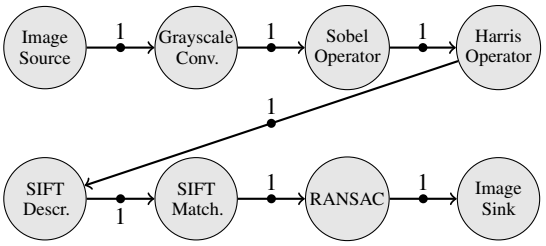

Figure 4. Actor graph of an object detection algorithm chain.

```
1 public class SIFTMatchingActor extends Actor {
2   val inPort = newInPort[SIFTMatchingToken]("in");
3   val outPort = newOutPort[RANSACToken]("out");
4   val kdtree = new KdTree();
5   var state:int = 1;
6
7   public def act() {
8     // check end condition
9     if (inPort(1) && outPort(1)) {
10      if (inPort.peek().isFinished()) {
11        inPort.read();
12        outPort.write(new RANSACToken());
13        stop();
14      }
15    }
16    // execute state machine
17    switch (state) {
18      case 1:
19        if (inPort(1)) {
20          val token = inPort.read();
21          kdtree.initialize(token.getFeatures());
22          state = 2;
23        }
24        break;
25      case 2:
26        if (inPort(1) && outPort(1)) {
27          val inToken = inPort.read();
28          val features = inToken.getFeatures();
29          val matchedFeatures = SIFTMatching.apply(
                kdtree, features);
30          val outToken = new RANSACToken(
                matchedFeatures);
31          outPort.write(outToken);
32        }
33        break;
34    }
35  }
36 }
```

Figure 5. Implementation of the SIFT matching actor.

stream of input images could be substantially improved. The latency for one image, of cause, remains the same. However, the slowest member of the chain limits the maximum throughput. Parallel X10 constructs can aid in the in the expedition of slow actors as well as appropriate mappings of algorithms to places containing an accelerator. Evaluating different mapping decisions can be done in the context of a design space exploration.

4.2 Shallow Water Equations

The shallow water equations are a set of hyperbolic partial differential equations used, for example, to simulate tsunami events. A property of hyperbolic PDEs is that changes in the simulation domain propagate locally instead of instantaneously over the entire domain. We implemented a solver for the shallow water equations, SWE-X10 (Pöppl and Bader 2016), that exploits the benefits of the local wave propagation, e. g., by only performing computations in actors that actually exhibit non-static behavior. The solver is based on a C++ implementation first described in Breuer and Bader (2012).

In both codes, the simulation domain is discretized into a Cartesian grid of equally sized cells with constant cell data. Moreover, both are based on a finite volume time stepping scheme, where the next time step is computed based on the state of the unknowns in the previous time step. The update is computed by solving a Riemann problem at

```
1 public class SimulationActor extends Actor {
2   private val outPorts:Rail[OutPort[Data]];
3   private val inPorts:Rail[InPort[Data]];
4   private val communicators:Rail[BlockCommunicator];
5   private val patchArea:SimulationArea;
6   private var block:MoveablePatchSWEBlock;
7   // Helper attributes ommitted.
8   // Initialization code omitted.
9
10  public def act() {
11    if (initialSend && mayRead()) {
12      sendData(currentTime, false);
13      initialSend = false;
14    } else if (currentTime < endTime
15        && mayRead() && mayWrite()) {
16      receiveData();
17      block.computeNumericalFluxes();
18      numIterations++;
19      block.updateCells(dt);
20      currentTime += dt;
21      sendData(currentTime,
22          (currentTime >= endTime));
23    } else if (currentTime >= endTime) {
24      stop();
25    }
26  }
27
28  private def mayRead() =
29    inPorts.reduce((i:InPort[Data], b:Boolean) =>
30        ((i == null || i(1)) && b), true);
31
32  private def mayWrite()  =
33    outPorts.reduce((o:OutPort[Data], b:Boolean) =>
34        ((o == null || o(1)) && b), true);
35
36  private def receiveData() {
37    for ([i] in communicators) {
38      if (communicators(i) != null) {
39        val data = inPorts(i).read();
40        communicators(i).setGhostLayer(data);
41      }
42    }
43  }
44
45  private def sendData(curTime:Float, dead:Boolean) {
46    for ([i] in communicators) {
47      if (communicators(i) != null) {
48        val data = communicators(i)
49            .getCopyLayer(curTime, dead);
50        outPorts(i).write(data);
51      }
52    }
53  }
54 }
```

Figure 6. Implementation of the `SimulationActor` class.

each edge between every two neighboring cells in the simulation. This dependency becomes problematic once the number of simulated cells gets large enough to warrant its distribution amongst multiple places. Now the cells at the boundary of each two neighboring distributed regions have to be exchanged between places in every time step.

To solve the problem of data exchange without global coordination, we chose the actor model. We subdivided the simulation domain into $d_y \times d_x$ quadratic patches of equal size. Each patch has a coordinate (i, j) that corresponds to its position in the simulation domain. These are each controlled by a single actor $a_{i,j} = (I_{i,j}, O_{i,j}, F_{\text{SWE}}, R_{\text{SWE}})$. Each of these has one outgoing and one incoming port for every patch boundary that is not an outer boundary of the simulation domain.

$$\begin{aligned} I_{i,j} = \{ip_{k,l} | \, &(k \in \{i-1, i+1\} \wedge 0 \leq k < d_y \wedge l = j) \\ &\vee (l \in \{j-1, j+1\} \wedge 0 \leq l < d_x \wedge k = i)\} \\ O_{i,j} = \{op_{k,l} | \, &(k \in \{i-1, i+1\} \wedge 0 \leq k < d_y \wedge l = j) \\ &\vee (l \in \{j-1, j+1\} \wedge 0 \leq l < d_x \wedge k = i)\} \end{aligned}$$

In our X10 implementation of the `SimulationActor` class, depicted in Figure 6, the `Rail inPorts` corresponds to $I_{i,j}$, while

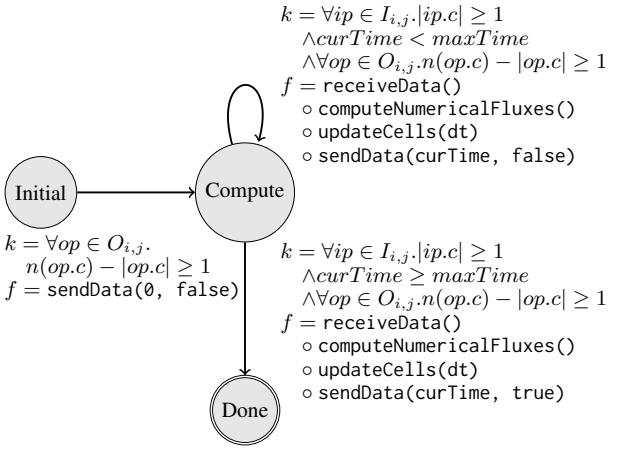

Figure 7. Finite state machine for a simulation actor in SWE-X10. It has three states: In the initial state, the setup takes place, the second state represents the main loop, with each transition being the computation of one timestep. The last state is reached when the simulation is finished.

`outPorts` corresponds to $O_{i,j}$. The called functions F_{SWE} are defined as follows: `receiveData` and `sendData` read data from the incoming ports and write to the outgoing ones, respectively, `computeNumericalFluxes` determines the updates for the next timestep by solving Riemann problems for every edge between two neighboring cells, and `updateCells` updates the variables stored in the cells and progresses the simulation.

The finite state machine R_{SWE}, given in Figure 7, defines the control flow of the actor. Its implementation is the `act` method of the `SimulationActor` class, shown in Figure 6. The method is called whenever there is a change in one of the actor's ports. There are three states the computation may embody: in the *Initial* state, the actor will write the initial cell values from the boundary of its domain to its `OutPorts`, thereafter switching to the *Compute* state. Here, the actor reads the data from the neighboring unknowns, computes a new time step, updates the block's unknowns and sends the new values from its own boundaries to the neighboring actors whenever there is data in all of the `InPorts` and capacity in all of the `OutPorts`. As long as the simulation has not progressed past `maxTime`, the actor will remain in the *Compute* state; afterwards, its state will move to *Done* and the computation concludes.

With the structure of an individual actor defined, we are able to formulate a description of the actor graph. The simulation domain is subdivided into $d_y \times d_x$ patches each controlled by one actor. Hence, the actor graph, G_a^{SWE}, may be described as follows:

$$\begin{aligned} G_a^{\text{SWE}} &= (A_{\text{SWE}}, C_{\text{SWE}}) \qquad (1) \\ A_{\text{SWE}} &= \{a_{i,j} | \, 0 \leq i < d_y \wedge 0 \leq j < d_x\} \\ C_{\text{SWE}} &= \{(op_{k',l'}, ip_{i',j'}) | \\ &\qquad op_{k',l'} \in O_{i,j} \wedge ip_{i',j'} \in I_{k,l} \wedge \\ &\qquad (|i-k| + |j-l| = 1) \wedge \\ &\qquad i' = i \wedge j' = j \wedge k' = k \wedge l' = l\} \\ n_{\text{SWE}}(c) &= 1 \qquad \forall c \in C_{\text{SWE}} \end{aligned}$$

As an example, the actor graph for a simulation run with 9 patches (or 3×3 actors) may be seen in Figure 8. Further descriptions of the software and a performance evaluation are shown in Pöppl and Bader (2016).

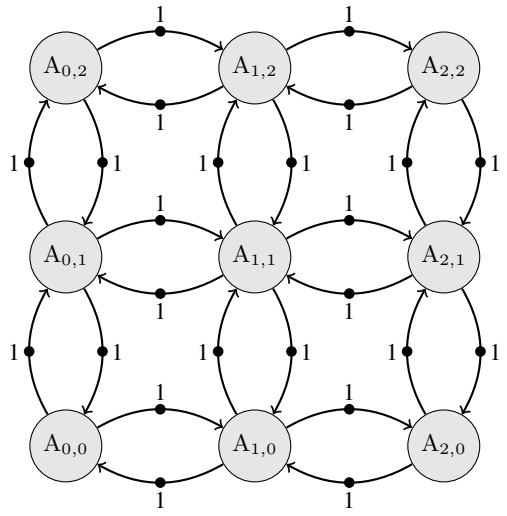

Figure 8. Sample actor graph for a simulation run with 3×3 actors. Each edge in the graph represents one channel. The number in the edge annotation represents the capacity n of the channel.

5. Related Work

X10's syntax has been influenced by the Scala programming language specification. Differently to X10, Scala innately includes already a basic actor model, and as of version 2.10, Scala employs the open source actor library Akka[3], which offers powerful concepts, such as hierarchical supervised and typed actors as well as actor migration upon failure, and to some extend also actor reallocation at runtime (*cluster sharding*). In comparison, our proposed implementation is directly written in X10, lightweight, and easier analyzable through the usage of state machines.

There has already been work done in the context of actor libraries in PGAS languages. Roloff et al. (2014) implemented a task library that served as precursor to actorX10. While their library already supports the notion of actors that are connected using queues with channels, there is no notion of ports, and there is no formalism provided. A dynamic migration of patches at run time is not supported.

Wei et al. (2012) proposed a streaming language that is being compiled down to X10. It also has a notion of actors and channels. However, it requires an external compiler, while our approach is a library that is compatible with the default X10 implementation.

Shali (2010) proposed an actor-oriented extension of the programming language Chapel. Their approach also enables the developer to distribute actors onto different shared-memory domains. However, instead of ports and channels, mailboxes are used. However, they do not consider a dynamic migration of actors between places.

Imam and Sarkar (2012) presented a unified concept of the actor model and the task-parallel async-finish model in Habanero-Java as well as Habanero-Scala, and demonstrated the benefits of such a combined model.

6. Further Work and Conclusion

In the future, we aim to introduce language extensions that enable the encoding of FSM semantics more formally. The application developer will be able to specify the states Q and transitions $t = (q, k, f, q') \in T$. Another point we are investigating is the annotation of non-functional requirements (quality numbers) for actors and channels. For example, one may want to annotate a channel with the maximum latency for a transfer operation such that the solution quality may still be guaranteed. Eventually, these constructs shall be integrated into X10 as part of our extensions for resource-aware programming (Hannig et al. 2011).

In this work, we presented a formalism and the X10-based framework implementation of an actor model. By making the interactions between concurrently executed strands of code explicit, the library simplifies reasoning about the concurrent behaviors and data flows in application codes significantly. We showcased two case studies. The first is an object detection chain from the domain of image processing, and the second is a distributed simulation for the shallow water equations. These examples serve to demonstrate the applicability of our approach as well as the synergy effects that may be gained by fusing the APGAS paradigm with an actor-based programming model. Further information about actorX10 is available at `http://actorx10.invasive-computing.org/`.

[3] `http://akka.io`

Acknowledgments

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

G. A. Agha. ACTORS: A model of concurrent computation in distributed systems. Technical Report AITR-844, MIT Artificial Intelligence Laboratory, June 1985.

A. Breuer and M. Bader. Teaching parallel programming models on a shallow-water code. In *Proceedings of the 11th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 301–308. IEEE Computer Society, 2012. doi: 10.1109/ISPDC.2012.48.

F. Hannig, S. Roloff, G. Snelting, J. Teich, and A. Zwinkau. Resource-aware programming and simulation of MPSoC architectures through extension of X10. In *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 48–55. ACM Press, June 2011. doi: 10.1145/1988932.1988941.

C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

S. M. Imam and V. Sarkar. Integrating task parallelism with actors. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 753–772. ACM, Oct. 2012. doi: 10.1145/2384616.2384671.

A. Pöppl and M. Bader. SWE-X10: An actor-based and locally coordinated solver for the shallow water equations. In *Proceedings of the Sixth ACM SIGPLAN X10 Workshop (X10)*. ACM, June 2016.

S. Roloff, F. Hannig, and J. Teich. Towards actor-oriented programming on PGAS-based multicore architectures. In *Workshop Proceedings of the 27th International Conference on Architecture of Computing Systems (ARCS)*, pages 1–2. VDE Verlag, Feb. 2014. ISBN 978-3-8007-3579-2.

A. Shali. Actor oriented programming in Chapel. `http://chapel.cray.com/education/cs380p-actors.pdf`, 2010. Accessed: 2016-04-27.

K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. FunState – an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):524–544, Aug. 2001. doi: 10.1109/92.931229.

H. Wei, H. Tan, X. Liu, and J. Yu. StreamX10: A stream programming framework on X10. In *Proceedings of the ACM SIGPLAN X10 Workshop (X10)*, pages 1:1–1:6. ACM, 2012. ISBN 978-1-4503-1491-6. doi: 10.1145/2246056.2246057.

A. Weichslgartner, D. Gangadharan, S. Wildermann, M. Glaß, and J. Teich. DAARM: Design-time application analysis and run-time mapping for predictable execution in many-core systems. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 34:1–34:10, Oct. 2014. doi: 10.1145/2656075.2656083.