Abstraction and Verification of Properties of a Real-Time Java

Nadezhda Baklanova and Martin Strecker

IRIT (Institut de Recherche en Informatique de Toulouse)
Université de Toulouse[⋆]
118 route de Narbonne, F-31062 Toulouse CEDEX 9, France
nadezhda.baklanova@irit.fr, martin.strecker@irit.fr

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, 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 effective 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 them, one may want to guarantee that the same resource is not accessed by different threads at the same moment of time. If one concentrates on this aspect, the behavior of a program may be naturally 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. It improves predictability of execution and guarantees there will be no delays due to locking conflicts. Based on an annotated Java program, a timed automaton is generated taking into account the

[⋆] Part of this research has been supported by the project *Verisync* (ANR-10-BLAN-0310) time required for execution of statements. Finally, we check the generated automaton for possible resource sharing conflicts using the Uppaal model checker in the generated automaton. The transformation sequence is shown in the diagram 1 below.

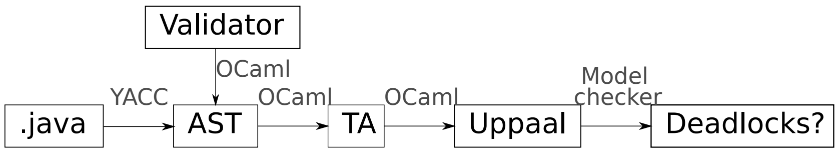


Fig. 1. Java program verification process

The overall aim is thus 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 chapter 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. A preliminary version of this chapter has appeared in [1].

1.1 Related Work

There are several tools for scheduling analysis of real-time tasks. Verification of scheduling strategies with timed automata is considered in [2]. 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 the Times tool.

Another approach is used in [3,4]. Here SCJ 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 timing model is based on WCET computation and predefined periods of tasks. 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. In order to manage this problem we suggest to define two semantics of Java execution. One treats locks in Java manner, i.e.

checks the resource monitor before acquiring the lock and does not allow double locking made by different threads. Another one just stores the number of times a resource lock has been acquired but does not check whether a resource has already been locked. These two semantics are equivalent for programs without resource sharing conflicts therefore the programs verified by our tool are correct during execution on JVM.

An approach in the opposite direction is described in [5]. The authors generate RTSJ code from a Uppaal model. Uppaal model of a system is supposed to be verified, and if the generation procedure is assumed to be correct, the output is a verified RTSJ program.

A translation from SystemC to UPPAAL is presented in [6]. 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 [7] a 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. The authors listed the limitations they had encountered: lack of memory and lack of UPPAAL integer range.

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. This model is not fine-grained, and the negative result may not be relevant in cases when developers try to minimize the length of critical sections.

A tool for automatic verifying the determinism of Java programs is described in [9]. In particular, parallel Java programs are checked for absence of race conditions. It allows not to use Java `synchronized` statements. The tool does not use any external checkers, the verification uses the internal abstract representation of a Java program.

A theoretical approach to managing resources in parallel programs is suggested in [10]. It is based on an enhanced version of rely-guarantee reasoning and allows to verify memory safety and of parallel programs.

2 Preliminaries

2.1 Real-Time Java

Specification of the Real-Time profile for Java was developed in the first half of the 2000s and aimed at making work with Java threads predictable and suitable for real-time applications. The specification addressed defining an explicit scheduler and scheduling strategies, advanced memory management and raw memory access, resource locking taking thread priorities into account, refined notion of time, several additions for threads, asynchronous event handling etc. [11]. The real-time specification introduced thread priorities, thread deadlines and explicit notion of scheduler which did not exist in usual Java.

In the middle of the 2000s the work on the specification of Safety-Critical Java was started. The main goal was to allow the SCJ applications to be highly reliable [12]. The SCJ specification defines more strictly the subset of possible programs and pays a lot of attention to the VM performance requirements [13]. Until now, there is no reference implementation of SCJ machine, however, there are projects such as the Open SCJ project [1] aiming at an open-source implementation of SCJ virtual machine.

2.2 Uppaal

Uppaal is a tool for modeling timed automata and verifying their properties. A timed automaton [14] is a Büchi automaton enhanced with clocks. States and edges may have Boolean constraints on clocks called invariants and guards respectively; edges may also have abstract "actions". There are two kinds of possible transitions: delay transition and action transition. During a delay transition an automaton stays in the same state, and time advances. During an action transition an automaton takes an edge and changes the state; some clocks may be reset to 0. A timed automaton is allowed to stay in a certain state as long as its invariant is true, and an automaton may take a particular edge, if its guard is true in the current moment of time.

In Uppaal "actions" are concrete arithmetic actions with variables or arrays whose values are preserved between TA states. Variables and array elements may also be used in edge constraints.

Properties to be verified in Uppaal are to be expressed in a subset of TCTL logic allowing a single path quantifier directly followed by a U operator [15].

3 Sample Usage

3.1 Input Program

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();
```

[1] http://www.ovmj.net/oscj/

```
t1=new Thread(r1,"t1");
t2=new Thread(r2,"t2");
```

Methods called on the thread start are the following:

```
private class Run1 implements Runnable{
  public void run(){
    int value,i;
    //@ 1 @//
    i=0;
    while(i<10){
      synchronized(res){
        //@ 2 @//
        value=Calendar.getInstance().get(Calendar.MILLISECOND);
        //@ 5 @//
        res.set(value);
      }
      try{

        Thread.sleep(10);
      }
      catch(InterruptedException e){
        System.out.println(e.getMessage());
      }
      //@ 2 @//
      i++;
    }
  }
}

private class Run2 implements Runnable{
  public void run(){
    int value,i;
    //@ 1 @//
    i=0;
    try{

      Thread.sleep(9);
    }
    catch(InterruptedException e){
      System.out.println(e.getMessage());
    }
    while(i<10){
      synchronized(res){
        //@ 4 @//
        value=res.get();
      }

      try{
        Thread.sleep(8);
      }
      catch(InterruptedException e){
        System.out.println(e.getMessage());
      }
      //@ 2 @//
      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 an 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.

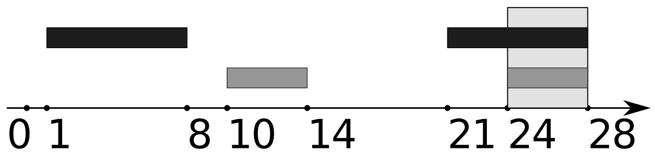


Fig. 2. Execution timeline. `t1` is dark gray, `t2` is gray. Conflict between 24 and 28 is shown in light gray.

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

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

3.2 Generated System

On the basis of the annotated program, our framework generates the following model which is passed to the model checker Uppaal for verification.

The generated timed automata are shown in Figure 3.

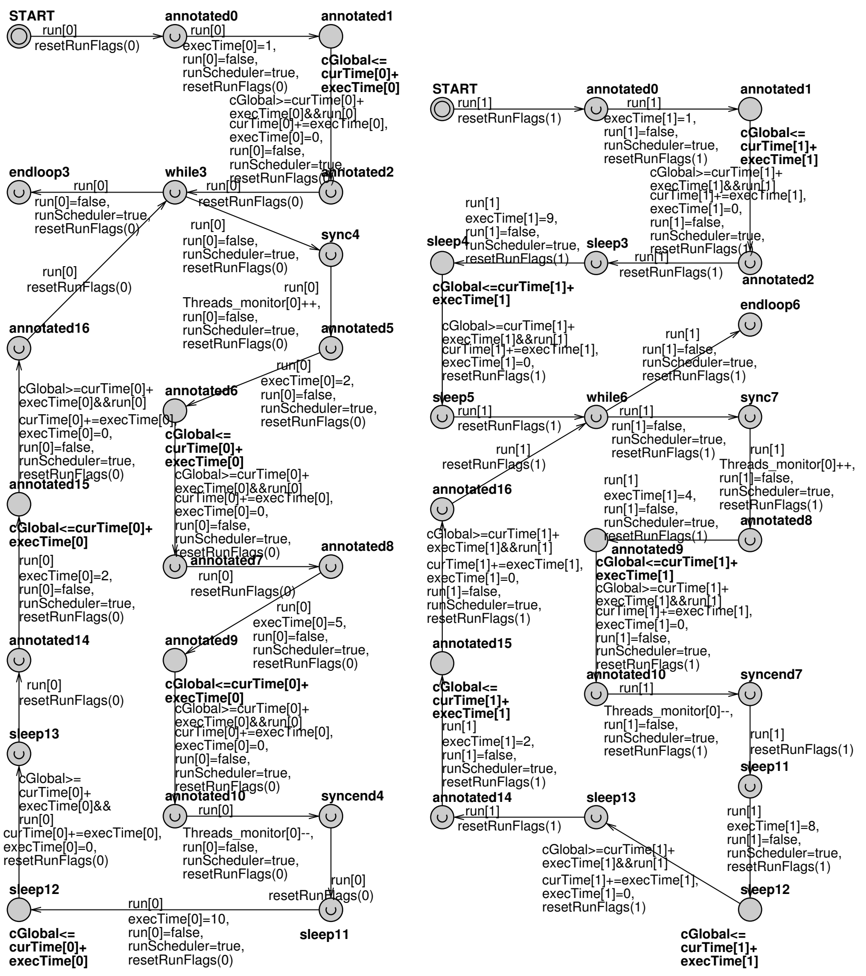


Fig. 3. Generated automata for threads $t1$ and $t2$.

The generated checking function is

```
int check_Threads(int m[1]){
  return forall(i: int[0,0]) m[i]<=1;
}
```

The generated formula for model checking is

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

When performing verification with Uppaal, this formula is evaluated to false, and the generated trace for counterexample stops in states $aut\_Run1\_t1.annotated9$ and $aut\_Run2\_t2.annotated8$ during the second loop iteration at the time moment 24.

4 Translation from Java to Abstract Syntax Tree

4.1 Source Language

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 annotations have syntax of Java comments, therefore, the annotated programs can be compiled to bytecode by usual Java compilers. We assume that a developer has information about execution time of particular statements. The time in the samples is in abstract time units but one can annotate a program with real values in microseconds based on computer architecture, compiler version, running software etc. Annotations may either contain an exact execution time or an interval in which the execution time lies.

Unfortunately, standard Java annotations cannot be added to arbitrary statements. Even the latest extension of Java annotations implemented in JDK 7 [12] 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. The parser produces an AST from Java code as a set of OCaml objects.

Java programs consist of a number of classes containing methods. Classes may have fields for storing object information. We concentrate on a subset of Java containing the most important syntax constructions. Target Java programs may contain the following statements:

```
type  stmt =
    Skip
      (* empty statement *)
  | Expr of expr
      (* expression 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){...} *)
  | Return of expr
      (* value return: return a; *)
  | AnnotStmt of annot * stmt
      (* annotated statement: //@ 4 @// a=3; *)
  | SyncStmt of expr * stmt
      (* synchronized statement: synchronized(a){...}
         *)
```

This is the representation of Java statements in AST written as OCaml type. Internally, statements and expressions have the same type, however, there is a difference on the semantics level. Expressions are supposed not to produce side effects whereas statements can induce state changes.

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`.

The analysed programs must have a proper structure which would guarantee the correctness of the generated model. 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. It is ensured in the parsing step.

We have developed a number of checking functions for testing the program correspondence to the structure requirements. These functions work after parsing on the semantics checking step. They return a Boolean value showing whether a check was successful. The checking functions traverse a method body recursively performing checks of the interesting cases. Currently, there are the following checks:

– `checkAliases`. The objects which can be accessed by different threads should not have more than one name, i.e. the correct program should not introduce aliases for them. If there are aliases in a program, there is no easy way of determining whether two variables point to the same object. This leads to inability to know which objects are locked at a particular moment of time.
– `checkAnnotCoverage`. The whole AST except the main method must be annotated with timing information. Each leaf or one of its parents must have an annotation in order to avoid undetermined execution duration. Method calls currently are not translated. However, since method calls are always leafs in the AST, we can use timing information instead of looking inside the method structure. The only exception must be a call of the `Thread.sleep` method since it takes time for execution but does not load CPU. Currently, we do not support wait/notify statements.
– `checkSyncArgument`. Argument of the `synchronized` statement is assumed to be an explicit object name, not an expression.
– `checkNestedSyncs`. Reentrant locks are not allowed, i.e. when the same thread acquires lock of the same object several times.
– `checkAnnotSync`. Annotated statements cannot contain synchronized statements since the automata generator treats annotated statements as atomic entities. If an annotated statement contains synchronized block, and during runtime there is a conflict between threads for the locked resource, our model cannot catch this conflict.
– `checkMainMethodPosition`. Program `main` method must be in the first class in order to make the generator simpler.

– `checkThreeadConstructors`. 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(Runnable target, String name),
  ```

  so the name of the object containing the thread logic is explicitly specified. `Runnable` object should implement `Runnable` interface or extend `Thread` class and override `run` method.

The `checkAliases` function first builds a list `initObjects` of the objects which can be accessed by different threads and then searches for assignments to these objects other than initializations. If an object is assigned a `new ...` expression, check for this assignment succeeds; if the expression which is assigned contains anything else except a constructor call, check for this assignment fails.

```
let rec checkAliasesInExpr initObjects=function
  |Assign(CallObject(o),e)->
    if mem o sharedFields then
      (match e with
        |CallMethod(New c,f,ps)->true
        |Null->true
        |_->false)
    else true
```

For the other cases the function looks inside statement bodies. Below there are semi-formal rules for these cases.

$$\frac{check\ e1 \quad check\ e2}{check\ (Seq\ e1\ e2)} \qquad \frac{check\ e1 \quad check\ e2}{check\ (Cond\ c\ e1\ e2)} \qquad \frac{check\ e}{check\ (While\ c\ e)}$$

$$\frac{check\ e}{check\ (Annot\ a\ e)} \qquad \frac{check\ e}{check\ (Sync\ obj\ e)}$$

The rest of checking functions are evident, and we do not show details of their implementation here.

5 Translation from Abstract Syntax Tree to Timed Automaton

5.1 Model of Java Program Execution

Multithreaded Java programs have a scheduler which selects a thread to be executed in the next moment of time. It non-deterministically selects a thread from those eligible for execution, and it can suddenly stop thread's execution and start executing another thread. Usual Java schedulers do not support thread priorities or task deadlines.

We model a Java scheduler as a separate automaton with three states:

– *waitScheduling*, where a scheduler waits for some time before starting the next scheduling cycle,
– *updateStatus*, where eligibility status of all threads is updated,
– *runThread*, where the scheduler gives control to any eligible thread.

If no thread can be scheduled, the scheduler returns to *waitScheduling* and waits for some time. Then it tries to schedule some thread again. When scheduled, a thread executes an atomic action and returns control back to the scheduler. The scheduler returns to the state *updateStatus* and updates thread eligibility flags.

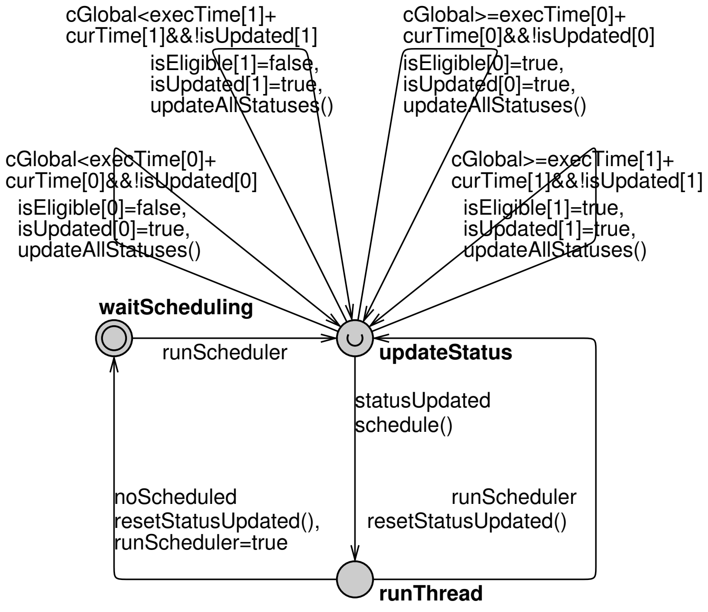


Fig. 4. Model of the scheduler for two threads.

Here, *execTime*[*i*] is an array containing the execution time of the next instruction for all threads taken from annotation values. *cGlobal* is a global clock. *runScheduler* is a Boolean flag indicating that it is scheduler's turn to proceed, *statusUpdated* shows that statuses of all threads were updated, *isScheduled*[*i*] is an array indicating whether a particular thread is scheduled for execution. *run*[*i*] is a global Boolean array showing that a thread with number *i* may proceed.

5.2 Semantics of Annotated Statements

Suppose we have an annotated statement

```
//@ 5 @//
a = b - 4;
```

which claims that the statement `a = b - 4;` takes 5 time units for execution. This time is considered as exact execution time, i.e. the exact amount of time when the thread executing this statement loads CPU. We consider the time when an annotation expires as a hard deadline. If a program does misses this deadline, the situation is critically incorrect, and the program cannot be verified because of incorrect annotations.

Formally, a small-step semantics of annotated statements may be written as following:

$$\frac{G \vdash (e, s) \xrightarrow{\delta} (e', s') \quad t - \delta \geqslant 0}{G \vdash (\mathit{Annot}\ t\ e;\ s) \xrightarrow{\delta} (\mathit{Annot}\ (t - \delta)\ e', s')}$$

$$\frac{t \geqslant 0}{G \vdash (\mathit{Annot}\ t\ (\mathit{Val}\ v), s) \xrightarrow{t} (\mathit{Val}\ v, s)}$$

Here G is a generated system, and the relation $G \vdash (e, s) \xrightarrow{t} (e', s')$ means there is a possible reduction of an expression e to an expression e' which takes time t and changes state from s to s'.

These rules should be read like

- if an expression e in the body of an annotated statement can be reduced to an expression e', and the system state is changed from s to s', and the reduction takes δ time units, and the deadline specified in the annotated statement is not missed, then we may reduce the initial annotated statement to the new one with the new body, e', being in the state s', and the new deadline, $(t-\delta)$.
- or, if an expression in the body of annotated statement has already been reduced up to the end, i.e. to a single value $\mathit{Val}\ v$, and the deadline specified in the annotation is not missed, we may reduce the annotated statement to the value of its body staying in the same state, and this reduction would take the rest of time specified in the annotation.

We assume that each thread is executed on its own processor i.e. the execution is purely parallel. Eligible threads do not wait until other threads free the processor. As soon as a thread becomes eligible for execution it starts executing.

5.3 Automata Generation

The generator translates each thread of a program to a separate automaton. At the beginning it creates a set of OCaml objects representing a timed automaton, and after that the timed automaton is printed in the format recognizable by Uppaal, which performs model checking.

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. The final state is required because a timed automaton is generated recursively, and it is necessary to determine where the previously generated parts 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 increase the expressiveness of the traditional TA model. Final states of the generated timed automata are always urgent, that means, the automata are not allowed to rest in these states for any time. One may find definitions related to timed automata in [15].

Since 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 guaranteed by the automata generator.

All statements except `Thread.sleep` and 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 contain `synchronized` blocks. Otherwise, a possible situation is when a thread tries to access an object field which is locked by another thread. In this situation JVM keeps the thread waiting until the lock is released, however, our translation does not notice this delay and produces an incorrect automaton.

The generated system has one global clock, `cGlobal` and several auxiliary variables. There are Boolean flags for each automaton, `run` and `runScheduler`, which are set to true if this automaton may advance in the current time moment. An integer array `curTime` represents the time when an automaton entered the state corresponding to an annotation statement. Another integer array, `execTime`, stores the duration of the currently executed statement. Finally, an integer array `<class_name>_monitor` stores the number of object locks for each shared object.

Annotations in Java programs contain relative time but timed automata use global time, therefore we need to keep track of how much time has passed since a program has been started. The only statements allowing time to increase are annotated statements and calls of `Thread.sleep`. Values for `execTime` are taken from timing annotations or method argument. 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 real execution time, `execTime`. `curTime` values may be different in different automata but the global time is always equal to `curTime` when its corresponding automaton is executing.

Basic items for building timed automata are statements: each statement is translated into a part of timed automaton.

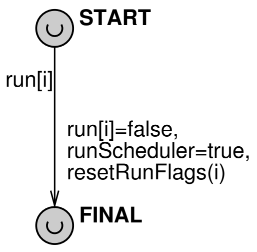
Translation from AST to timed automata skips field and variable declarations because they do not change the state of a program. At the same time, all the objects declared in the main program class get a monitor.

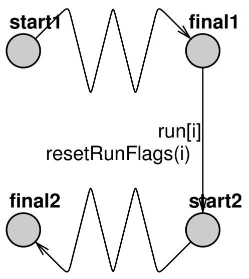
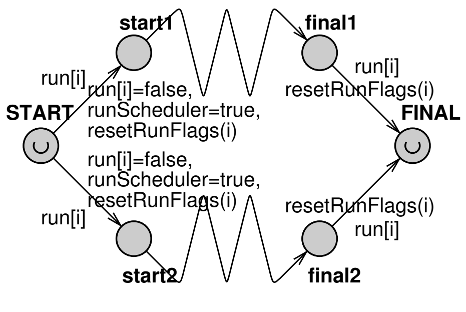
Boolean conditions inside `while` and `if` statements are not translated. It is assumed that any of the two possible ways can be taken during runtime.

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

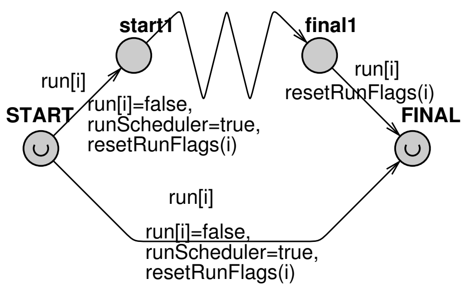
The rules for mapping other AST statements to the parts of a timed automaton are listed in the table below.

Table 1. Translation rules.

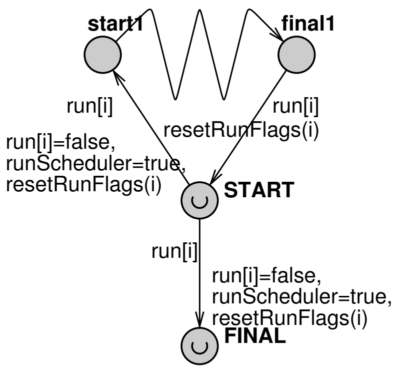
`Assign(v,e)`: add two urgent states: $ASSIGNMENT1$ and $ASSIGNMENT2$, which are start and final, and a transition with a guard and an update between them. The guard checks `run[i]`, i.e. whether this automaton is allowed to proceed in the current moment of time. The update sets `run[i]` to false, `runScheduler` to true and other `run`[`j` $\neq$ `i`] to false.	 START
	run[i]
	run[i]=false,
	runScheduler=true,
	resetRunFlags(i)
	FINAL

`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. The edge has a guard checking `run[i]` and an update setting `run`[`j` $\neq$ `i`] to false.	 start1 final1 run[i] resetRunFlags(i) final2 start2
`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 $FINAL$. The edges from $START$ to $start1$, $start2$ have guards checking `run[i]` and updates resetting `run[i]` to false,	 start1 final1 run[i] run[i]=false, runScheduler=true, resetRunFlags(i) run[i] resetRunFlags(i) START FINAL run[i]=false, runScheduler=true, resetRunFlags(i) run[i] resetRunFlags(i) run[i] start2 final2

`runScheduler` to true and `run`[`j` $\neq$ `i`] to false. The edges from $final1$ and $final2$ have only guards checking `run[i]` and updates setting `run`[`j` $\neq$ `i`] to false. If one of the branches is absent, e.g. there is no else branch, a transition from $START$ to $FINAL$ is added. This transition has a guard checking `run[i]` and updates resetting `run[i]` to false, `runScheduler` to true and `run`[`j` $\neq$ `i`] to false.

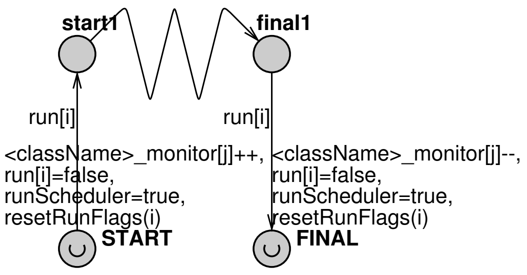


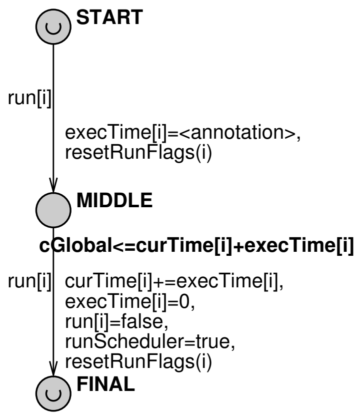
`Loop(e,c1)`: suppose $a1$ is the automaton for $c1$, add two urgent states $START$ and $FINAL$, which are the start and final states of the new automaton, and edges from $START$ to $start1$, from $final1$ to $FINAL$ and from $START$ to $FINAL$. The edges from $START$ to $start1$ and from $final1$ to $START$ have guards checking `run[i]` and updates resetting `run[i]` to false, `runScheduler` to true and `run`[`j` $\neq$ `i`] to false. The edge from $START$ to $FINAL$ has only a guard checking `run[i]` and an update setting `run`[`j` $\neq$ `i`] to false.



`Expr(e)`: not translated except methods for thread management. For translation of `Thread.sleep` see the `Annot` item.

`Sync(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. Also, both edges have guards checking `run[i]` and updates resetting `run[i]` to false, `runScheduler` to true and `run`[`j` $\neq$ `i`] to false.



`Annot(a,c1)`, `Thread.sleep(a)`: add three states: *START*, *MIDDLE* and *FINAL*, and edges from *START* to *MIDDLE* and from *MIDDLE* to *FINAL*. *START* and *FINAL* are the start and final states of the new automaton, both are urgent. The edge from *START* to *MIDDLE* has a guard checking `run[i]`, an update setting `execTime` to the execution time indicated in the annotation and another update setting `run`[`j` $\neq$ `i`] to false. The edge from *MIDDLE* to *FINAL* has a guard checking `run[i]` and several updates. First, there is an update increasing `curTime` to `execTime`. Second, there is an update resetting `execTime` to zero. Third, there are updates setting `run[i]` to false, `runScheduler` to true and `run`[`j` $\neq$ `i`] to false.	 START
	run[i]
	execTime[i]=<annotation>,
	resetRunFlags(i)
	MIDDLE
	cGlobal<=curTime[i]+execTime[i]
	run[i]
	curTime[i]+=execTime[i],
	execTime[i]=0,
	run[i]=false,
	runScheduler=true,
	resetRunFlags(i)
	FINAL

5.4 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 TCTL formulas [16]. The basics of TCTL and its applications are described in [17]. Together with automata code our generator produces a file with properties to check. The negative property of the generated system is whether there are two threads accessing the same resource at the same time. We check the positive variant of it. There is a function `check_<class_name>` checking that all the elements of the monitors array are less or equal to 1. If this property holds for all states of all possible paths, the system does not have resource sharing conflicts. In Uppaal syntax the property looks like

$$A[]check\_\langle className\rangle(\langle className\rangle\_monitor).$$

If the property does not hold, Uppaal produces a trace violating the check.

6 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 allow to assume 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. In order to avoid state explosion, large parts of code should be included into annotated statements. It allows to abstract from particular statements and generate an automaton with quite a few states.

6.1 Interval Annotations

We have considered an approach to make our analysis more precise. Currently, an annotation is exact time required for execution of a statement. Certainly, it is not a realistic model as one can never know before execution itself how much time it will take. The real execution time depends on the contents of the processor cache, also on the compiler optimisation level and many other things. For this reason we considered a simple model where timing annotation is an interval, and execution time must lie within it. However, this naive model cannot represent the execution flow correctly; consider an example when a loop body has an interval annotation.

```
while ( true ) {
//@ 3 - 5 @//
... //some actions
}
```

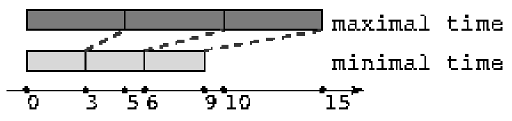


Fig. 5. Execution timeline for intervals.

On the execution timeline to the right one can see that after the third round of loop execution the interval of non-determinism became longer than the loop execution time itself. The better model is still a question for further investigation; one of the possible examples is the model discussed in [18]. The approach suggested by the authors is to keep non-determinism for separate steps of execution, however, sets of instructions have a hard deadline. This may help to solve the mentioned unlimited growth between best and worst execution time.

6.2 Future Work

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. Baklanova, N., Strecker, M., Féraud, L.: Resource sharing conflicts checking in multithreaded Java programs. In: Journées FAC'12. (April 2012)
2. Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Schedulability analysis of fixed-priority systems using timed automata. Theor. Comput. Sci. **354**(2) (2006) 301–317
3. Bøgholm, T., Kragh-Hansen, H., Olsen, P.: Model based schedulability analysis of real-time systems. Master's thesis, Aalborg University (2008)
4. Bøgholm, T., Kragh-Hansen, H., Olsen, P., Thomsen, B., Larsen, K.G.: Model-based schedulability analysis of safety critical hard real-time Java programs. In Bollella, G., Locke, C.D., eds.: JTRES. Volume 343 of ACM International Conference Proceeding Series., ACM (2008) 106–114
5. Hakimipour, N., Strooper, P., Wellings, A.: A model-based development approach for the verification of real-time java code. Concurrency and Computation: Practice and Experience **23**(13) (2011) 1583–1606
6. Herber, P., Pockrandt, M., Glesner, S.: Transforming systemc transaction level models into uppaal timed automata. In: Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on. (July 2011) 161 –170
7. Cordovilla, M., Boniol, F., Noulard, E., Pagetti, C.: Multiprocessor schedulability analyser. In Chu, W.C., Wong, W.E., Palakal, M.J., Hung, C.C., eds.: SAC, ACM (2011) 735–741
8. Ravn, A.P., Schoeberl, M.: Cyclic executive for safety-critical java on chip-multiprocessors. In Kalibera, T., Vitek, J., eds.: JTRES. ACM International Conference Proceeding Series, ACM (2010) 63–69
9. Vechev, M.T., Yahav, E., Raman, R., Sarkar, V.: Automatic verification of determinism for structured parallel programs. In Cousot, R., Martel, M., eds.: SAS. Volume 6337 of Lecture Notes in Computer Science., Springer (2010) 455–471
10. Tofan, B., Schellhorn, G., Bäumler, S., Reif, W.: Embedding rely-guarantee reasoning in temporal logic. Technical Report 2010-07, Informatik (2010)
11. The Real-Time for Java Expert Group: The Real-Time Specification for Java. (January 2006)
12. The Open Group JSR: JSR-302 Safety Critical Java Technology Specification. `http://jcp.org/en/jsr/detail?id=308` (2010)
13. Henties, T., Hunt, J.J., Locke, D., Nilsen, K., Schoeberl, M., Vitek, J.: Java for safety-critical applications. 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009) (March 2009)
14. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126** (1994) 183–235
15. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In Desel, J., Reisig, W., Rozenberg, G., eds.: Lectures on Concurrency and Petri Nets. Volume 3098 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2004) 87–124 10.1007/978-3-540-27755-2.
16. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e. (June 1990) 414 –425
17. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
18. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: a time-triggered language for embedded programming. Proceedings of the IEEE **91**(1) (2003) 84–99