A Formal Model of Resource Sharing Conflicts in Multi-Threaded Java *

Nadezhda Baklanova
Université de Toulouse / IRIT
118 route de Narbonne
F-31062 Toulouse Cedex 9
nadezhda.baklanova@irit.fr

Martin Strecker
Université de Toulouse / IRIT
118 route de Narbonne
F-31062 Toulouse Cedex 9
martin.strecker@irit.fr

ABSTRACT
We present a tool for analyzing resource sharing conflicts in multi-threaded Java programs. We model execution of Java programs 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 ongoing work on formalization of Java semantics and the semantics of timed automata and partial verification of the translation procedure.

1. INTRODUCTION

1.1 General overview

Along with increasing usage of multi-threaded 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 [11, 23], 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.

*Research supported in part by the project *Verisync* (ANR-10-BLAN-0310)

Critical sections or locks are used in many applications in order to ensure concurrent access to objects. Even under the idealizing assumption that we can show absence of deadlocks, the temporal behavior of a thread cannot be predicted in a modular fashion, because a thread may be delayed indefinitely before getting access to the critical section or lock. However, we claim that with some global control, locking is not necessary if it is possible to verify that resource access of several threads does not occur concurrently, based on the temporal behavior of the threads.

The purpose of the article is twofold: Firstly, we present a tool for checking resource sharing conflicts in concurrent Java programs based on the statement execution time. This gives a “time-triggered” [15] 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 system of timed automata (TA) which is then model checked by a tool for timed automata (concretely, UPPAAL).

Secondly, we are interested in showing that this abstraction is sound, in the sense that if the model checker does not detect any conflicts, erroneous executions of the Java program are indeed excluded. For this, we develop a formal semantics of a real-time extension of Java, and use it to show a simulation property between the runs of Java programs and their corresponding TA abstraction.

In this paper, after an informal introduction (Section 2), we describe a prototype analyzer written in OCaml (Section 3). We then present the formal semantics of a multi-threaded, timed version of Java (Section 4), define the abstraction mechanism more formally and finally take a glance at the proof of soundness of the abstraction (Section 5).

The paper makes some simplifying assumptions; we state the most important ones here and take up some of them again in Section 6 to indicate how we can lift them.

- We only model a modest subset of Java (more details about the language features are in Section 4). In particular, method calls are not treated at all. We could inline static, non-recursive calls, which is however contrary to our long-term goals, namely to obtain a mod-

ular analysis.

- Our analysis is geared towards *verification* and not *inference*. In particular, the code has to be annotated with timing information, which we do not infer ourselves. We assume that this restriction can be lifted by a coupling with WCET analyzers.

- We assume that there is a fixed number of lockable objects that are allocated at program initialization time. This assumption is realistic in the restricted context of some embedded systems, but requires deeper analysis in the general case of programs allocating new objects.

- We assume that the execution of expressions takes no time. This is a preliminary simplification introduced in order to get our formal model straight and will be removed, expectedly without difficulty, in a more complete model. In a similar vein, at the present stage, we do not yet commit to a specific dialect or version of real-time Java.

A prototype of the analysis described in this paper is operational and can be obtained from the authors' web page[1]. The formal modeling of the semantics and the simulation proofs are carried out in the Isabelle proof assistant [18]; regularly updated snapshots are available from the same page.

1.2 Related work

Verification of concurrent systems with imposed timing constraints is a popular topic. Schedulability analysis of systems with timing and resource constraints is described in [8]. The authors consider not only periodic tasks but also sporadic tasks. They build a timed automata model addressing both timing and resource constraints which is further model checked with a TIMES tool.

An extension to multiprocessor systems is described in [6], where the authors perform schedulability analysis of a hard real-time system on a multiprocessor architecture. All tasks have hard deadlines, and the authors apply brute-force search combined with UPPAAL for ensuring search completeness.

The paper [20] contains schedulability analysis of multi-threaded 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.

Concurrent systems with fixed timing constraints can be modeled in timed automata with maximum $n + 1$ clocks where n is the number of tasks ([7]). The proposed approach can model both periodic and sporadic tasks.

The paper [25] tells about a tool JPF for model checking of Java concurrent programs during runtime by running it on a special JVM. It also has an extension for verification of Real-Time Java programs.

[1] http://www.irit.fr/~Nadezhda.Baklanova/jtres2013.html

The paper [10] demonstrates code generation for Real-Time Java from timed automata models. Based on a timed automata model Java code for tasks with hard deadlines can be generated. The correctness of the model is verified with UPPAAL. The inverse direction, namely abstraction from Java to TA, is described in [5]. This work, similar in spirit to ours, differs in that we are in addition interested in resource access conflicts, and a major concern for us is to obtain formal soundness proofs of our abstractions, based on a formal real-time Java semantics.

The use of abstractions of concurrent programs to a qualitative temporal logic (and not quantitative, as in our case) is described in [24]. Proof obligations extracted from the abstraction allow to show that a given program is linearizable and that its threads can be verified modularly.

Another approach to building of verified real-time systems are time-triggered architectures [15]. A timed-triggered system is a set of nodes each of them has a timing interface. The nodes can communicate with each other according to the interface specifications.

A language Giotto for time-triggered systems is presented in [12]. With the help of Giotto, hard timing constraints for real-time systems can be easily ensured which makes it suitable for real-time and safety-critical applications. The further development of Giotto is described in [13]. The authors discuss the concept of Hierarchical Timing Language which is the extension of Giotto which enhances modularity of programs.

Another tool for creating time-triggered systems is Periodic Finite State Machines [19]. The authors extend the finite state machines [9] with the notion of time and periodic tasks.

2. INTRODUCTORY EXAMPLE

To show the main idea, we present an example of a concurrent Java program (Figure 1). 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.

One of the possible executions is shown in Figure 2.

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

$$A\Box\ \forall(i : int[0, objNumber - 1])\ \forall(j : int[0, threadNumber - 1])\ \neg waitForLocksSet[i][j], \quad (1)$$

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

But in our case, there is a possible conflict, at instant 21 (in Figure 2, the red areas overlap). This conflict is also detected by `Uppaal`, which produces a trace leading to this error situation.

```
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++;
    }
  }
}
```

Figure 1: A concurrent Java program with possible resource sharing conflict.

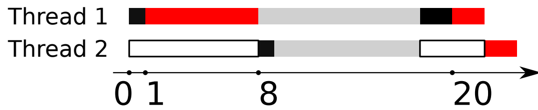


Figure 2: Possible execution flow. Black areas represent execution without locks, red areas - execution within a critical section, grey areas - sleeping, white areas - waiting for processor time.

3. ABSTRACTING JAVA TO TA

We consider a sequential model of program execution when a Java program executes on a single processor. This seems to be the focus of the current RTSJ (Real-Time Specification of Java [21]) standard. We can also accommodate multi-processor systems, but details are left for future work.

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.

3.1 TA and Uppaal

Timed automata are a popular formalism for modeling timed systems [2]. There are several model checkers like Uppaal, CADP, Kronos. We selected Uppaal [4] because of its popularity and good community support.

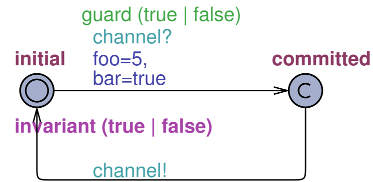


Figure 3: Uppaal primitives.

Timed automata consist of locations, transitions and background clocks. Several TAs can be run in parallel and share clocks and channels. All clocks advance uniformly. Uppaal extends classical timed automata with a small C-like programming language and allows to use variables in guards, invariants and updates.

A TA has one initial location. Locations can have invariants - Boolean conditions involving clocks (see Figure 3). A TA can stay in a location as long as its invariant is true; invariants enforce a TA to move ahead. In Uppaal, a user can declare some locations to be committed; this means that an automaton must leave this location immediately after it has entered it.

Transitions can have guards, channels and updates. Guards are Boolean conditions involving clocks which allow an automaton to take the transition only if the guard is true.

Channels serve to communication between automata. Automata in Uppaal can listen or call channels. If an automaton calls a channel, another one must be listening to this channel, and both automata advance simultaneously.

Updates can update variables and also reset clocks to 0.

The properties to be model checked are represented as TCTL (Timed Computation Tree Logic) formulas [1]. Uppaal can provide a counterexample (trace) if the model to be checked does not satisfy the property.

3.2 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 the `Runnable` interface must be nested classes in the class containing the `main` method. The required program structure is shown in Figure 4.

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. An 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

```java
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(r1,"t1"); //thread creation
    t1.start(); //thread start
  }
  private class Run1 implements Runnable{
    public void run(){ //thread logic
        implementation
      ...
    }
  }
}
private class Res1{ //resouce classes
  ...
}
```

Figure 4: Required program structure

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 important 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.

3.3 Execution model

In the sequential model we assume that at each moment of time only one of the threads or the scheduler can execute. Threads which do not execute at 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.

The building blocks and their translation are the following for the sequential model (also refer to Section 4.1 for the abstract syntax of our Java fragment):

(a) Assignment (5a). Three new locations 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 location `MIDDLE` is committed since we assume that any statement except the annotated one does not take time for execution.

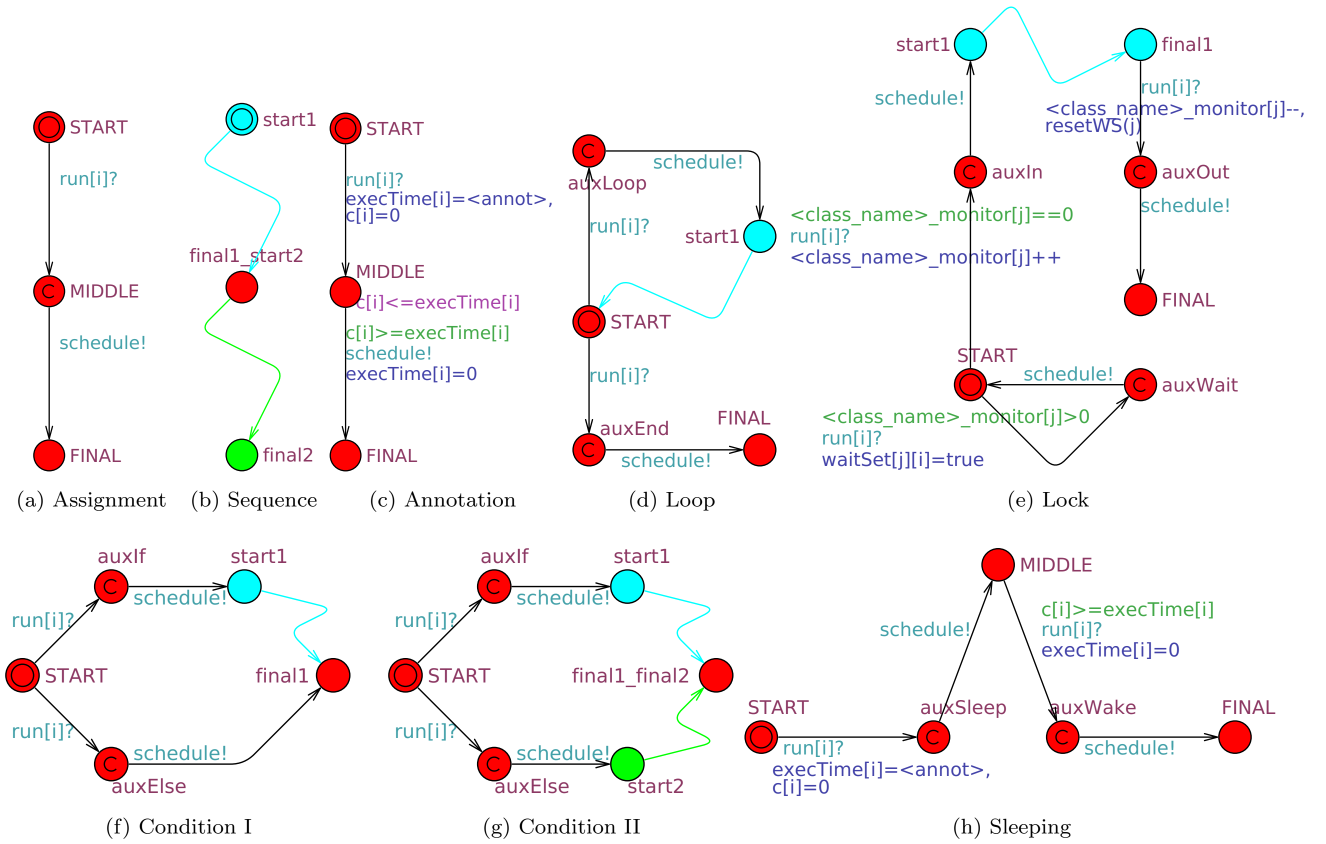
(b) Sequence (5b). Having two automata with start and final locations called `start1`, `start2` and `final1`, `final2` correspondingly, the locations `final1` and `start2` are merged.

(c) Annotation (5c). Three new locations 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 location `MIDDLE` has an invariant forbidding the automaton to stay in this location if the value of the clock `c[i]` bypasses `execTime[i]`. The transition from `MIDDLE` 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` location as long as the annotation claims.

(d) Loop (5d). Two meaningful locations and two auxiliary locations 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 location of the automaton corresponding to the loop body is merged with the `START` location.

(e) Lock (5e). Two meaningful locations and three auxiliary locations are added. The transition from `START` to `auxIn` listens to the channel `run[i]`, has a guard checking whether a lock for the object in the argument of the `synchronized` statement is not taken by other threads and increments the monitor value for the locked object. The transition from `final1` to `auxOut` listens to the channel `run[i]` and decrements the monitor value for the locked object. The transition from `START` to `auxWait` listens to the channel `run[i]`, has a guard checking whether a lock for the necessary object has already been acquired and updates the wait-for-lock set adding the current automaton to the wait-for-lock set corresponding to the locked object. The transitions from `auxIn` to `start1`, from `auxOut` to `FINAL`, from `auxWait` to `START` call the `schedule` channel.

(f) Condition (5f, 5g). Three new locations and four transitions are added. If both if and else branches are presented, the final locations of automata representing the branch internals are merged. The locations `auxIf` and `auxElse` are auxiliary locations 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`



(a) Assignment (b) Sequence (c) Annotation (d) Loop (e) Lock

(f) Condition I (g) Condition II (h) Sleeping

Figure 5: Building blocks for automata. Elements added on the current step are red; blue and green elements have been generated in the previous step.

(or to `final1` in case of absence of the else branch) call the channel `schedule`.

(g) Sleeping (5h). The automaton for sleep statements resembles the automaton for annotated statements with additional elements for returning control to the scheduler during sleeping. There are three meaningful and two auxiliary locations with transitions connecting them into a chain. The auxiliary locations, `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 the 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` location 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.

3.4 Scheduler

We currently only model an abstract scheduler and not a particular scheduling policy, which should be a refinement of the abstract one. It is the most non-deterministic scheduler one can conceive, and its sole purpose is to grant an exclusive right to run to a thread. Conflict-free programs can be expected to be conflict-free also under a more restrictive policy. Conversely, however, our liberal scheduler may allow conflicts to arise that might be prevented under a specific policy.

The scheduler model (Figure 6) has three locations: `scheduling`, `runThread`, `wait`. The scheduler starts in the location `scheduling` which has transitions for updating thread eligibility statuses. When all thread statuses are updated, the scheduler moves to the location `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 location `runThread`. When the thread has finished its execution, it calls a channel `schedule`, and the scheduler returns back to the `scheduling` location, and the new scheduling cycle starts. If there was no thread eligible for execution, the scheduler goes to the `wait` location 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 performed 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

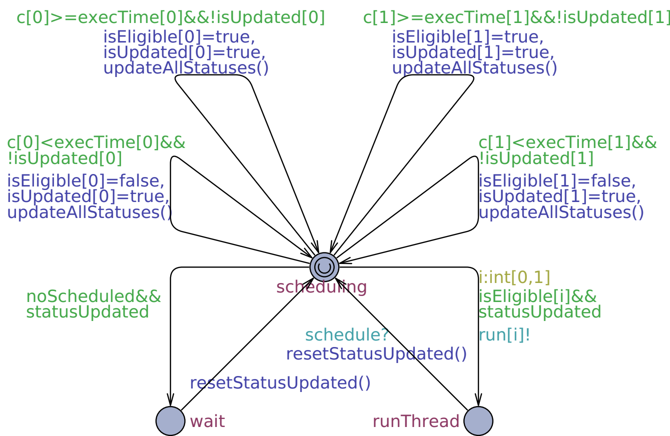


Figure 6: Scheduler for a system with two threads.

the thread with index `i` cannot be scheduled.

3.5 Model checking

After having built the system of timed automata from the initial Java program, we search for possible resource sharing conflicts by model checking. The property to be verified (1) ensures that at every execution path at every moment of time any wait-for-lock set is empty. That means, no resource sharing conflicts are found.

4. JAVA MODEL

One of the main purposes of this work is to formally prove the correctness of our abstraction. The general setup is similar to a formal compiler correctness proof, such as in the Jinja project [14] and its multi-threaded extension JinjaThreads [17] or the formal verification of a C compiler in the Compcert project [16]. This section takes a glimpse at the formalization of our fragment of real-time Java, and Section 5 provides a correctness argument of the abstraction.

4.1 Syntax

The abstract syntax is displayed in Figure 7. We make a distinction between expressions and statements: expressions are pure, *i.e.* they cannot change the state of a program, and they always return a value. Statements can change the state of a program, like heap values or locks, and they do not return anything. Expressions are evaluated at once, without any time loss, whereas statements are unfolded and evaluated step by step, and each of them takes some time to execute.

For now, we do not consider method calls or exceptions and concentrate on a straightforward execution flow. Neither do we work with `wait` and `notify` statements, nor with static variables.

Statements `InAnnot`, `InSync`, `WaitForSync`, `InSleep` cannot be present in a fresh initial program: they are used in intermediate steps during the execution.

4.2 Semantics

```
expr ::= Val literal | Var vname | Bop expr bop
     expr | Field expr vname cname

stmt ::= Empty
  | VarAssign vname expr
  | FieldAssign expr vname cname expr
  | Seq stmt stmt
  | If expr stmt stmt
  | While expr stmt
  | Annot annot stmt
  | InAnnot time stmt
  | Sync expr stmt
  | InSync expr stmt
  | Sleep expr
  | InSleep expr
```

Figure 7: Java grammar of the model. **vname** and **cname** arguments are strings, **literal** is an integer or boolean literal, or address (incl. **null**), **bop** is a binary operation like +, *, &, | etc.

```
full_state ::= {
  threads, // set of available threads
  th_info, // thread info: statement to be
      evaluated and state
  lk_info, // locks acquired by threads
  pl_info, // global time
  gl_info, // global state information;
      currently only heap, might also include
      static vars
  wl_info, // wait-for-lock sets
  pendingAct, // action still to be carried out
       by platform / scheduler
  deadlines, // annotation deadline for each
      thread
  sleepUntil, // sleep time value, time until a
       thread wakes up
  iheap, // a stamp of initial heap
  ilocals // a stamp of initial local variables
}
```

Figure 8: Program state structure.

The notion of program state (Figure 8) is complex and cannot be described in detail here. The state stores thread-local information for all threads such as local variable values and the expression to be evaluated, the lock state of shared objects, current time, action to be passed to the next step, wait-for-lock sets and deadlines of annotated and sleep statements.

Each object has lock information associated with it. It stores a flag indicating whether the object is locked or not, and a thread name which has locked this object. One thread can lock the same object several times, which corresponds to the reentrant locks behavior in Java. If a thread cannot acquire a lock, it puts itself into a wait-for-lock set for this object where it is stored until the required object is unlocked. Several threads can wait for the same lock; when the lock is released, all the threads are pushed out of the wait-for-lock set, and one of them finally acquires the lock.

When a thread enters an annotated or a sleep statement, it sets the corresponding deadlines. A deadline of annotated statement is absolute time before which the thread must finish to execute this annotated statement: not later than annotation value plus current time. Beyond annotated

$$\frac{preconditions \quad state\,update}{(e,s) \xrightarrow{action} (e',s')}$$

Figure 9: General form of reduction rules.

statements, the deadline is infinity. A deadline of a sleep statement is absolute time after which the thread is allowed to continue its execution, and infinity beyond sleep statements.

When a thread makes a step forward, it emits an action whose effect will be processed in the next step. The action depends on the type of the step: lock, start annotation, simple evaluation etc.

A single Java execution step is represented by four consequent substeps: scheduler, platform, evaluation and platform. The evaluation semantics is the semantics of a single thread, the scheduler semantics is responsible for thread scheduling, the platform semantics formalizes the notion of time advancement and works with deadlines.

$$\bullet \xrightarrow{scheduler} \circ \xrightarrow{platform} \circ \xrightarrow{evaluation} \circ \xrightarrow{platform} \bullet \rightarrow \ldots$$

Such a model is similar to the TA execution model which can be represented by repeating delay and transition steps:

$$\bullet \xrightarrow{transition} \circ \xrightarrow{delay} \bullet \xrightarrow{transition} \circ \xrightarrow{delay} \bullet \rightarrow \ldots$$

The scheduler behaves like a usual thread, which is allowed only to change the currently running thread and update threads' execution statuses like "eligible for execution", "sleeping" or "running". We do not model a particular scheduling strategy which can be added to the general model later. Currently, there are no constraints for selecting a thread from those eligible for execution.

For the moment, we have concentrated on the specification and abstraction of single thread semantics, namely, the evaluation and platform steps. The evaluation step represents small-step semantics of Java, where each step reduces an expression to be evaluated and also updates a part of the state. The platform step updates the part of the state responsible for time information: time and deadlines for annotated and sleep statements.

Every rule for both evaluation and platform steps (see Figure 9 for the general format) has some preconditions above the line, defining when this rule can be used, including parts of the state which are to be updated, and the reduction itself below the line. Below the line, there is a pair of statement and state which is transformed into another pair. The action above the arrow is used for communication between evaluation and platform steps. The selection of the proper rule on the evaluation step depends on the current statement to be evaluated. On the platform step the rule is selected based on the action produced by the previous evaluation step.

It is possible to reason about exact time values only on start/end of annotated or sleep statements. Inside an annotated statement, the exact time is not known but only its upper bound. If execution of annotated statement finishes before the deadline, the thread waits the remaining time until the deadline passes. If execution of annotated statement took more, and deadline was violated, the semantics would get stuck.

By means of example, two rules of evaluation semantics and two rules of platform semantics are presented in Figure 10. When reducing `Seq e1 e2` statement, the first substatement `e1` is reduced up to the end, and then `e2` is reduced. The rules for reducing substatements are other rules of the evaluation semantics.

When an annotated statement starts to reduce, the rule checks whether the deadline is correctly set to ∞ and updates the state `s` by setting the pending action to `StartAnnot t` where `t` is the annotation value. This action is passed to the platform semantics which finds some $\delta > 0$ which is a duration of the evaluation step execution. If such a δ is found, the semantics updates the deadlines to the value when the execution will have to finish and also the current time which is increased by δ.

The last rule is the rule of the platform semantics for end of annotated statement. There, it also finds some δ which is a duration of the evaluation step, resets deadlines to ∞ and updates the time as the execution took exactly the amount of time specified in the annotation. If execution of the annotated statement took less time than it was specified in the annotaion, the semantics consumes the remaining time so that its WCET becomes equivalent to the real execution time.

5. CORRECTNESS OF ABSTRACTION

A common way to prove the semantics preservation between two systems is to prove a simulation property. If a transition system TS_2 simulates another transition system TS_1, any step in TS_1 has a similar step in TS_2. We adopt general definitions from [3] for our transition systems.

Let $TS_1 = (S_1, \rightarrow_1)$ and $TS_2 = (S_2, \rightarrow_2)$ be transition systems where S_i is a set of states and $\rightarrow_i$ is a transition relation. For each s_i $Post(s_i) = \{s_i' : \ s_i \rightarrow_i s_i'\}$ denotes a set or states which have a transition from s_i to them, and $I_i = \{s_i : \exists s_i'.\ s_i \rightarrow_i s_i'\}$ is a set of all states which can be starting states of some transition.

Definition 1. *A simulation for (TS_1, TS_2) is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that*

1. $\forall s_1 \in I_1.\ \exists s_2 \in I_2.\ (s_1, s_2) \in \mathcal{R}$
2. *for all $(s_1, s_2) \in \mathcal{R}$ for any $s_1' \in Post(s_1)$ there exists $s_2' \in Post(s_2)$ and $(s_1', s_2') \in \mathcal{R}$.*

Definition 2. *A normed simulation for (TS_1, TS_2) is a pair $(\mathcal{R}, \nu_1)$ consisting of a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ and a function $\nu_1 : S_1 \rightarrow \mathbb{N}$ such that*

1. $\forall s_1 \in I_1.\ \exists s_2 \in I_2.\ (s_1, s_2) \in \mathcal{R}$
2. *for all $(s_1, s_2) \in \mathcal{R}$ for any $s_1' \in Post(s_1)$ at least one of the following conditions holds:*

$$\textsc{Seq}\ \frac{(e1, s) \xrightarrow{act} (e1', s')}{(Seq\ e1\ e2, s) \xrightarrow{act} (Seq\ e1'\ e2, s')}\ \textsc{eval} \qquad \textsc{StartAnnot}\ \frac{es_deadliness = \infty \quad s' = s\ \{es_pendingAct := StartAnnot\ t\}}{(Annot\ t\ e, s) \xrightarrow{StartAnnot\ t} (InAnnot\ t\ e, s')}\ \textsc{eval}$$

$$\textsc{StartAnnot}\ \frac{\exists\delta > 0.\ es_time\ s + \delta \leq t + es_time\ s \wedge s' = s\ \{es_deadlines := t + es_time\ s,\ es_time := es_time\ s + \delta\}}{(e, s) \xrightarrow{StartAnnot\ t} (e', s')}\ \textsc{pltf}$$

$$\textsc{EndAnnot}\ \frac{\exists\delta > 0.\ es_time\ s + \delta \leq es_deadlines\ s \wedge s' = s\ \{es_deadlines := \infty,\ es_time := es_deadlines\ s + \delta\}}{(e, s) \xrightarrow{EndAnnot} (e', s')}\ \textsc{pltf}$$

Figure 10: Some semantics reduction rules.

- *there exists* $s_2' \in Post(s_2)$ *with* $(s_1', s_2') \in \mathcal{R}$.
- $(s_1', s_2) \in \mathcal{R}$ *and* $\nu_1(s_1') < \nu_1(s_1)$.

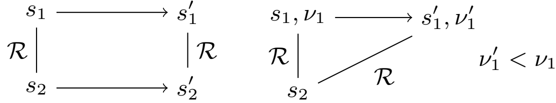


Figure 11: Two kinds of steps in weakly similar systems.

The second case in Figure 11 represents a silent step, when two different states of one system are mapped to a single state of another system.

We want to formalize and verify the translation procedure from Java to TA. However, it is too complex to be verified in one step, therefore we divided the whole translation into two stages. On the first stage, we abstract from the concrete variables, `if` and `while` conditions, also fold constants in expressions as well as remove the internal structure of annotated statements so that they are translated into an atomic construction. The other constructors are essentially the same as on the Java level. After the first stage it remains to generate a sequence of TA locations with names for each constructor and to transform the preconditions of the semantic rules into TA notions such as guards or invariants. On each stage we want to prove that the system after the translation simulates the system before the translation.

$$Java \xrightarrow{simulation} NLTA \xrightarrow{simulation} TA$$

The intermediate construction is called nameless timed automata (NLTA). It does not contain the information from the Java level which is lost in the TA model, neither does it contain explicit named locations from the TA definition. The grammar is presented in Figure 12. The semantic rules are similar to their analogs in Java semantics with the difference that `NLIf` and `NLWhile` do not have conditions so that any of the branches can be taken. The concrete values of time and adresses do not need to be evaluated again. Finally, `NLAssign` does not have any parameters since we abstract from concrete variables and expressions.

Taking into account the abstractions made during the translation from Java to NLTA, its algorithm is quite straitforward (Figure 13). We translate each Java constructor to the corresponding NLTA constructor and recursively translate each argument, if any. The arguments of `Sleep` and `Sync` statements are folded to a concrete value.

```
nlaut ::= NLEmpty
  | NLAssign
  | NLSeq nlaut nlaut
  | NLIf nlaut nlaut
  | NLWhile nlaut
  | NLAnnot time
  | NLInAnnot time
  | NLSleep time
  | NLWakeup time
  | NLSync addr nlaut
  | NLInSync addr nlaut
```

Figure 12: Grammar of nameless timed automata.

```
jToNL NLEmpty = NLEmpty
jToNL (VarAssign vname expr) = NLAssign
jToNL (FieldAssign expr vname cname expr) =
    NLAssign
jToNL (Seq stmt1 stmt2) = NLCompos (jToNL
    stmt1) (jToNL  stmt2)
jToNL (If expr stmt1 stmt2) = NLIf (jToNL
    stmt1) (jToNL  stmt2)
jToNL (While expr stmt1) = NLWhile (jToNL
    stmt1)
jToNL (Sleep t) = NLSleep (exprToInt t)
jToNL (InSleep t) = NLWakeup (exprToInt t)
jToNL (Annot time stmt) =  NLAnnot time
jToNL (InAnnot time stmt) =  NLInAnnot time
jToNL (Sync obj stmt) =  NLSync (exprToObj obj)
     (jToNL  stmt)
jToNL (InSync obj stmt) = NLInSync (exprToObj
    obj) (jToNL  stmt)
```

Figure 13: Translation of statements.

The definition of a NLTA state (Figure 14) is remarkably simplified in comparison with a Java state. It contains only information about acquired locks, wait-for-lock sets, local time of the automaton and the currently active automaton. All the components have the same meaning as in the Java state except the time (Figure 15). In contrast to Java global time, each nameless automaton has its own local time represented by a pair $(current_time, deadline)$ where $current_time$ is the analog of Java time and is the same for all automata, and $deadline$ is the deadline computed based on the time value of the current annotated or sleep statement or infinity, if the automaton is not inside an annotated or sleep statement.

Since the execution semantics are defined for a thread-local (automaton-local) state, we use them in translation. The local states extract the information related to the thread

```
nlGlState ::= {
  gl_auts, // a set of automata
  gl_monitors, // locks info
  gl_locksWaitFor, // wait-for-lock sets
  gl_time, // local times for each automaton
  gl_runningAut // currently running automaton
}
```

Figure 14: NLTA global state.

```
esToLocState es = {
  monitors=(es_locks es),
  locksWaitFor=(es_waitForLocks es),
  loc_time=(es_time es,
            min (snd(es_deadlines es))
                (snd(es_sleepUntil es))),
  runningAut=runThread es }
```

Figure 15: Translation of states.

(automaton) from the global state.

The necessary parts of Java full state are simply translated to their analogs in NLTA except the time which is constructed from Java time and deadline values. The first component of NLTA time is the Java global time, the second component is the minimum of deadlines for annotated and sleep statements. Since we do not allow nested annotated statements or a sleep statement inside an annotated statement, at least one of the deadlines is infinity at each moment of time. Note that we assume translated Java programs to be correct, i.e. they do not throw exceptions and do not violate deadlines. In an exceptional situations Java semantics gets stuck, and the correctness of translation is not guaranteed.

Silent steps (see def. 2) can occur inside an annotated statement only, therefore deadline of annotated statement is finite, and possible time values are bounded from above. As a measure we use the Java global time. Each semantics step on the Java level takes some positive time δ for execution (an integer value, thus precluding Zeno behavior), therefore the global time strictly increases during the execution. As a function ν_1 we could take

$$\nu_1(stmt, state) = es_deadlines\ state - es_time\ state.$$

If $\nu_1(x) < 0$, the deadline is violated, and the execution is not correct, i.e. cannot be translated. Otherwise, ν_1 monotonely decreases when the execution advances.

Combining translation rules for Java statements and Java states we can prove simulation of Java and NLTA. Currently, we have finished the proof of correctness of the first stage of translation from Java to NLTA for a single thread. To prove the full case it remains to inject the scheduler semantics between thread-local steps.

THEOREM 1. *The functions (`jToNL`, `esToLocState`) and the introduced measure ν_1 are a normed simulation of Java and NLTA transition systems.*

The final goal is to show the simulation between Java and TA semantics and prove that the correctness formula established by model checking (1) makes indeed a correct prediction about the behaviour of our concurrent Java programs. 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.

6. CONCLUSIONS

We have presented a tool for static analysis of concurrent Java programs which allows to find possible resource sharing conflicts. We have also presented a formal semantics of multithreaded Java with time aspect and a partial formalization of the translation procedure performed by the tool.

In the nearest future we want to add support of SCJ constructs such as subclasses of `Mission` with their semantics, in particular, `CyclicExecutive`. Another question open to discussion is implementing a particular scheduling strategy in order to restrict the number of possible executions and make the analysis more precise.

We may now review some of the fundamental assumptions of our approach and possible future work in the long-term.

- *Obtaining WCET annotations:* In the examples of this paper, the WCET annotations are fictitious values. To obtain realistic anotations, we plan to couple our analysis with WCET analysis tools especially geared at Java, such as [22].
- *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), Martin Schoeberl (DTU) and Jan-Georg Smaus (IRIT) for discussions about this work.

7. REFERENCES

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414 –425, June 1990.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] E. Fersman and W. Yi. A generic approach to schedulability analysis of real-time tasks. *Nord. J. Comput.*, 11(2):129–147, 2004.

[9] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 18(6):742–760, 1999.

[10] N. Hakimipour, P. Strooper, and A. Wellings. A model-based development approach for the verification of real-time Java code. *Concurrency and Computation: Practice and Experience*, 23(13):1583–1606, 2011.

[11] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Lecture Notes in Computer Science*, pages 300–314. Springer-Verlag, 2001.

[12] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[13] C. M. Kirsch and A. Sokolova. The logical execution time paradigm. In S. Chakraborty and J. Eberspächer, editors, *Advances in Real-Time Systems*, pages 103–120. Springer Berlin Heidelberg, 2012.

[14] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.

[15] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[16] X. Leroy. A formally verified compiler back-end. *CoRR*, abs/0902.2137, 2009.

[17] A. Lochbihler. Verifying a compiler for Java threads. In A. D. Gordon, editor, *European Symposium on Programming (ESOP'10)*, volume 6012 of *LNCS*, pages 427–447. Springer, Mar. 2010.

[18] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002.

[19] R. Obermaisser, C. E. Salloum, B. Huber, and H. Kopetz. Modeling and verification of distributed real-time systems using periodic finite state machines. *Comput. Syst. Sci. Eng.*, 23(4), 2008.

[20] 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.

[21] The Real-Time for Java Expert Group. *The Real-Time Specification for Java*, Jan. 2006.

[22] M. Schoeberl and R. Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, JTRES '06, pages 202–211, New York, NY, USA, 2006. ACM.

[23] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. Wait-free linked-lists. In J. Ramanujam and P. Sadayappan, editors, *PPOPP*, pages 309–310. ACM, 2012.

[24] B. Tofan, G. Schellhorn, S. Bäumler, and W. Reif. Embedding rely-guarantee reasoning in temporal logic. Technical Report 2010-07, Informatik, 2010.

[25] W. Visser, K. Havelund, G. P. Brat, and S. Park. Model checking programs. In *ASE*, pages 3–12, 2000.