Finding sufficient conditions to guarantee freedom from deadlock when using futures in X10 programs

Research Paper

Henk Erik van der Hoek
h.e.k.vanderhoek@student.utwente.nl

ABSTRACT
X10 is a new object oriented programming language developed by IBM. X10 extends sequential Java, i.e. Java without support for threads, with new concurrency primitives. Asynchronously executed expressions, or futures, are one of these new concurrency primitives. The incorrect use of futures may however lead to deadlock situations. This paper formulates a model to describe the interaction between futures and gives a pattern to recognize deadlock situations in this model. Finally, this paper gives an algorithm to recognize this pattern in a subset of the X10 language in an automated manner.

1. INTRODUCTION
Over the last decades, hardware designers have found ways to increase processor power year after year. Therefore the design of a processor has been improved in three main areas: increasing clock speeds, optimizing execution flow and improving cache structures. Unfortunately, the end is near. Hardware engineers are reaching physical limits. CPUs running on more than 4Ghz are still rare although Intel released a 2Ghz processor back in 2001.

Chip manufacturers have found other ways to improve the performance of their products. The performance in the near term future will grow by making chips with multiple cores. A core is a CPU in itself, with its own registers, cache and operations. Today a chip may be composed of two or four cores but this number is likely to increase. However, a dual core 2Ghz processor is not the same as a 4Ghz single core processor in terms of performance, because most software, especially desktop software, has been designed with a single (core) processor in mind.

Making software for multiple core systems requires writing in languages with support for concurrent programming. In a concurrent program several processes are executed in parallel. This can be implemented by multitasking, in which processes are switched frequently, and by parallel execution in multiple cores. Concurrent programming is not new. Operating systems, for example, are exploiting concurrent programming in order to run multiple programs simultaneously. And webservers can handle multiple requests at the same time. So although concurrent programming has been around for years, with the advent of multiple core processors, concurrent programming skills becomes necessary for every application developer.

However, writing concurrent programs is difficult. People often find the concepts of concurrent programming hard to grasp. Bugs may appear only under heavy load or may even disappear after adding debug code. Several different techniques exist to support the development of concurrent programs. These techniques include model checking and synchronization policy encapsulation [1].

Another approach is to provide high level language constructs to support the development of concurrent programs. This approach is taken by the developers of the X10 language. X10 is an experimental language, developed by IBM Research, designed for high productivity programming of scalable applications. X10 is a statically typed, class based object oriented programming language [2].

1.1 The X10 programming language
X10 extends sequential Java, i.e. Java without support for threads, locks and synchronization, with (among others):

- A partitioned global address space
- New concurrency primitives
- A rooted exception model

1.2 A partitioned global address space
One major difference between Java and X10 is the use of a partitioned global address space [3]. The address space for mutable objects is divided over several places. The number of places is fixed during runtime. Places are a virtual concept and the mapping between virtual places and physical nodes is done during deployment. Using virtual places instead of physical nodes as the level of abstraction has the advantage of being able to write programs without a specific hardware configuration in mind.

Every activity (the X10 equivalent of a thread) starts in exactly one place. It remains in the same place during its lifetime. An activity can only access mutable objects in its own place. Accessing and manipulating mutable objects in another place can be done by creating a new activity at the remote place.

Mutable objects are always local to one place, whereas immutable data can be accessed directly by every activity. The design of the partitioned global address space is illustrated in figure 1.

1.3 New concurrency primitives
New concurrency primitives are another major difference between Java and X10. These new primitives use a higher level of abstraction compared to the thread and locks used in Java. We will briefly explain these new concurrency primitives.
async & finish An activity can spawn new activities. New asynchronous activities are spawned by the async (P) S primitive. The statement S will be executed in a place specified by the place designator P. The place designator (P) may be omitted if the place can be unambiguously inferred by static analysis of the data accessed by the statement S. An example of this is shown in program 1.

Program 1 Example of the async and finish primitives
class Sum {
    protected int oddSum, evenSum = 0;
    public Sum () {
        final int c = 100;
        finish {
            async {
                for (int i = 1; i <= c; i+=2)
                    oddSum += i;
            }
            for (int i = 0; i <= c; i+=2)
                evenSum += i;
        }
        System.out.println("odd: " + oddSum + ", even: " + evenSum);
    }
    public static void main (String[] a) {
        new Sum ();
    }
}

The sum for the odd and the even numbers are calculated concurrently in this example. As shown above, spawning activities in X10 is much simpler than creating threads in Java. This increases the productivity of the programmer.

It is sometimes necessary to wait for the termination of a child activity before proceeding. For example, we can only write the calculated sums to the console after both the even and the odd sum are calculated. The finish primitive is used for this purpose. The finish primitive will suspend the activity until all the activities that are recursively spawned within the scope of this finish primitive terminate. The main program is implicitly surrounded by a finish primitive. This ensures that the main activity can only terminate if all other activities are terminated.

Futures Futures are asynchronously executed expressions. Futures are created using the future primitive. The result of a future can be obtained by using the force operation. This operation will block until the expression has been calculated. An example is shown in program 2.

Program 2 Example of the future primitive
future<int> f = future { expensiveToExecute () ;
    .. something else ..
    int result = f.force ();
}

Clocks Clocks add a new way to synchronize activities. The finish primitive and force operation only allow synchronization on the termination of activities. Clocks allow activities to synchronize between arbitrary statements. Every clock is registered to zero or more activities. A clock advances to next phase if and only if all its currently registered activities have quiesced. An activity can quiesce by either performing a resume operation on the clock or by terminating.

Program 3 Example of the clock primitive
class ClockExample {
    public class ClockExample {
    
    }
    public static void main (String[] a) {
        new ClockExample ();
    }
    public ClockExample () {
        final clock c = clock.factory.clock ();
        for (int i = 0; i < 10; i++)
            async clocked (c) {
                process (c, new Particle ());
            }
    }
    public void process (final clock c, Particle p) {
        p.updateAcceleration ();
        c.resume ();
        p.updateSpeed ();
        c.resume ();
        p.updatePosition ();
    }
}

This enables programmers to write activities that progress in phases. An example of this is shown in figure 3. In this example one activity is used to calculate the acceleration, speed and position of a particle. Imagine that one should know the acceleration of all particles before it is possible to calculate the speed of a particle and that one should know the speed of all particles before it is possible to calculate the position position of a particle. Clearly, synchronization between the activities is needed, a clock is used
Atomic blocks avoid the use of locks when multiple activities can manipulate shared data. A traditional Java program would synchronize on the the implicit lock of the list by using the synchronized keyword. This may however cause the program to reach a deadlock state. Because an atomic block does not require locking there is no risk of reaching a deadlock state.

Conditional atomic blocks Conditional atomic blocks are similar to unconditional atomic blocks. However, entrance to a conditional atomic block is guarded. The execution of when (c) S suspends until the guard c becomes true. The guard c and the statement S are conceptually executed in an single step, other activities are suspended during it’s execution. A conditional atomic block can be used for example to create a bounded buffer, as shown in program 5.

Program 5 Example of the when primitive

```java
class Buffer {
    const int SIZE = 3;
    protected Item[] array = new Item[SIZE];
    protected int getPointer = 0;
    protected int usedSlots = 0;

    public Item get () {
        when (usedSlots != 0) {
            Item item = array[getPointer];
            getPointer = (getPointer + 1) % SIZE;
            usedSlots--;
            return item;
        }
    }
}
```

Just like unconditional atomic blocks, spawning new activities or performing potential blocking operations in unconditional atomic blocks is not allowed. Because a unconditional atomic block is a potential blocking operation, nesting conditional atomic blocks is impossible.

1.4 A rooted exception model

In a Java program it is not possible to catch an exception thrown by the run method of a thread. Instead, the thread will terminate and the exception will be written to the System.err stream. The X10 language uses a rooted exception model to catch exceptions thrown by abruptly terminating activities. Therefore, a parent child relationship between an activity and the activity that it creates is defined. Usually, the child and the parent activities continue to execute in parallel. However, the parent activity has to wait in order to catch any exception thrown by the child activity. Fortunately, we can easily synchronize on the termination of an activity by using a finish primitive. An example of the rooted exception model is shown in 6, the exception thrown in the child activity is caught by the parent activity.

Program 6 Example of the rooted exception model

```java
// Parent activity
try {
    finish async {
        // Child activity
        throw new x10.lang.Exception ();
    }
} catch (x10.lang.Exception e) {
    System.out.println ("Exception caught");
}
```

2. PROBLEM STATEMENT

This paper proposes to exploit the semantics of the new X10 concurrency primitives to conduct static deadlock analysis on X10 programs. We will use the following definition of deadlock: “a set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.” [5] This definition is more general than the definition by Coffman [6] which deals explicitly with processes and resources.

[7] already proves that a program containing only async, finish and atomic concurrency primitives will never deadlock. As a result, a user of the X10 programming language can write a program with these three primitives without worrying about deadlock. Futures, however, can cause a program to reach a deadlock state. An example is shown in code listing 7. This program will reach a deadlock state because future Y forces future y to finish and vice versa.

In a presentation [8] Sarkar already states that restricted use of futures can guarantee freedom from deadlocks. He also gives one sufficient condition:

- Ensure that the activity that creates the future also performs the force() operation

Enforcing this condition however restricts the expressiveness of the language. We would like less restrictive conditions. This results in our problem statement:

—Which other conditions are sufficient to guarantee freedom from deadlock when using futures in X10 programs? And can we conduct static analysis to find possible violations of these conditions?—
We will focus on deadlocks caused by futures. Unrestricted use of conditional atomic expressions, available in X10 as the when primitive, may also result in reaching a deadlock state. With static analysis we mean finding possible violations at compile time.

The following questions will be addressed:

- Which situations will lead to deadlock?
- Is there a pattern in these situations?
- Is there an algorithm to find this pattern?

The remainder of this paper is organized as follows: First, we will describe and explain a model to reason about X10 programs with futures. Thereby introducing a pattern to recognize deadlock situations in this model. Secondly, we will describe an algorithm to find this pattern in a subset of the X10 language in an automated manner.

3. A MODEL TO RECOGNIZE DEADLOCK SITUATIONS

Every X10 program can be modeled as a dependency graph. A dependency graph has three different types of edges:

- Normal edges: A normal edge connects successive sequential instructions.
- Spawn edges: Every instruction that creates a new activity will create a spawn edge from that instruction to the first instruction of the new activity.
- Dependency edges: Every instruction that blocks until the termination of another activity will create a dependency edge. E.g. there is an edge from instruction i to instruction j if and only if i must finish before j can start execution.

We require that all instructions, except instructions with an incoming dependency edge, are non blocking.

A cycle in this dependency graph means that the program is prone to deadlock. A topological ordering in which all the instructions can be executed must exist. There is no topological ordering in a dependency graph with a cycle. Therefore, a cycle in the dependency graph may result in reaching a deadlock state.

We will explain our approach with two examples. We can model the program DeadlockExample shown above as a dependency graph. This dependency graph is shown in figure 2. The activities are organized into three lanes: the main activity lane, x future lane and the y future lane. We will explain the labeled edges:

a) The creation of future x. Remember that every future creates a new activity.

b) The creation of future y.

c) Call to the force method on y by the first future. This creates a dependency edge to the last instruction of the y activity.

d) Call to the force method on x by the second future.

The DeadlockExample program will always reach a deadlock state. It is however possible to create a program that only reaches a deadlock state after some unlucky timing by the activity scheduler. An example of such a program is given in listing 8

The program shown below will only reach a deadlock state if the third future is spawned and assigned to x (line 13) before the force operation is executed in the second future (line 19).

A non deterministic deadlock situation as shown in program 8 is caused by initializing the same future variable more than once. The program is vulnerable to deadlock if at least one initialization leads to a cycle in the dependency graph. Deadlock situations which only manifest themselves in a non deterministic manner are notoriously hard to debug. Therefore we recommend, as a best practice, to always declare a future variable as immutable. X10 allows this by means of the final keyword.

Using a dependency graph we have created a model of the runtime behaviour of a X10 program. This model is useful to reason about possible deadlock situations. However, it may not always be possible to create this dependency graph at compile time. Therefore, in the next section we will place restrictions on the X10 language to enable compile time deadlock analysis.

---

**Program 7** Trivial deadlock example

```java
public class DeadlockExample {
    protected final future<int> x;
    protected final future<int> y;

    public DeadlockExample ()
    {
        x = future { m1 ()};
        y = future { m2 ()};
    }

    public int m1 ()
    {
        return y.force();
    }

    public int m2 ()
    {
        return x.force();
    }

    public static void main (String[] args)
    {
        new DeadlockExample();
    }
}
```

---

**Figure 2:** Runtime model of the DeadlockExample program
4. THE ALGORITHM

First, we start will describing the main idea of the algorithm. Secondly, we will give a more detailed explanation of a few key elements of the algorithm. Finally we will use an example to walk through the algorithm step by step.

4.1 Assumptions

We will assume the following restrictions on the X10 language to enable compile time deadlock analysis. Additionaly these restrictions are used to keep the scope of our project under control.

- A subset of the X10 language, namely Java extended with X10 futures, will be used. We do not allow the use of any other X10 feature.
- When a force operation is executed on a future variable or when a future is initialized it should be unambiguous to which future declaration this variable refers.
- We only consider programs with one class definition. The class can have just one constructor and the use of method overloading is not allowed.

4.2 Global description

The algorithm constructs a dependency graph as described in section 3. The algorithm starts by making a subgraph for each method. These subgraphs are then linked together to form the complete graph. The algorithm is shown in pseudo code in program 9 and will now be discussed step-by-step:

1. By using ANTLR and AST is created from the X10 source code.
2. The createSubgraphs function walks through the AST and creates subgraphs of the dependency graph for each method. This function is discussed in more detail below.
3. The final dependency graph is now created by merging the subgraphs together with the use of the resolve function. This function will also be discussed in more detail below.
4. A cycle in this final dependency graph implies the possibility of a deadlock situation.

Program 9 High level overview of the algorithm in pseudo code

```java
AST tree = parse (source)
List<Graph> subgraphs = createSubgraphs (AST)
Graph graph = resolve (subgraphs)
if (graph.hasCycles ()
   Deadlock possible
```

4.3 The createSubgraphs function

The createSubgraphs function returns a list with one subgraph for each method. A subgraph can contain the following node types:

- **MethodEntry** Every subgraph starts with a MethodEntry node.
- **MethodCall** This node indicates a method call at the corresponding position in the program. The name of the called method is stored in the MethodCall node.
- **ForceCall** This node indicates a force operation is being executed on a future at the corresponding position in the program. The name of the future is stored in the FutureCall node. The dependency edge from this node to the corresponding future is added later in the resolve function.
- **FutureInitialization** and **FutureStart** During the initialization of a future two nodes are added to the subgraph. A FutureInitialization node is added as a sequential step in the current activity and a FutureStart node is added as the first node of the future activity. In addition to this and edge is added from the FutureInitialization node to the FutureStart node. This edge is depicted in figures 2 and 3 as the striped edges.

In program 10 the function createSubGraphs is shown in pseudo code. A visitor pattern had been used to walk through the AST.
Program 10 The createSubgraphs function

```java
function createSubgraphs (AST) {
    TreeVisitor v = new TreeVisitor()
    v.parseProgram (AST)
    return v.subgraphs
}
```

class TreeVisitor extends EmptyVisitor {
    Graph currentSubgraph
    Node currentNode
    List<Graph> subgraphs

    function parseMethod (MethodNode m) {
        currentSubgraph = new Graph()
        currentNode = new MethodEntryNode (m.name)
        currentSubgraph.add (currentNode)
        parseStatements (m.statements)
        subgraphs.add (currentSubgraph)
    }

    function parseMethodCall (MethodCall m) {
        Node node = new MethodCallNode (m.name)
        node.addIncomingNode (currentNode)
        currentSubgraph.add (node)
        currentNode = node
    }

    function parseForceCall (ForceCall m) {
        Node node = new ForceCallNode (m.future)
        node.addIncomingNode (currentNode)
        currentSubgraph.add (node)
        currentNode = node
    }

    function parseFutureInitialization (FutureInitialization m) {
        Node tmp = currentNode
        FutureInitializationNode node = new FutureInitializationNode (m.future)
        node.addIncomingNode (currentNode)
        currentSubgraph.add (node)
        lastFutureStartNode = new FutureStartNode (m.future)
        currentSubgraph.add (lastActivityStartNode)
        currentNode = lastFutureStartNode
        lastActivityStartNode.addIncomingNode (node)
        parseExpression (m.expression)
        currentNode = node
    }
}
```

4.4 The resolve function

The resolve function creates a dependency graph from the different subgraphs. In order to construct this graph the following steps are executed:

1. A new graph result is created. All nodes from all subgraphs are added to this new graph.
2. An edge is added from each MethodCallNode to the corresponding MethodEntryNode.
3. An edge is added from each ForceCallNode to the last sequential node of the corresponding future. These edges are depicted as the dotted edges in figures 2 and 3.

The function resolve is given in pseudo code in program 11. The resolve function uses an extra function lastSequentialNode. This function returns the last node within a certain activity.

Program 11 The resolve function

```java
function resolve (List<Graph> subgraphs) {
    // Step 1
    Graph result = new Graph()
    foreach Graph subgraph : subgraphs
        result.add (subgraph)
    // Step 2
    foreach MethodEntryNode n : result.nodes
        foreach MethodCallNode m : result.nodes
            if (n.getMethod() == m.getMethod())
                n.addIncomingNode (m)
    // Step 3
    foreach FutureStartNode n : result.nodes
        foreach ForceCallNode m : result.nodes
            if (n.getFuture() == m.getFuture()) {
                Node node = lastSequentialNode (n)
                node.addIncomingNode (m)
            }
    return result
}
```

4.5 Example

We walk through the complete algorithm with the program given in program 7. In the first step an AST is constructed which then serves as input for the createSubgraphs function. This function is created using the subgraphs shown in figure 4. The different node types are indicated with the use of different background patterns:

In the next step a graph is created by calling the resolve function on the list of subgraphs. By doing this every MethodCall node is connected to the corresponding MethodEntry node. Then every ForceCall node is connected with the last nodes of the corresponding future.

The result of the resolve function is shown in figure 2. In the last step of the algorithm the graph is checked for any cycles. The graph in our example contains a graph which leads us to conclude a deadlock situation can occur in this program.
5. DISCUSSION

The algorithm described above has been implemented with the use of ANTLR. An ANTLR TreeParser was used to walk through the AST. The implementation has been tested with 16 test cases, two of which are the programs in 7 and 8. In section 4.1 we made a number of assumptions. We will now discuss which adaptations to the algorithm are required in order for us to remove these assumptions or at least loosen them. First, we will first look at adding other X10 concurrency primitives. Secondly, we look how to enable multiple constructors and method overloading.

5.1 Atomic

The specification of the X10 language dictates one cannot create new activities or execute potentially blocking operations within an unconditional atomic block. From this it follows that all instructions within one unconditional atomic block can be modeled as one node in the dependency graph. This will not create new possibilities for cycles in the dependency graph.

5.2 Async

The instructions in an async block are executed in a new activity. We can model this by adding an Async node in the current activity. We then add a spawn edge from this Async node to the first instruction within the scope of the Async primitive. This approach is also described in [7]. However, an async block does not lead to a dependency edge, so an asynchronously executed statement cannot lead to new cycles in the dependency graph.

5.3 Finish

A finish primitive will suspend the current activity until all the activities that are spawned within the scope of this finish primitive terminate. This can be modeled by adding a Finish node in the dependency graph and then adding a dependency edge from this Finish node to the last sequential node of every activity created within the scope of this finish block. Further investigation is necessary to see if adding this new dependency edge introduces new cycles.

5.4 Multiple constructors and method overloading

The algorithm described above cannot handle multiple constructors and method overloading. However support for this functionality can be added by not only storing the method or constructor name in the MethodCall and the MethodEntry nodes but also the order and type of the formal parameters.

6. RELATED WORK

This section elaborates on the relationship between this paper and work done in the past. [4] already proves that any program written with async, finish, atomic and clocks is free of logical deadlock. Programs are free of logical deadlock if they are guaranteed to never reach a deadlock state in case of unbounded computation, memory and communications resources. However, physical nodes have limited resources. Freedom of physical deadlock for any program written with async, finish and atomic is proven by [7]. As a consequence, a user of the X10 programming language can write a program with these three primitives without worrying about deadlock. Our paper shows that deadlock with futures is possible. [7] uses a directed acyclic graph to model X10 programs with async, finish and atomic primitives.

7. CONCLUSION AND FUTURE WORK

The main contribution of this paper lies in formulating a model to describe the interaction between futures in an X10 program at runtime. Secondly, this papers gives a pattern to recognize a deadlock situation in this model.

We restricted the X10 language to enable compile time analysis. Traceability of every future variable was a necessary restriction. This makes it possible for us to associate a call to a force operation with a future initialization. Enforcing this restriction however reduces the expressiveness of the language. Using data flow analysis may be another way to associate a call to a force operation to the right future initialization. More information about dataflow analysis can be found in [9]. A deadlock detection based on dataflow analysis would have to handle more realistic X10 programs, thereby increasing its value as a practical tool for application programmers.

REFERENCES


