Resource Sharing Conflicts Checking in Multithreaded Java Programs

Nadezhda Baklanova, Louis Féraud, Martin Strecker

IRIT (Institut de Recherche en Informatique de Toulouse)
Université de Toulouse [⋆]

Abstract. We present a tool for analysing resource sharing conflicts in multithreaded Java programs. Java programs are translated to timed automata models verified afterwards by the Uppaal model checker. Analysed programs are annotated with timing information indicating the execution duration of a particular statement. Based on the timing information, the analysis of execution paths is performed, which gives an answer whether resource sharing conflicts are possible in a multithreaded Java program. If the analysis succeeds, time-consuming resource locks may be eliminated from the Java program.

Keywords: timed automaton, Java, multithreading, deadlock, resource sharing conflict, Uppaal.

1 Introduction

Parallel computations quickly develop nowadays, and the problem of debugging multithreaded programs arises. It is known to be a very difficult problem for a software developer, and thorough testing cannot discover all the fatal errors in a program due to unpredictability of execution. One type of errors are resource sharing conflicts. In order to avoid these errors one may want to guarantee that the same resource is not accessed by different threads at the same moment. If one concentrates on this aspect, the behavior of a program may naturally be modelled by timed automata and then one may find error-prone places in the program using a timed automata model checker.

In order to achieve this goal, we need to enrich the Java language with annotations indicating time information. The annotations show how much time is required for executing a statement and, consequently, how much time is required for a thread to have an exclusive access to a resource. It allows to avoid usage of `synchronized` statements in programs after verifying that no resource is used simultaneously by two or more threads. Based on an annotated Java program, a timed automaton is generated taking into account time required for execution of statements. Finally, we check the generated automaton for possible resource sharing conflicts using Uppaal model checker in the generated automaton. The transformation sequence is shown in the diagram 1 below.

[⋆] Part of this research has been supported by the ANR project *Verisync*

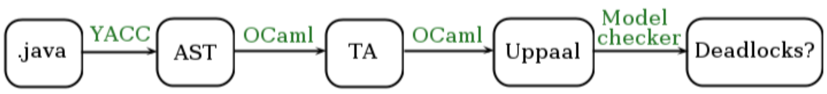


Fig. 1: Java program verification process

The overall aim is to replace a lock-driven protocol for resource conflict avoidance by a time-driven approach. If a check on the abstract level of timed automata indicates no resource access conflict, then also the underlying Java program can be expected to run without conflicting access to resources. In this case, locking even becomes superfluous. If, however, the check fails, nothing can be said about the behaviour of the Java program when executed, just like for an ill-typed program.

The purpose of the present paper is to sketch the overall approach and define the correspondence between Java and timed automata, without giving a proof of the soundness of the abstraction, which remains for future work.

Related work

There are several tools for scheduling analysis of real-time tasks. Verification of scheduling strategies with timed automata is considered in [6]. However, it operates with a high level notion of abstract tasks and does not look inside the source code. The authors perform schedulability analysis with the fixed-priority scheduling strategy by translating a system to be verified to a timed automaton and verifying it afterwards with Times tool.

Another approach is used in [3,4]. Here Java source code analysis is performed using timed automata. An automaton is generated from the source code, and every statement is mapped to a certain part of the automaton. The translation procedure described in [3] contains an inconsistency between Java semantics and model semantics of the generated system. Locking mechanism is implemented in Uppaal model as monitors which are incremented when a lock is acquired and decreased when a lock is released. However, there are no checks before acquiring the lock in the model. It makes the situation when two threads have locked the same resource at the same time possible, but it does not correspond to the JVM behavior.

A translation from SystemC to Uppaal is presented in [7]. One of the purposes of this work is to give a formal semantics to the (only informally defined) SystemC language. The differences between SystemC and Java, as far as the translation to Uppaal is concerned, still has to be explored.

In [5] schedulability analysis of a set of tasks is performed by exhaustive search combined with Uppaal for determining when the search is complete. Again, the internal structure of tasks is not taken into account which makes impossible to do conclusions about thread interactions. Brute force is not the suitable tactics for verifying large systems. The authors listed the limitations they had encountered: lack of memory and lack of Uppaal integer range. For small system this approach works well but large system verification may require optimized algorithms.

The paper [8] contains schedulability analysis of multithreaded SCJ (Safety Critical Java) programs and takes resource sharing into account. Resources are considered to be locked during the whole execution of a task. Analysis is performed by UPPAAL modeling taking into account the resource locks. However, this analysis does not model the exact behavior of a program since developers usually try to minimize the length of critical sections thus critical sections can be treated more efficiently.

2 Sample usage

Before describing our approach more in detail, we illustrate it here with a small example. The outermost class containing the `main` method is called `Threads`. Two threads `t1`, `t2` are declared in the `main` method. `Run1` and `Run2` are nested classes inside the `Threads` class implementing the `Runnable` interface.

```
Threads ts;
Run1 r1;
Run2 r2;
Thread t1,t2;
ts=new Threads();
r1=ts.new Run1();
r2=ts.new Run2();
ts.res=new Res();
t1=new Thread(null,r1,"t1");
t2=new Thread(null,r2,"t2");
```

Methods called on the thread start are the following:

```
private class Run1 implements Runnable{
  public void run(){
    int value,i;
    //@0@//
    i=0;
    while(true){
      synchronized(res){
        //@1@//
        // acts like a random generator
        // producing an arbitrary natural number
        value=Calendar.getInstance().get(Calendar.MILLISECOND);
        //@ 5 @//
        res.set(value);
      }
      try{
        //@10@//
        Thread.sleep(10);
      }
      catch(InterruptedException e){
        System.out.println(e.getMessage());
      }
      //@0@//
      i++;
    }
  }
}

private class Run2 implements Runnable{
  public void run(){
    int value,i;
    //@0@//
    i=0;
    try{
      //@9@//
      Thread.sleep(9);
    }
    catch(InterruptedException e){
      System.out.println(e.getMessage());
    }
    while(true){
      synchronized(res){
        //@ 4 @//
        value=res.get();
      }
      try{
        //@10@//
        Thread.sleep(10);
      }
      catch(InterruptedException e){
        System.out.println(e.getMessage());
      }
      //@0@//
      i++;
    }
  }
}
```

Here `res` is a resource declared in the main class. Calls of the `res.get()` and `res.set()` methods are "actions" using the locked resources which are preceded by timing annotations showing the amount of time the "action" requires. During the translation they are considered to be abstract statements inside the locked region taking n time units for execution.

Both threads do some "actions" requiring exlusive lock with the resource and then sleep for some time. The `synchronized` statements are a potential source of resource sharing conflict if both threads wake up simultaneously. One can see the resource access conflict in the execution timeline 2 showing times when the threads demand an exclusive lock for the resource.

`res` has type `Res` which is a simple class allowing to read and write to one field.

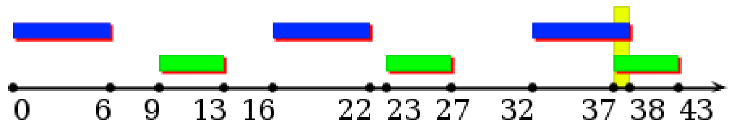


Fig. 2: Execution timeline. `t1` is blue, `t2` is green. Conflict between 37 and 38 is shown in yellow.

```java
class Res{
  private int i;
  public void set(int j){
    i=j;
  }
  public int get(){
    return i;
  }
}
```

The generated timed automata are shown in Figure 3.

The generated formula for model checking is

$$A[]check\_Threads(Threads\_monitor).$$

When performing verification, this formula is evaluated to false, and the generated trace for counterexample stops in states $aut\_Run1\_t1.annotated6$ and $aut\_Run2\_t2.annotated6$ during the third loop iteration at the time moment 38.

3 Translation from Java to Abstract Syntax Tree

Considering the idea of "extended Java" a possibility to write annotations for every Java statement is added to Java syntax. These annotations contain time required for executing the whole block or statement next of the annotation. The time in the samples is in abstract time units but one can annotate program with real values in microseconds based on computer architecture, compiler version, running software etc.

Assumptions

The first thing is to generate abstract syntax tree (AST) of a program written in "extended Java". Unfortunately, standard Java annotations cannot be added to arbitrary statements. Even the latest extension of Java annotations implemented in JDK 7 [1] does not allow to annotate executable statements (assignments, loops, conditions etc.) which are of the most interest for us. For this reason we used self-written parser of the "extended Java" language. The parser is written with OcamlYACC and recognizes Java with several restrictions.

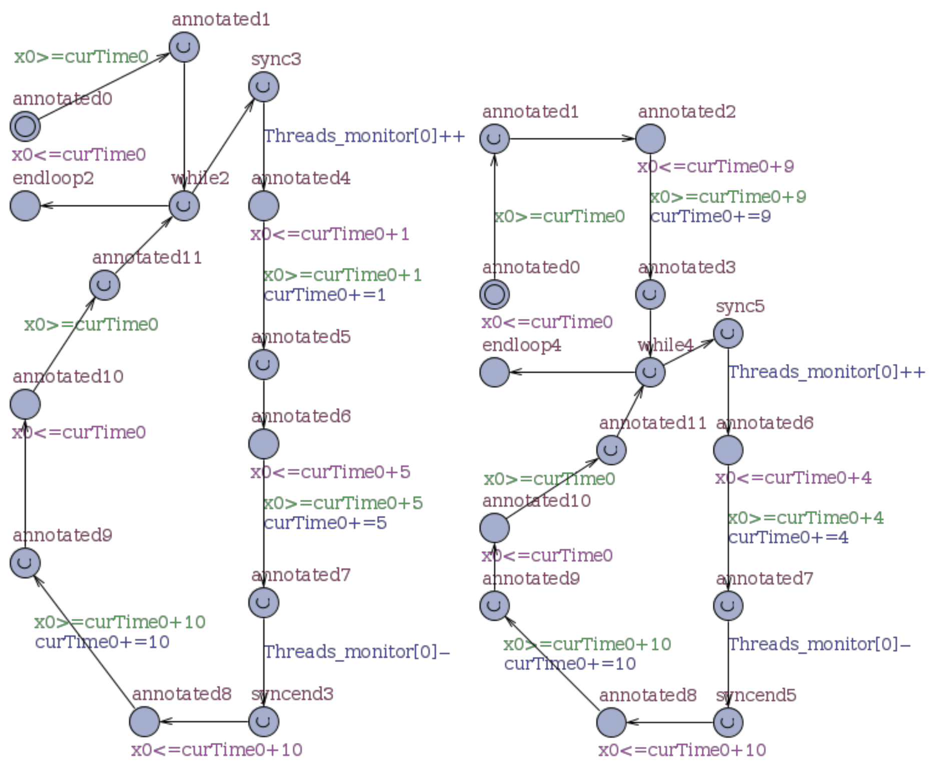


Fig. 3: Generated automata for threads a, b (two times) and c.

Method calls except initializing constructors are not translated therefore each method call used in a translated program is assumed to have a timing annotation. Cast operators are not supported for now. Since we perform static analysis, dynamic features are excluded, namely, arrays or references to `this` instead of specifying an object name explicitly. `try...catch` constructions can be parsed, however, code in the `catch` block is not translated, i.e. `try block1 catch block2` is considered to be equivalent to `block1`.

In order to make the AST generator simpler all used packages are supposed to be imported in the header of a program. Local variables must be declared in the beginning of methods before statements, and declaration statements cannot be combined with assignments. This assumption helps to avoid problems with scope of the variables declared in the middle of a method. Argument of the `synchronized` statement is assumed to be an explicit object name, not an expression. Moreover, reentrant locks are not allowed, i.e. when the same thread acquires lock of the same object several times. It is the developer's responsibility to avoid reentrant locks in the processed programs.

Threads are supposed to be declared in the `main` method which is an entry point of the program and must be declared in the first class of a program. The `main` method cannot contain any code except thread declarations, initializations and calls for starting the threads. Threads are assumed to be created with the constructor

```
Thread(ThreadGroup group, Runnable target, String name)
```

because it is the most general one, and `Runnable` object should implement `Runnable` interface, not be a subclass of `Thread`.

The parser produces AST from Java code as a set of OCaml objects.

4 Translation from Abstract Syntax Tree to Timed Automaton

Since the output of the parser is a structure of OCaml objects, the generator of timed automata from AST is written in OCaml. At the beginning it generates a set of OCaml objects representing a timed automaton, and after that the timed automaton is printed in format recognizable by Uppaal which is used for model checking. The abstract representation of a timed automaton as OCaml objects is based on the abstract definition of timed automaton and does not depend on a particular implementation of a model checker.

The Ocaml type for an automaton looks like

```
type ta = Empty
        |TA of (node list) * (urgent list) *
           (committed list) * (edge list) * start *
           final
```

Here `start` and `final` are start and final states of the timed automaton. Final state is required because a timed automaton is generated recursively, and it is necessary to determine where the previously generated parts of an automaton finish, although there is no such a notion in the definition of timed automata. Committed and urgent are state characteristics specific for Uppaal, however, they can be modeled by a standard timed automaton, i.e. they do not add anything new to the initial definition. Final states of the generated timed automata are always urgent, that means, they are not allowed to rest in these states for any time. One may find definitions related to timed automata in [2].

As method calls are not translated, only `run` methods of `Runnable` objects are translated to timed automata because they are the only methods which can contain executable code. Each thread declared and initialized in the `main` method is mapped to a separate automaton (template in the Uppaal terminology). The system has one global clock and a global array of object monitors.

An object monitor is an integer variable which is incremented when this object is locked and decremented when the lock is released. In Uppaal model monitors are implemented as an array of integers, each object is encoded as an array item; consistency of indices is watched by the automata generator.

All statements except the annotated ones are assumed not to take any time for execution; for this reason all the states without timing information are made urgent in Uppaal model. Time is not allowed to pass when an automaton is in urgent state.

Statements annotated with timing information are treated as a "black box" and are supposed not to produce side effects. With side effects a possible situation is when a thread tries to access an object field which is locked by another thread. In this situation JVM makes the thread waiting until the lock is released, however, our translation does not notice this delay and produces an incorrect automaton. Annotation is considered as execution time of the block next to the annotation. If some resources are locked inside the annotated block, they must be released in the same block, i.e. sets of locked resources before annotated statement and after must coincide.

Each template has a local integer variable `curTime` representing the time when an automaton entered the state corresponding to an annotation statement. Annotations in Java programs contain relative time but timed automata use global time, therefore we need to keep track how much time has passed since a program has been started. The only statements allowing time to increase are annotated statements. Suppose t was the global time when an automaton entered a state corresponding to an annotated statement. When it leaves this state, model time and `curTime` variable are increased by n where n is the timing annotation of the statement. `curTime` may be different in different automata but it always concurs with the global time when its corresponding automaton is executing.

Basic items for building timed automata are statements: each statement is translated into a part of timed automaton. The AST can contain the following statements:

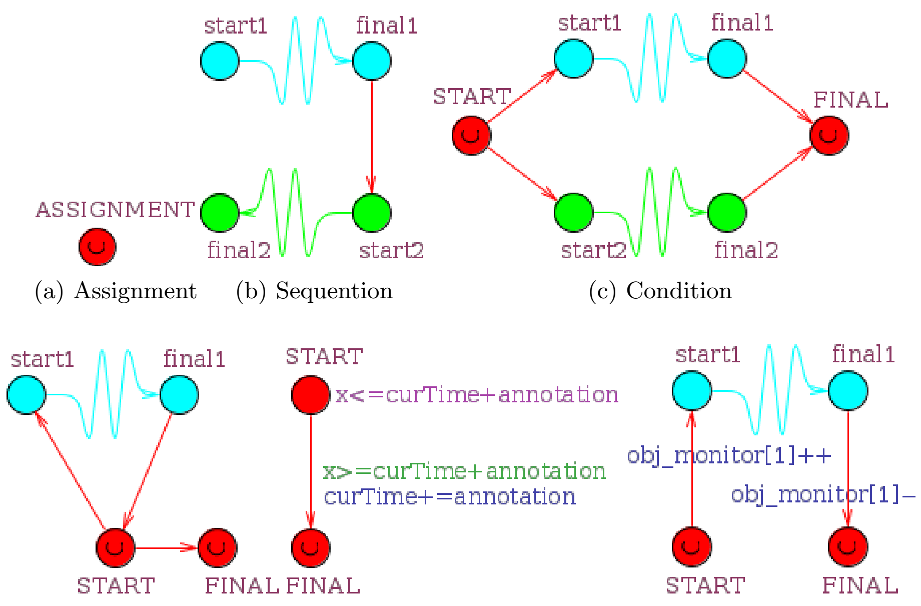
```
type  stmt =
    Skip
      (* empty statement *)
  | Assign of var * expr
      (* assignment statement: a=5+4; *)
  | Seq of stmt * stmt
      (* seqence of two statements: a=4; b=5; *)
  | Cond of expr * stmt * stmt
      (* conditional statement: if(a=1) {...} else
         {...} *)
  | While of expr * stmt
      (* loop statement: while(a<5){...} *)
  | CallC of callExp
      (* method call: a.toString(); *)
  | Return of expr
      (* value return: return a; *)
  | AnnotStmt of annot * stmt
      (* annotated statement: //@ 4 @// a=3; *)
  | SyncStmt of expr * stmt
```

```
    (* synchronized statement: synchronized(a){...}
       *)
```

Translation from AST to timed automata skips field and variable declarations because they do not change the state of a program. At the same time, all the objects declared in the main program class get a monitor.

Boolean conditions inside `while` and `if` statements are not translated. It is assumed that any of the two possible ways can be taken during runtime.

`Skip` and `Return` statements are mapped to an empty automaton because they do not influence the state of a program.



(a) Assignment (b) Sequention (c) Condition

(d) Loop (e) Annotation (f) Locking

Fig. 4: Automata generated from basic statements. Red elements are newly added by the translation of the corresponding Java statement into TA.

The rules for mapping other AST statements to the parts of a timed automaton are the following (see Figure 4):

– `Assign(v,e)`: add an urgent *ASSIGNMENT* state which is start and final for this automaton.
– `Seq(c1,c2)`: suppose $a1$ and $a2$ are the automata for $c1$ and $c2$ respectively, add an edge from $final1$ to $start2$, $start1$ is the start state, $final2$ is the final state.
– `Cond(e,c1,c2)`: suppose $a1$ and $a2$ are the automata for $c1$ and $c2$ respectively, add two urgent states *START* and *FINAL*, which are the start and

final states of the new automaton, and edges from $START$ to $start1$ and $start2$, from $final1$ and $final2$ to $FINAL$.
- `Loop(e,c1)`: suppose $a1$ is the automaton for $c1$, add two urgent states $START$ and $FINAL$, which are the start and final states of the new automaton, and edges from $START$ to $start1$, from $final1$ to $FINAL$ and from $START$ to $FINAL$.
- `CallC(ce)`: currentrly not translated
- `AnnotStmt(d,c1)`: add two states $START$ and $FINAL$, and an edge from $START$ to $FINAL$, which are the start and final states of the new automaton. $FINAL$ is urgent. x is the global clock, $curTime$ is the local variable. $START$ has an invariant $x <= curTime + d$, the edge has a guard $x >= curTime + d$ and an update action $curTime+ = d$. This combination of guard and invariant ensures that timed automaton will be in this state exactly d time units. In our model it means that the annotated statement requires d time units for execution.
- `SyncStmt(e,c1)`: suppose $a1$ is the automaton for $c1$, add two urgent states $START$ and $FINAL$, which are the start and final states of the new automaton, and edges from $START$ to $start1$, from $final1$ to $FINAL$. We assume that expression e is a field declared in the outermost class. Its monitor is incremented when the edge from $START$ to $start1$ is taken and decremented when the edge from $final1$ to $FINAL$ is taken.

Model checking

Our initial goal was to check whether there are possible resource sharing conflicts during program execution. UPPAAL provides an ability to check properties of timed automata expressed with CTL formulas. The interesting property is whether for all paths through timed automaton for any state in this path any of the object monitors do not have more than one lock or, with UPPAAL syntax, $A[]check(obj\_monitors)$ where `obj_monitors` is an array with object monitors of the main program class, and `check` is a function assuring that all the monitors are less than 1. Thus, if `check` is evaluated to true, no resource is locked by two or more threads simultaneously. That means threads do not wait for a resource unlock during program execution in JVM. If the property is satisfied, no resource sharing conflicts or deadlocks may occur.

5 Conclusions

We presented the very first steps of an approach for generating timed automata from Java programs. The Java language is extended with timing annotations, which makes possible to check resource sharing conflicts and deadlocks in a generated system. We expect that replacing a lock-controlled resource access policy by a time-driven approach allows for better temporal and functional predictability, while allowing for greater flexibility than, say, synchronous languages.

The approach has been implemented in a prototype tool, and first tests seem to suggest that this approach works. However, the number of states increases rapidly with the growth of program size. That makes this approach difficult to apply for large systems, and more powerful abstractions will have to be developed.

Further work may be performed in two directions: Firstly, more Java source code statements and thread-specific methods should be translated to timed automata. Secondly, the adequacy of the translation algorithm is expected to be verified with a proof assistant, based on a formal semantics of Real-Time Java. The final aim of the future work is to support the constructions of Real-Time Java and have a formally verified translation procedure.

References

1. JSR 308. http://jcp.org/en/jsr/detail?id=308.
2. J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-27755-2.
3. T. Bøgholm, H. Kragh-Hansen, and P. Olsen. Model based schedulability analysis of real-time systems. Master's thesis, Aalborg University, 2008.
4. T. Bøgholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K. G. Larsen. Model-based schedulability analysis of safety critical hard real-time Java programs. In G. Bollella and C. D. Locke, editors, *JTRES*, volume 343 of *ACM International Conference Proceeding Series*, pages 106–114. ACM, 2008.
5. M. Cordovilla, F. Boniol, E. Noulard, and C. Pagetti. Multiprocessor schedulability analyser. In W. C. Chu, W. E. Wong, M. J. Palakal, and C.-C. Hung, editors, *SAC*, pages 735–741. ACM, 2011.
6. E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theor. Comput. Sci.*, 354(2):301–317, 2006.
7. P. Herber, M. Pockrandt, and S. Glesner. Transforming systemc transaction level models into uppaal timed automata. In *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, pages 161 – 170, July 2011.
8. A. P. Ravn and M. Schoeberl. Cyclic executive for safety-critical java on chip-multiprocessors. In T. Kalibera and J. Vitek, editors, *JTRES*, ACM International Conference Proceeding Series, pages 63–69. ACM, 2010.