Towards Monitoring Temporal Properties with JamaicaVM

Clemens Ballarin
acias GmbH
Haid-und-Neu-Straße 18
76131 Karlsruhe, Germany
ballarin@acias.com

ABSTRACT

JamaicaVM is a real-time capable Java virtual machine used in the embedded domain. We report on work towards monitoring properties, specified with linear temporal logic, of (concurrent) applications running on JamaicaVM. The approach is based on observing events emitted by the scheduler of the virtual machine while it runs. Consequently it is also amenable to monitoring internals of the virtual machine itself.

Categories and Subject Descriptors


General Terms

Concurrency, Java, linear temporal logic, monitoring, runtime verification

1. INTRODUCTION

Concurrent systems are inherently hard to debug, and the widespread use of multicore processors has increased the problem. While previously the concurrent activities of a program were eventually linearised to a sequence of instructions in software (by the scheduler of the operating system) these decisions are now taken by processor hardware at a much finer level of granularity, leading to an enormous growth in possible execution paths. Consequently ensuring the proper functioning of concurrent software including synchronisation primitives such as monitors and rendezvous themselves has become more difficult.

An approach to debugging concurrent systems is to systematically analyse traces emitted by the system for undesired behaviour. This approach is known as runtime verification [9], and in this work we investigate its feasibility in the context of the Java virtual machine JamaicaVM [1] and for properties concerning the order in which events occur. Linear temporal logic (LTL) is a formalism capable of expressing such properties [10]. By using tracing features available with JamaicaVM, both user applications and the VM itself can be monitored for properties specified in this logic. Unlike many other approaches to runtime verification, our approach is directly based on the infinite-trace semantics of LTL and not limited to safety properties. Instead we make the assumption that the observed application executes a cyclic activity, which is a reasonable restriction in the domain of embedded real-time systems.

The consumer-producer relationship is a classical example in the domain of concurrency, and it will be used here as the fil rouge of the paper. Our version of the consumer-producer relationship is the Java program shown in Figure 1 consisting of the classes SharedBuffer and ProducerAndConsumer. With exception of the statements in lines 21 and 29 on the left side, which are specific to JamaicaVM, the code is based on the standard Java API only.

Readers unfamiliar with Java should not be deterred by the apparent size of the example. A shared buffer object — that is, an instance of SharedBuffer, consists of a state, which is either EMPTY or FULL, and, in the latter case, the single value stored in the buffer. Synchronisation is achieved by declaring the methods for accessing the buffer synchronized, which protects them through a monitor associated with the buffer. The methods write(int) and read() are provided for writing and reading, respectively. After accessing the value field, they set the state accordingly and notify all threads waiting on the monitor of this change. The buffer implementation is fairly simple, and it is the duty of the application to ensure that a buffer is only read when it is full and written when it is empty. To this end, the method await(State) is provided, which blocks until the buffer is set to the state given in the argument. The main method, in ProducerAndConsumer, allocates a buffer and then starts two concurrent threads, which transfer a continuous stream of data through the buffer.

2. JAMAICA VM

JamaicaVM is a real-time capable Java execution platform that is available for a wide range of operating systems and hardware platforms and that supports the Real-Time Specification for Java [11]. Its garbage collector can be used even in domains where hard real-time guarantees need to
import static com.aicas.jamaica.lang.Scheduler.recordUserEvent;

public class SharedBuffer {
    public static enum State {
        EMPTY, FULL;
    }
    private State state = State.EMPTY;
    private int value;
    public synchronized void await(State expected) throws InterruptedException {
        while (state != expected)
            wait();
    }
    public synchronized void write(int data) {
        value = data;
        recordUserEvent("write");
        state = State.FULL;
        notifyAll();
    }
    public synchronized int read() {
        int data;
        recordUserEvent("read");
        state = State.EMPTY;
        notifyAll();
        return data;
    }
}

(a) Shared buffer with mutually exclusive access

private static class Producer implements Runnable {
    private SharedBuffer channel;
    public Producer(SharedBuffer buffer) {
        channel = buffer;
    }
    public void run() {
        int i = 0;
        while (true) {
            channel.write(i++);
            try {
                channel.await(SharedBuffer.State.EMPTY);
            } catch (InterruptedException e) {
            }
        }
    }
}

private static class Consumer implements Runnable {
    private SharedBuffer channel;
    public Consumer(SharedBuffer buffer) {
        channel = buffer;
    }
    public void run() {
        while (true) {
            try {
                channel.await(SharedBuffer.State.FULL);
            } catch (InterruptedException e) {
            }
            channel.read();
        }
    }
}

(b) Producer and consumer threads

Figure 1: Two threads communicating through a shared buffer

Figure 2: JamaicaVM scheduler event trace visualised by the ThreadMonitor
Today, LTL is widely used — for example as input and eventuality and will be adopted here as the specification temporal logic, LTL) is well suited for expressing invariance are temporal in nature. Linear-time temporal logic (linear constraints on a concurrent system, and these constraints are expected to alternate, and it is the purpose of synchro-

3. LTL AND BÜCHI AUTOMATA

In the example program from Figure 1, write and read are expected to alternate, and it is the purpose of synchronisation to ensure this. In general, synchronisation imposes constraints on a concurrent system, and these constraints are temporal in nature. Linear-time temporal logic (linear temporal logic, LTL) is well suited for expressing invariance and eventuality and will be adopted here as the specification language. Linear temporal logic was introduced to computer science by Pnueli [10] as a tool for verifying concurrent programs. Today, LTL is widely used — for example as input language of the Spin model checker [7]. Our presentation of LTL loosely follows [13].

3.1 Linear Temporal Logic

LTL is propositional logic augmented by temporal operators. LTL formulas are built from propositional variables using negation (¬), conjunction (∧), the binary operator “until” (U) and the unary operator “next” (X). If P is a finite set of propositional variables and φ an LTL formula with propositional variables from P then φ is called a formula over P.

Let P be a finite set of propositional variables. LTL formulas are interpreted in infinite words over the alphabet 2^P. Intuitively, a word u ∈ (2^P)^ω represents an infinite sequence of state transitions, and u(i) represents the set of events that occur in the ith transition. The semantics of LTL formulas is given in

**DEFINITION 1.** Given an LTL formula φ over P, a word u ∈ (2^P)^ω and i ≥ 0, the meaning of “φ holds in u at i”, denoted u, i |= φ, is defined thus:

- u, i |= p if p ∈ u(i), for every p ∈ P
- u, i |= ¬φ if u, i |= φ, for every formula φ over P
- u, i |= φ ∧ ψ if u, i |= φ and u, i |= ψ, for formulas φ and ψ over P
- u, i |= φ U ψ if there exists j > i such that u, j |= ψ and u, i' |= ψ for all i' such that i ≤ i' < j
- u, i |= X φ if u, i + 1 |= φ

That is, φ U ψ means that φ holds from the current point in time until a point is reached in the future where ψ holds; X φ means that φ holds in the next step. We will say that an LTL formula φ holds in u, u |= φ if it holds at the beginning. The language defined φ is

\[ L(φ) = \{ u ∈ (2^P)^ω \mid u, i |= φ \} \]

Other commonly used temporal operators can be defined in terms of U:

- F φ = true U φ (“finally”)
- G φ = ¬F ¬φ (“globally”)

Here are examples of LTL formulas, where w and r are propositional variables:

1. G(w → r): whenever w then not r
2. G(w → F r): every w is followed (possibly at the same point in time) by an r
3. G(w → X(¬w U r)): every w is followed later by an r and w does not occur in between.

LTL formulas will be used for expressing properties of event traces emitted by a Java program running on JamaicaVM. For example, if w represents the user event write and r the user event read, the third example formula reads “every write is followed by a read and no write occurs in between. This formula is expected to hold for the Java program from Figure 1.

In fact, since read and write should not occur simultaneously, the formula can be strengthened slightly to

\[ G(w → ¬r \land X(¬w U r)) \]
For the Java program, the converse of (1) — that is, every read is followed by a write — holds as well, and consequently also their conjunction:

\[ G(w \rightarrow \neg r \land X(\neg w \lor r)) \land G(r \rightarrow \neg w \land X(\neg r \lor w)) \quad (2) \]

Note that (2) requires the program, if either a write or a read occur, to run indefinitely, while (1) does not. It only requires that every write is eventually followed by a read. That is, if there are no more writes, the program may as well terminate.\(^3\)

3.2 Automata

Finite automata are simple model of computation and are well-known in computer science. Finite automata accept finite words over finite alphabets, and the sets of accepted words are the regular languages (sets of words generated by regular expressions) over these alphabets. Büchi automata are a model of infinite computation — that is, of processes that do not terminate. Consequently Büchi automata accept infinite words. Büchi automata and LTL formulas are closely related. For every LTL formula \( \varphi \) there is a Büchi automaton that accepts exactly the language of \( \varphi \).

We first introduce the necessary automata theory from the literature, and then state the relation to LTL. The exposition of Büchi automata follows Khoussainov and Nerode’s textbook\(^4\), which proved to be an excellent and accessible introduction.

**Definition 2.** A finite automaton over the alphabet \( \Sigma \) is a quadruple \( A = (S, I, T, F) \) where

- \( S \) is a finite nonempty set called set of states,
- \( I \subseteq S \) is the set of initial states,
- \( T \subseteq S \times \Sigma \times S \), \( T \neq \emptyset \) is the transition table, and
- \( F \subseteq S \) is the set of final states.

**Definition 3.** A run of a finite automaton \( A \) on \( u = \sigma_1 \cdots \sigma_m \in \Sigma^* \) is a sequence \( s_1, s_2, \ldots, s_m+1 \) of states such that \( s_i \in I \) and \( (s_i, \sigma_i, s_{i+1}) \in T \) for all \( i \leq m \). The automaton accepts \( u \) if there is a run \( s_1, \ldots, s_m+1 \) on \( u \) such that \( s_{m+1} \in F \).

The mechanics of Büchi automata is identical to that of finite automata, and Definition 2 also applies to these. The notion of run and the acceptance criterion need to accommodate infinite words, though.

**Definition 4.** A run of a Büchi automaton \( A \) on \( u = \sigma_0 \sigma_1 \cdots \in \Sigma^\omega \) is an infinite sequence \( s_0 s_1 \cdots \) of states such that \( s_0 \in I \) and \( (s_i, \sigma_i, s_{i+1}) \in T \) for all \( i \in \mathbb{N} \). The automaton accepts \( u \) if there is a run \( s_0 s_1 \cdots \) on \( u \) and a final state \( s \) such that \( s \) appears in \( u \) infinitely often.\(^5\)

While a finite automaton accepts a word if it stops in a final state, a Büchi automaton accepts a word if it visits an accepting state infinitely often. A good intuition for this is that the run eventually becomes cyclic. A language \( L \) is called recognisable by a class of automata (either finite or Büchi) if there is an automaton \( A \) of that class such that \( u \in L \) if, and only if \( A \) accepts \( u \). Formally, the intuition of cyclic runs is expressed by Büchi’s Characterisation:

**Theorem 5.** An \( \omega \)-language \( L \subseteq \Sigma^\omega \) is Büchi recognisable if, and only if \( L \) is a finite union of sets \( VW^\omega \) where \( V, W \subseteq \Sigma^* \) are recognisable by finite automata.

For every LTL formula \( \varphi \) there is a Büchi automaton that accepts exactly the words \( L(\varphi) \).\(^6\) Considerable effort went into making this result practically applicable, and several translators are available. In this work we used LTL 2 BA\(^3\), which is publicly available as a web service at \( \text{http://www.liv.ens-cachan.fr/~yasmin/ltl2ba/} \).

Figures 3 and 4 show Büchi automata generated by LTL 2 BA from the formulas (1) and (2) respectively.\(^4\) In the diagrams, initial states are indicated by “start” and final states by double circles. The alphabet is \( 2^P \), where \( P \) is the set of propositional variables. Transitions are labeled with subsets of \( 2^P \), which, as a short notation, are denoted with propositional formulas. A propositional formula \( \chi \) represents the set \( X \subseteq 2^P \) such that \( \sigma \in X \) if, and only if \( \chi \) holds for \( \sigma \).

We will now illustrate Büchi’s Characterisation on the automaton in Figure 3. Considered as a finite automaton, it recognises the regular language

\[ W = ((\neg w) \lor (\neg r) (\neg w) (\neg w \land r))^\omega. \]

Since there is only one final state, it is easy to see that as a Büchi automaton it recognises the language \( W^\omega \), and it

\(^3\)Büchi’s original result is for formulas of the monadic second order logic of the successor function SIS. Since every LTL formula can be converted to an equivalent SIS formula, the result also applies to LTL\([3]\).

\(^6\)In June 2013, using LTL 2 BA version 1.1.
Event sequence

<table>
<thead>
<tr>
<th>Event sequence</th>
<th>Current states</th>
</tr>
</thead>
<tbody>
<tr>
<td>{kind=MONITOR_EXIT, delta=4117, threadID=8, monitor=12330}</td>
<td>(2, 3, 4)</td>
</tr>
<tr>
<td>{kind=MONITOR_ENTER, delta=290, threadID=7, monitor=12330}</td>
<td>(2, 3, 4)</td>
</tr>
<tr>
<td>{kind=STATE_CHANGE, delta=488, threadID=7, newState=3, causingThread=8}</td>
<td>(2, 3, 4)</td>
</tr>
<tr>
<td>{kind=STATE_CHANGE, delta=3647, threadID=8, newState=4, causingThread=8, reason=16}</td>
<td>(2, 3, 4)</td>
</tr>
<tr>
<td>{kind=STATE_CHANGE, delta=4311, threadID=7, newState=0, causingThread=7}</td>
<td>(2, 3, 4)</td>
</tr>
<tr>
<td>{kind=USER, delta=3650, threadID=7, text=write, userKind=0}</td>
<td>(4, 5) final</td>
</tr>
<tr>
<td>{kind=MONITOR_EXIT, delta=3975, threadID=7, monitor=12330}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=MONITORENTER, delta=377, threadID=8, monitor=12330}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=STATE_CHANGE, delta=285, threadID=8, newState=3, causingThread=7}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=STATE_CHANGE, delta=4872, threadID=7, monitor=12330, owner=8}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=STATE_CHANGE, delta=399, threadID=7, newState=4, causingThread=7, reason=8}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=STATE_CHANGE, delta=5262, threadID=8, newState=0, causingThread=8}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=MONITOR_EXIT, delta=2919, threadID=8, monitor=12330}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=MONITORENTER, delta=256, threadID=7, monitor=12330}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=STATE_CHANGE, delta=302, threadID=7, newState=3, causingThread=8}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=STATE_CHANGE, delta=4851, threadID=8, monitor=12330, owner=7}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=STATE_CHANGE, delta=403, threadID=8, newState=4, causingThread=8, reason=8}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=STATE_CHANGE, delta=4946, threadID=7, newState=0, causingThread=7}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=MONITOR_EXIT, delta=3173, threadID=7, monitor=12330}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=MONITORENTER, delta=286, threadID=8, monitor=12330}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=STATE_CHANGE, delta=323, threadID=8, newState=3, causingThread=7}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=STATE_CHANGE, delta=3301, threadID=7, newState=4, causingThread=7, reason=16}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=STATE_CHANGE, delta=4228, threadID=8, newState=0, causingThread=8}</td>
<td>(1, 2, 4)</td>
</tr>
<tr>
<td>{kind=USER, delta=3344, threadID=8, text=read, userKind=0}</td>
<td>(2, 3)</td>
</tr>
<tr>
<td>{kind=MONITOR_EXIT, delta=3394, threadID=8, monitor=12330}</td>
<td>(2, 3, 4)</td>
</tr>
<tr>
<td>{kind=MONITORENTER, delta=179, threadID=7, monitor=12330}</td>
<td>(2, 3, 4)</td>
</tr>
</tbody>
</table>

Figure 5: Analysis of the trace from Figure 2

passes through the final state whenever processing a word from $W$ is completed. Choosing the language $\varepsilon$ that only contains the empty word for $V$ completes the characterisation.

4. ANALYSING SCHEDULER EVENTS

Our goal is now to design a tool that reads scheduler data from JamaicaVM via a network connection. The user input is LTL formulas, and at the core Büchi automata verify at

from JamaicaVM via a network connection. The user input

4.1 Processing Prefixes

Our approach to this problem is pragmatic and based on what one does frequently in everyday life. When trying to find out whether a system is still alive, and no other means are available, one will usually wait for a reasonable amount of time. If the system shows no signs of life, one will usually wait for a reasonable amount of time. If the system shows no signs of life, one concludes that the system hangs (and, in the case of software, aborts it). This means that one makes an assumption, perhaps based on previous experience, at what interval the system emits events.

Transferred to the monitoring problem, if the execution path of the monitored application is cyclic, and the maximal duration $d$ of one iteration is known, it is sufficient to ensure that the Büchi automaton passes through a final state once per interval $[t, t + d)$. If this is not the case the formula is not satisfied by the trace. Otherwise, the automaton has passed through the final state in $[t, t + d)$ and we know that the formula was still satisfied in the previous interval $[t − d, t)$. A formal justification of this reasoning can be given by Büchi’s Characterisation (Theorem 5), which says that accepted traces are elements of sets $\{W\}^*$, and observing that the assumption of a cyclic computation with bounded duration of iterations maps to an upper bound on the length of words in the sets $W$. If a final state is encountered in $[t, t + d)$, a word $u \in W$ for some $W$ from the characterisation theorem must just have been processed, and a word in $W \in \{W\}^*$ for suitable $n$ and $W$ has been processed so far. This word is a finite prefix of an infinite word accepted by the automaton.

4.2 Experiments

In order to evaluate the approach in practice, a Java library for reading and processing JamaicaVM’s scheduler events was provided, and on top of it the automaton from Figure 3 implemented. This experimental monitor was used to analyse traces created by the consumer-producer program. Figure 3 shows part of the created log, observed events to the left, state of the automaton to the right. A scheduler event consists of a number of fields, and which fields are present depends on the event kind. The delta field contains the elapsed time since the previous event (here in ticks of the CPU cycle counter). The other fields are more or less self-explanatory. The trace is the same as the one visualised by the ThreadMonitor in Figure 3 and it contains two user events write and read. This is where the state of the automaton changes.

The automaton is non-deterministic, and one event can trigger several transitions. Hence the automaton is in general in several states at the same time — that is, it performs...
several computations simultaneously. We see that at the write event, the automaton passes from state 3 to 5 and then to 1. At read it goes back from 1 to 3.

This is not the only computation that takes place, but all other computations eventually fail — for instance, right before the write event, the automaton is also in states 2 and 4; since there are no matching transitions, these computations fail. The reason, why 4 occurs again at write is the transition from 3 to 4. (The states 2 and 4 have no transitions through which final states can be reached, and therefore could have been removed from the automaton.)

The shown section of the trace contains one iteration of a cyclic process (as long as the program correctly alters the timing behaviour of programs and is done in the Java PathExplorer [5]. The other lacks the next operator \( X \) [4]. Both make the logic less expressive, for a formula that states that every \( w \) is eventually followed by an \( r \) and vice versa only holds for finite words that contain neither \( w \) nor \( r \).

Our approach is based on the standard semantics of LTL, and its advantages are that standard tools such as the construction of Büchi automata from formulas can be used without modification and that it is amenable to the full set of properties expressible in LTL. Based on the assumption that the application is eventually cyclic, and given an upper bound \( d \) on the maximal duration of a single iteration, it can be decided with latency no larger than \( d \) whether the observed trace is a prefix of a word for which a formula holds.

6. ACKNOWLEDGEMENTS

The author would like to thank Christian Haack for his feedback on a draft of this paper. The work was funded in part by the European Union as Artemis Joint Undertaking, Project CONCERTO, Grant Agreement 333053.

7. REFERENCES


