A Formal Model of Resource Sharing Conflicts in Multithreaded Java *

Nadezhda Baklanova and Martin Strecker

Institut de Recherche en Informatique de Toulouse (IRIT), Université de Toulouse

{nadezhda.baklanova,martin.strecker}@irit.fr

Abstract. We present a tool for analysing resource sharing conflicts in multithreaded Java programs. We consider two models of execution: purely parallel one and sequential execution on a single processor. A Java program is translated into a system of timed automata which is verified by the model checker UPPAAL. We also present our work in progress on formalisation of Real-Time Java semantics and the semantics of timed automata.

Keywords. resource sharing, Java, timed automata, model checking

Key terms. Model, Development, ConcurrentComputation, FormalMethod, QualityAssuranceProcess

1 Introduction

Along with increasing usage of multithreaded programming, a strong need of sound algorithms arises. The problem is even more important in programming of embedded and real-time systems where liveness conditions are extremely important. To certify that no thread would starve or would be deadlocked, lock-free and wait-free algorithms have been developed. Lock-free algorithms do not use critical sections or locking and allow to avoid thread waiting for getting access to a mutual exclusion object. Nevertheless, only one thread is guaranteed to make progress. Wait-free algorithms prevent starvation by guaranteeing a stronger property: all threads are guaranteed to make progress, eventually. Such algorithms for linked lists, described for example in [4,11], are very complex, difficult to implement and, consequently, hard to verify.

What is worse, these algorithms seem to be incompatible with hard real-time requirements: the progress guarantees are not bounded in time. Thus, a lock-free insertion of an element into a linked list by a thread may need several (possibly infinitely many) retries because the thread can be disturbed by concurrent threads. Under these conditions, it is not possible to predict how much time is needed before the thread succeeds.

* Part of this research has been supported by the project *Verisync* (ANR-10-BLAN-0310)

Critical sections are used in many applications in order to ensure concurrent access to objects although if the scheduling order is wisely planned, locks are not necessary since threads access objects at different moments of time. This is the motivation for our work: we develop a tool for checking resource sharing conflicts in concurrent Java programs based on the statement execution time. This gives a "time-triggered" [6] flavor to our approach of concurrent system design: resource access conflicts are resolved by temporal coordination at system assembly time, rather than during runtime via locking or via retries (as in wait-free algorithms).

We assume that a program is annotated with WCET information known from external sources. The checker translates a Java program into a timed automaton which is then model checked by a tool for timed automata (concretely, UPPAAL).

In this paper, after an informal introduction (Section 2), we present a formal semantics of the components of the translation, namely Timed Automata (Section 3) and a multi-threaded, timed version of Java (Section 4). Then we describe the mechanism of the concrete translator written in OCaml (Section 5) and give some preliminary correctness arguments (Section 6) – the formal verification still remains to be done. The conclusions (Section 7) discuss some restrictions of our current approach and possibilities to lift them.

Status of the present document: We give a glimpse at several aspects of our formalisation, which is far from being coherent. Therefore, this paper is rather a basis for discussion than a finished publication.

2 Informal Overview

To show the main idea, we present an example of a concurrent Java program. It is a primitive producer-consumer buffer with one producer and one consumer where both producer and consumer are invoked periodically. The program is annotated with information about statement execution time in `//@ ... @//` comments.

```
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);
      }
      Thread.sleep(10);
      //@ 2 @//
      i++;
```

```
      }
    }
  }

  private class Run2 implements Runnable{
    public void run(){
      int value,i;
      //@ 1 @//
      i=0;
      Thread.sleep(9);
      while(i<10){
        synchronized(res){
          //@ 4 @//
          value=res.get();
        }
        Thread.sleep(8);
        //@ 1 @//
        i++;
      }
    }
  }
```

One of the possible executions is shown in Figure 1.

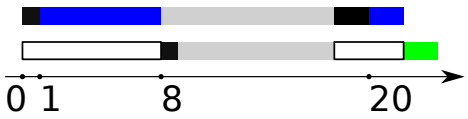


Fig. 1: Possible execution flow. Black areas represent execution without locks, blue and green areas - execution within a critical section, grey areas - sleeping, white areas - waiting for processor time.

After having translated this program to a system of timed automata we run the UPPAAL model checker to determine possible resource sharing conflicts. The checked formula is

$$A[]\forall(i : int[0, objNumber - 1])\forall(j : int[0, autNumber - 1])waitSet[i][j] < 1, \quad (1)$$

where *waitSet* is an array of boolean flags indicating whether a thread waits for a lock of a particular object. If all array members in all moments of time are false, no thread waits for a lock therefore no resource sharing conflicts are possible.

3 Timed Automata Model

Timed automata are a common tool for verifying concurrent systems; the underlying theory is described in [1]. We formalize the basic semantics of an extension of timed automata used in the UPPAAL model checker. The formalized syntax and semantics are adapted from [3].

An automaton edge is composed of a starting node, a condition under which the edge may be taken (guard), an action, clocks to reset and a final node.

type-synonym $('n, 'a)$ *edge* $= 'n \times$ *cconstr* $\times 'a$ *list* $\times$ *id set* $\times 'n$

An invariant is a condition on a node which must be satisfied when an automaton is in this node.

type-synonym $'n$ *inv* $= 'n \Rightarrow$ *cconstr*

An automaton consists of a set of nodes, a starting node, a set of edges and an invariant function.

type-synonym $('n, 'a)$ *ta* $= 'n$ *set* $\times 'n \times ('n, 'a)$ *edge set* $\times 'n$ *inv*

We use the model checker UPPAAL which proposes an extension of classical timed automata with variables. A full state comprises a node and a valuation function. There are two types of variables: integer (*aval*) and boolean (*bval*), and clock variables which have a special semantic status in timed automata. The available transitions can be defined knowing given this state.

record *valuation*=
  aval:: *id*⇒*nat*
  bval:: *id*⇒*bool*
  cval:: *id*⇒*time*

type-synonym $'n$ *state* $= 'n \times$ *valuation*

A timed automaton can perform two types of transitions: timed delay and edge taking. If an automaton takes an edge, variables may be updated or clocks may be reset to 0. The `Transition` constructor takes a list of variable and clock updates as an argument.

datatype $'a$ *ta-action* $=$ *Timestep time* | *Transition* $'a$ *list*

Accordingly, the timed automata semantics has two rules: delay and transition. If an automaton is delayed, it stays in the same node, and values of all clocks are increased to the value *d*. The invariant of the current node must be satisfied.

If an automaton takes a transition, the node is changed, and clock values remain unchanged unless they are reset to 0. The invariants of both starting and final nodes must be satisfied as well as the guard of the taken transition. If the action of the transition involves variable updates, the valuation function is updated as well. This is done by the function application *eval-stmts varv a*.

$$\frac{varv' = varv(\!|cval := add\ (cval\ varv)\ d|\!) \quad varv \models invs\ l \quad varv' \models invs\ l \quad l \in nodes}{(nodes,\ init,\ edges,\ invs) \vdash (l,\ varv)\ -Timestep\ d\rightarrow\ (l,\ varv')}$$

$$\frac{\begin{array}{c}(l,\ g,\ a,\ r,\ l') \in edges \\ varv \models g \quad varv' = eval\text{-}stmts\ varv\ a(\!|cval := reset\ (cval\ varv)\ r|\!) \\ varv' \models invs\ l' \quad l \in nodes \quad l' \in nodes\end{array}}{(nodes,\ init,\ edges,\ invs) \vdash (l,\ varv)\ -Transition\ a\rightarrow\ (l',\ varv')}$$

A network of timed automata is a product automaton with exceptions in case of handshaking actions [2]. Handshaking allows to synchronize two automata so that both take an edge simultaneously.

$$\frac{\begin{array}{c}(s_1,\ g_1,\ a,\ cs_1,\ s_1{}') \in edges_1 \quad (s_2,\ g_2,\ a,\ cs_2,\ s_2{}') \in edges_2 \\ a \in getEdgeActions\ edges_1 \cap getEdgeActions\ edges_2\end{array}}{((s_1,\ s_2),\ g_1\ \lceil\wedge\rceil\ g_2,\ a,\ cs_1 \cup cs_2,\ s_1{}',\ s_2{}') \in edges\text{-}shaking\ edges_1\ edges_2}$$

$$\frac{\begin{array}{cc}(s_1,\ g,\ a,\ cs,\ s_1{}') \in edges_1 & a \in getEdgeActions\ edges_1 \\ a \notin getEdgeActions\ edges_2 & s_2 \in getEdgeNodes\ edges_2\end{array}}{((s_1,\ s_2),\ g,\ a,\ cs,\ s_1{}',\ s_2) \in edges\text{-}shaking\ edges_1\ edges_2}$$

$$\frac{\begin{array}{cc}(s_2,\ g,\ a,\ cs,\ s_2{}') \in edges_2 & a \in getEdgeActions\ edges_2 \\ a \notin getEdgeActions\ edges_1 & s_1 \in getEdgeNodes\ edges_1\end{array}}{((s_1,\ s_2),\ g,\ a,\ cs,\ s_1,\ s_2{}') \in edges\text{-}shaking\ edges_1\ edges_2}$$

4 Java Model

The look of the Java semantics has been inspired by the Jinja project [5] and its multithreaded extension JinjaThreads [7]. The Java execution flow is modeled by three transition relations: evaluation, scheduler and platform. The evaluation semantics is the semantics of a single thread, the scheduler semantics is responsible for thread scheduling, and the platform semantics formalizes the notion of time advancement. Taking into account the passage of time is the essential increment wrt. the above-mentioned semantics.

Following Jinja, we do not distinguish expressions and statements; their datatype is the following:

datatype *expr*
= *Val val*
| *Var vname*
| *VarAssign vname expr*
| *Cond expr expr expr*
| *While expr expr*

| *Annot annot expr*
| *Sync expr expr*
| *Sleep expr*

... and others.

The system state is large and complex; it stores local information of all threads such as local variable values and expression to be evaluated, shared objects, time, actions to be carried out by the platform, locks and wait sets. Given this state and scheduler logic, the further execution order is deterministic.

record *full-state* =
threads :: *id* ⇒ *schedulable option* — threads pool
th-info :: *thread-id* ⇒ *thread-state option* — expression to be evaluated and state
sc-info :: *schedule-info* — locks and thread statuses
pl-info :: *platform-info* — global time
gl-info :: *heap* — heap state
ws-info :: *waitSet* — wait sets
running-th :: *thread-id* — currently running thread
pending-act :: *action* — action to be carried out by platform

Evaluation semantics depends solely on local and heap variables therefore we use the reduced variant of full state for thread evaluation step.

record *eval-state* =
ev-st-heap :: *heap*
ev-st-local :: *locals*

When a thread expression is reduced, the duration of the performed action is not taken into account on the evaluation step, so the action type is passed further to the platform step where time advances according to the action. For now we assume that any action of a particular type takes a fixed amount of time for execution.

The evaluation rules take an expression and a local state and translate them to the new pair of expression and state and also emit an action for the platform step. Here are some examples of evaluation rules.

$$\frac{fs \vdash (e,\ s)\ -act\rightarrow\ (e',\ s')}{fs \vdash (VarAssign\ vr\ e,\ s)\ -act\rightarrow\ (VarAssign\ vr\ e',\ s')}$$

$$fs \vdash (VarAssign\ vr\ (Val\ vl),\ s)\ -EvalAct\ (exec\text{-}time\ VarAssignAct)\rightarrow\ (Val\ Unit,\ s(\!|ev\text{-}st\text{-}local := ev\text{-}st\text{-}local\ s(vr \mapsto vl)|\!))$$

Several rules use information about locks and wait sets that is not included in the local state therefore they pull it from the full state.

$$\frac{\neg\ locked\ a\ fs}{fs \vdash (Sync\ (Val\ (Addr\ a))\ e,\ s)\ -lock\text{-}action\ a\ fs\ 0\rightarrow\ (Sync\ (Val\ (Addr\ a))\ e,\ s)}$$

$$\frac{\mathit{locked}\ a\ \mathit{fs} \qquad \mathit{fst}\ (\mathit{the}\ (\mathit{sc\text{-}lk\text{-}status}\ (\mathit{sc\text{-}info}\ \mathit{fs})\ a)) \neq \mathit{runnint\text{-}th}\ \mathit{fs}}{\mathit{fs} \vdash (\mathit{Sync}\ (\mathit{Val}\ (\mathit{Addr}\ a))\ e,\ s) -\mathit{lock\text{-}action}\ a\ \mathit{fs}\ 0\rightarrow (\mathit{Throw}\ [\mathit{ResourceSharingConflict}],\ s)}$$

$$\frac{\mathit{locked}\ a\ \mathit{fs}}{\mathit{fs} \vdash (\mathit{Sync}\ (\mathit{Val}\ (\mathit{Addr}\ a))\ (\mathit{Val}\ v),\ s) -\mathit{unlock\text{-}action}\ a\ \mathit{fs}\ 0\rightarrow (\mathit{Val}\ \mathit{Unit},\ s)}$$

5 Abstracting Java to Timed Automata

We consider two models of program execution. The first one is purely parallel, i.e. each thread is assumed to execute on its own processor so that no thread waits for CPU time. Another model is the sequential one when a program executes on a single processor. The parallel model is easier, however, it does not seem to be realistic. The sequential model represents the realistic situation for real-time Java applications since the RTSJ (Real-Time Specification of Java [8]) specifies the behavior for monoprocessor systems only.

The translated Java programs must be annotated with timing information about execution time of the following statement. The translation uses timing annotations to produce timed automata which model the program. The obtained system is model checked for possible resource sharing conflicts.

5.1 General principles

We suppose that the translated program has a fixed number of threads and shared fields, all of them defined statically. The initialization code for threads and shared fields must be contained in the `main` method. The classes implementing `Runnable` interface must be nested classes in the class containing the `main` method. The required program structure is shown in the figure 2.

Each thread created in the program is translated into one automaton, and one more additional automaton modeling the Java scheduler is added to the generated system.

Java statements are translated into building blocks for condition statement, loop etc. which are assembled to obtain the final automaton. Annotated statement is translated into its own block. Method calls and wait/notify statements are not translated for now.

The timed automata system contains an array of object monitors representing acquired locks on shared objects. When a thread acquires a lock of an object, the monitor corresponding to this object is incremented, and when the lock is released, the monitor is decremented.

There is a number of checks which are performed before on the program source code which guarantee correctness of the generated model. One of the most critical is the requirement that the whole parse tree must be annotated, i.e. for any leaf of the AST there is a timing annotation somewhere above this leaf. With this requirement the behavior of the generated system can be determined in each moment of time.

```
public class Main{
  Res1 field1; //shared fields declaration
  public static void main(String[] args){
    Run1 r1; //declarations of Runnable object instances
    Thread t1,t2; //thread declarations
    r1=new Run1(); //Runnable objects initialization
    field1=new Res1(); //shared fields initialization
    t1=new Thread(null,r1,"t1"); //thread creation
    t1.start(); //thread start
  }
  private class Run1 implements Runnable{
    public void run(){ //thread logic implementation
      ...
    }
  }
}
private class Res1{ //resouce classes
  ...
}
```

Fig. 2: Required program structure

5.2 Parallel model

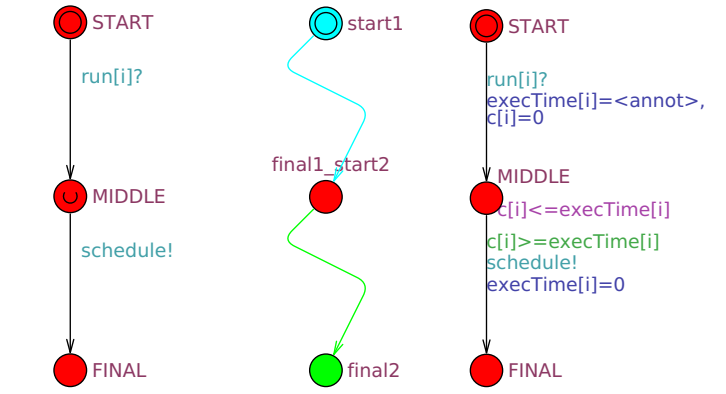
In the parallel model threads are supposed not to wait for processor time if they want to execute a statement. However, threads can wait for a lock if they need one which has been taken by another thread. Since this model is not very realistic we concentrate on the sequential model further.

5.3 Sequential model

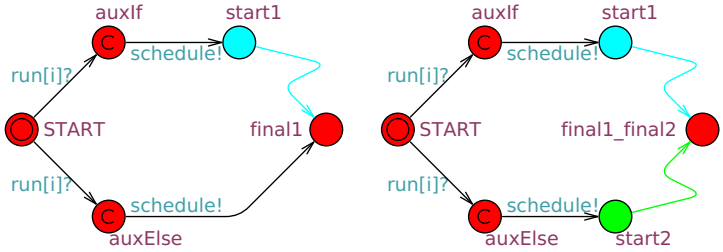
In the sequential model we assume that at every moment of time only one thread or scheduler can execute. Threads which do not execute in a particular moment of time wait for processor time. Also threads can wait for a lock; waiting does not consume CPU time.

Automata communicate with the scheduler through channels: if the scheduler has selected one thread, it sends a message to it so the thread starts executing. After finishing its execution, the thread sends a message to the scheduler, and the next scheduling cycle starts. The scheduler uses channels `run[i]` to call the i-th automaton, and the automata use the channel `scheduler` to give the control back to the scheduler.

There is an array of clocks `c[i]`, each of them corresponding to one thread automaton. These clocks are used to calculate time of annotated statements execution or sleeping time. There is also one clock `cGlobal` used for tracking global time.



(a) Assignment (b) Sequence (c) Annotation



(d) Condition I (e) Condition II

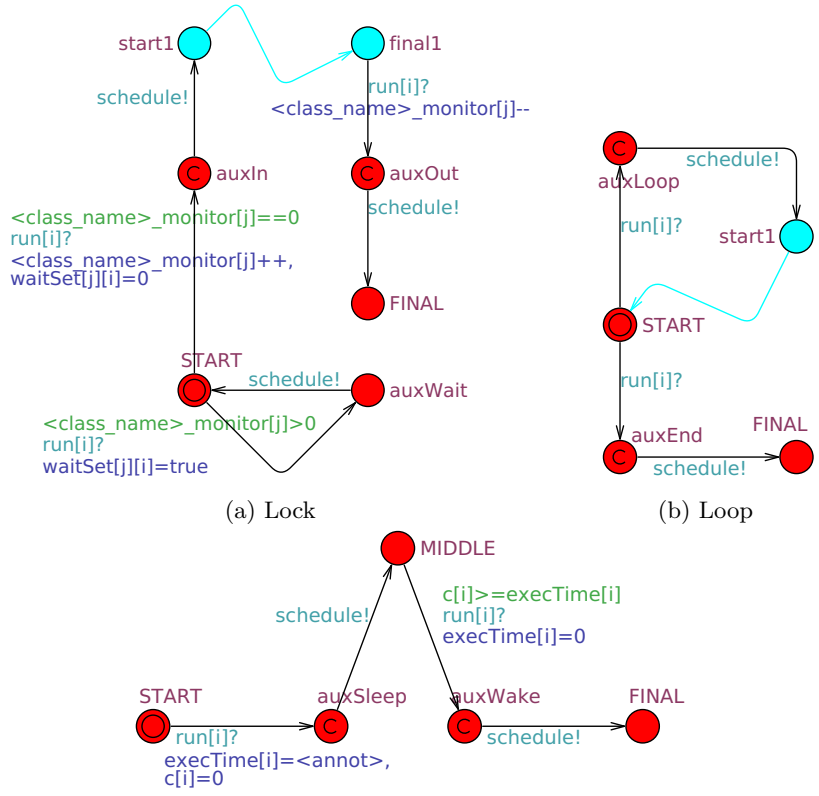
Fig. 3: Building blocks for automata. Elements added on the current step are red; blue and green elements have been generated in the previous step.

The building blocks and their translation are the following for the sequential model:

(a) Assignment (3a). Three new states and two transitions between them are added. The transition from `START` to `MIDDLE` listens to the channel `run[i]`, and the transition from `MIDDLE` to `FINAL` calls the channel `schedule`. The state `MIDDLE` is urgent since we assume that any statement except the annotated one takes time for execution.

(b) Sequence (3b). Having two automata with start and final states called `start1`, `start2` and `final1`, `final2` correspondingly, the states `final1` and `start2` are merged.

(c) Annotation (3c). Three new states and two transition between them are added. The transition from `START` to `MIDDLE` listens to the channel `run[i]`, sets the variable `execTime[i]` to the value of the current annotation and resets the clock `c[i]` to 0. The transition from `MIDDLE` to `FINAL` calls the channel `schedule` and resets the variable `execTime[i]` back to 0. The state `MIDDLE` has an invariant forbidding the automaton to stay in this state if the value of the clock `c[i]` bypasses `execTime[i]`. The transition from `MIDDLE`



(a) Lock (b) Loop

(c) Sleeping

Fig. 4: Building blocks for automata. Elements added on the current step are red; blue and green elements have been generated in the previous step.

to `FINAL` has a guard enabling this transition only if the value of `c[i]` is greater or equal to `execTime[i]`. The invariant and the guard ensure that the automaton would be in the `MIDDLE` state as long as the annotation claims.

(d) Condition (3d,3e). Three new states and four transitions are added. If both if and else branches are presented, the final states of automata representing the branch internals are merged. The states `auxIf` and `auxElse` are auxiliary states introduced to divide listening and calling transitions therefore they are made committed. Two transitions from `START` to `auxIf` and `auxElse` listen to the channel `run[i]`, and the transitions from `auxIf` to `start1` and from `auxElse` to `start2` (or to `final1` in case of absence of the else branch) call the channel `schedule`.

(e) Lock (4a). Two meaningful states and two auxiliary states are added. The transition from `START` listens to the channel `run[i]` and has a guard checking whether a lock for the object in the argument of the `synchronized` statement is not taken by other threads.

(f) Loop (4b). Two meaningful states and two auxiliary states are added. One transition from the `START` goes to the next loop iteration, another one exits the loop. Both transitions from `START` to `auxLoop` and `auxEnd` listen to the channel `run[i]`. The transitions from `auxLoop` to `start1` and from `auxEnd` to `FINAL` call the channel `schedule`. Both `auxLoop` and `auxEnd` are made committed. The final state of the automaton corresponding to the loop body is merged with the `START` state.

(g) Sleeping (4c). The automaton for sleep statement resembles the automaton for annotated statement with additional elements for returning control to the scheduler during sleeping. There are three meaningful and two auxiliary states with transitions connecting them into a chain. The auxiliary states, `auxSleep` and `auxWake`, are committed. The transition from `START` to `auxSleep` listens to the channel `run[i]`, sets the variable `execTime[i]` to the duration of sleeping period and resets the clock `c[i]` to 0. The transition from `auxSleep` to `MIDDLE` calls the channel `schedule` so that the scheduler can schedule other threads. The transition from `MIDDLE` to `auxWake` listens to the channel `run[i]` and has a guard enabling this transition only if `c[i]` is greater or equal to `execTime[i]`. The update on this channel resets the value of `execTime[i]` back to 0. Unlike the automaton for the annotated statement, there is no invariant in the `MIDDLE` state because a thread is not obliged to continue its execution right after it has woken up. It may wait for processor time before. The transition from `auxWake` to `FINAL` calls the `schedule` channel.

5.4 Scheduler

The Java scheduler maintains thread statuses and grants permission to execute to threads. The scheduler model has three states: `scheduling`, `runThread`, `wait`. The scheduler starts in the state `scheduling` which has transitions for updating thread eligibility statuses. When all thread statuses are updated, the scheduler moves to the state `runThread` calling the channel `run[i]` for some thread with index `i` which is eligible for execution. While the thread is executing, the scheduler stays in the state `runThread`. When the thread has finished its execution, it calls a channel `schedule`, and the scheduler returns back to the `scheduling` state, and the new scheduling cycle starts. If there was no thread eligible for execution, the scheduler goes to the `wait` state where it can stay for some time and repeat scheduling.

Each thread gets two transitions for status updates. One assumes that a deadline for an action performing by a thread has passed, another one assumes that the deadline has not been reached yet. In the first case the flag `isEligible[i]` is set to true, and the thread with index `i` can be scheduled for execution. Otherwise, `isEligible[i]` is set to false, and the thread with index `i` cannot be scheduled.

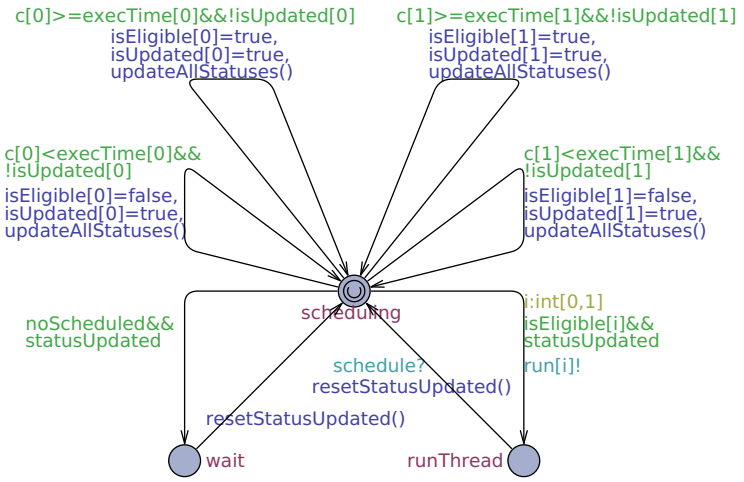


Fig. 5: Scheduler for a system with two threads.

6 Correctness of Abstraction

Java threads have a complex lifecycle shown in Figure 7. We adopt it to our model with limitations (see Figure 6). States in building blocks for automata can be attributed to a single class. For example, the start and end states of any building block belong to the *scheduling* state.

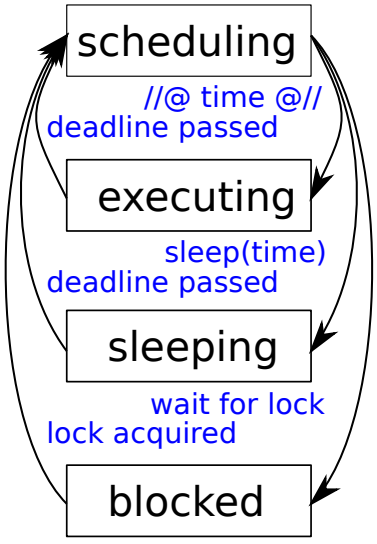


Fig. 6: Thread model used by the translation.

Each thread automaton starts in the *scheduling* state; after being scheduled it can go to one of the three states below. A thread is in the state *executing* if there is an assignment or an annotated statement to be executed. A thread goes to the *sleeping* state if it has met a sleep call; if a thread wants to acquire a lock but it cannot because another thread owns it, the active thread has to go to the *blocked* state and wait until another thread releases the lock.

The listed states are general, and automata may perform several internal steps while staying in the same generalized state.

Based on the state classification above, the simulation function can be easily built by mapping states of Java semantics to states of TA semantics

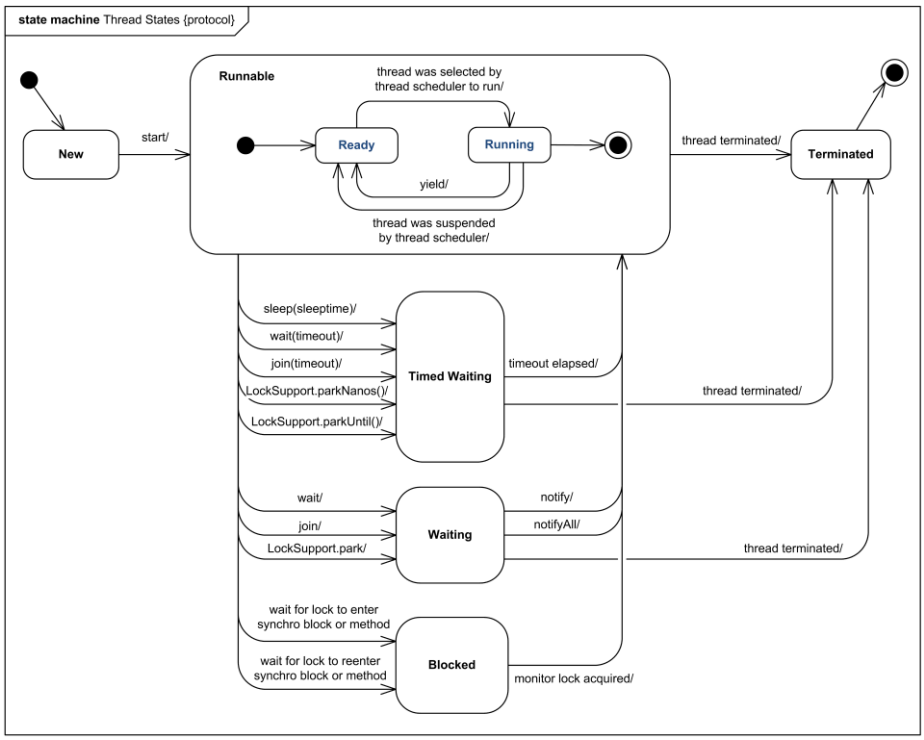


Fig. 7: Java 6 thread model (taken from [12]).

belonging to the same class. This is a preliminary thought concerning a construction of the simulation function.

6.1 Outline of Correctness Proof

We are in the process of carrying out a formal correctness proof. In the foreseeable future, we will concentrate on the following ingredients:

1. We will define a simulation relation $\preceq$ between the TA semantics (Section 3) and the Java execution semantics (Section 4). Both semantics are instances of timed transition systems.
2. We show that for each thread i, the Java execution semantics of thread i is simulated by the execution semantics of the TA obtained from abstracting (Section 5) the thread. Thus: Let J_i be the code executed by thread i. Let $TA_i = abstr(J_i)$ be the automaton obtained by the abstraction function sketched in Section 5. Then we show that $J_i \preceq TA_i$.
3. We establish an analogous simulation relation between the Java code of the scheduler J_S and the corresponding timed automaton TA_S (Figure 5), and show $J_S \preceq TA_S$.

4. Timed automata can be run in parallel, as sketched at the end of Section 3, by combining them with a parallel composition operator $\otimes_{TA}$. Similarly, there is a parallel operator $\otimes_J$ for composing thread-local actions (see statement evaluation semantics of Section 4) to obtain a combined transition semantics for the full Java state. We show that these operators are compatible with simulation: $J \preceq TA \wedge J' \preceq TA' \Longrightarrow J \otimes_J J' \preceq TA \otimes_{TA} TA'$. Combined with the results of (3), we show that our TA abstraction correctly simulates our full Java execution semantics.

Future work will concentrate on establishing a correspondence between timed transition systems (TTSs), logics, and simulation, in the following sense:

1. We will formalize a notion of execution trace of a TTS, formalize an appropriate temporal logic and establish the traditional model relation between traces and formulae of the temporal logic. In this context, we hope to be able to extend previous work described in [9].
2. We will show that simulation $\preceq$ induces a trace inclusion property, which again induces preservation of a certain class of models of formulae.
3. Using this result and the system refinement property of (4), we will show that the correctness formula established by model checking (1) makes indeed a correct prediction about the behaviour of our concurrent Java programs. Differently said: A Java program certified as free of resource access conflicts by model checking has no execution traces in which two threads access a resource at the same time.

7 Conclusions

This paper describes work in progress. The generation of Timed Automata out of Java code is still undergoing major changes, and the precise definition of a correctness statement and its proof still have to be done.

We may however comment on some fundamental assumptions of our approach, which may in part seem unrealistic.

- *Obtaining WCET annotations:* In the examples of this paper, the WCET annotations are fictitious values. There are WCET analysis tools especially geared at Java [10] that could be used to provide these annotations.
- *Worst-case vs. exact execution time:* Even then, the problem remains that our assumptions refer to the exact execution time of the code, whereas a WCET analysis only provides an upper bound. We intend to introduce `sleep` statements at the end of annotation statements so that a thread remains blocked until its WCET has indeed elapsed.
- *Granularity:* The size of code blocks we analyze (included, for example, in annotation statements) is not supposed to be in the order of a few instructions, but in the order of several hundred instructions. This is meant to reduce the relative error when estimating the WCET, and also to obtain reasonably-sized timed automata.

– *Non-interruptible annotation statements:* We presently assume that annotation statements are not interrupted, without verifying it. Future work will try to extend our approach in such a way
  - that threads are annotated with more accurate timing information (such as: periodicity) so that the mentioned assumption can be verified;
  - this assumption can be relaxed and interruption by higher-priority threads is possible, again under the hypothesis that the release parameters of periodic threads are known.

Acknowledgements. We are grateful to Marie Duflot-Kremer, Pascal Fontaine and Stephan Merz (Loria) and Jan-Georg Smaus (Irit) for discussions about this work.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
2. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
3. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets, Lecture Notes in Computer Science, vol. 3098, pp. 87–124. Springer Berlin / Heidelberg (2004), 10.1007/978-3-540-27755-2
4. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Lecture Notes in Computer Science. pp. 300–314. Springer-Verlag (2001)
5. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. ACM Trans. Program. Lang. Syst. 28(4), 619–695 (2006)
6. Kopetz, H., Bauer, G.: The time-triggered architecture. Proceedings of the IEEE 91(1), 112–126 (2003)
7. Lochbihler, A.: Verifying a compiler for Java threads. In: Gordon, A.D. (ed.) European Symposium on Programming (ESOP'10). LNCS, vol. 6012, pp. 427–447. Springer (Mar 2010)
8. The Real-Time for Java Expert Group: The Real-Time Specification for Java (Jan 2006)
9. Schimpf, A., Merz, S., Smaus, J.G.: Construction of Büchi automata for LTL model checking verified in Isabelle/HOL. In: Nipkow, T., Urban, C. (eds.) 22nd Intl. Conf. Theorem Proving in Higher-Order Logics (TPHOLs 2009). Lecture Notes in Computer Science, vol. 5674. Springer, Munich, Germany (2009)
10. Schoeberl, M., Pedersen, R.: WCET analysis for a Java processor. In: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems. pp. 202–211. JTRES '06, ACM, New York, NY, USA (2006)
11. Timnat, S., Braginsky, A., Kogan, A., Petrank, E.: Wait-free linked-lists. In: Ramanujam, J., Sadayappan, P. (eds.) PPOPP. pp. 309–310. ACM (2012)
12. uml-diagrams.org: Java 6 thread states and life cycle. http://www.uml-diagrams.org/examples/java-6-thread-state-machine-diagram-example.html?context=stm-examples