A formal model of the Java\textsuperscript{*} multithreading system
and its validation on a known problem

Serge Chaumette
Serge.Chaumette@labri.u-bordeaux.fr

Asier Ugarte,
Asier.Ugarte@labri.u-bordeaux.fr
LaBRI, Laboratoire Bordelais de Recherche en Informatique,
Université Bordeaux I,
351 Cours de la Libération,
33405 TALENCE,
FRANCE

Abstract

Java is becoming more and more important in various communities. It is widely used for developing classical, distributed and real time applications. One of its key features in these domains is its multithreading system.

Although a specification exists, it is informally written in the English language. Therefore, an additional formalization effort is required. This paper focuses on that point.

Our aim is to provide a model that can be reused and help in the processes of: using Java threads to gain deep knowledge of their behavior; designing new threading systems taking the best out of Java threads still avoiding their main drawbacks; proving properties – this is for instance what we need in other research projects carried out in our team on automatic distribution of objects.

The model that we have set up uses transition systems. To check that it corresponds to the informal specification, we use the MEC model checker. Also, we use MEC to automatically check the properties we are interested in. For example, we use it to exhibit a known problem of the Java threading system: the handling of \texttt{long} and \texttt{double} variables.

1. Introduction

The work presented in this paper takes place in the framework of a project carried out at the Laboratoire Bordelais de Recherche en Informatique (LaBRI), Université Bordeaux I. The aim of this project is to provide a distributed platform\cite{6,7,8} that offers homogeneous access to heterogeneous resources of a network. This platform is based on Java\cite{5}, RMI and CORBA\cite{26}. The basic execution unit that we provide to the programmer is the thread. A thread is a flow of execution within a process. Multithreaded systems offer multiple flows of execution, i.e. multiple threads, within the same process. Java is multithreaded\cite{20}.

The specification of the Java Virtual Machine as published by Sun in [21] covers all the aspects of the execution of a Java application on that platform. Since they are provided in the English language, not using a formal notation, these specifications are not always quite clear and straightforward to understand. Therefore, a formalization effort is required if we want to come with unambiguous, widely understood specification of the Java multithreading system.

In this paper, our aim is to explain how we build a formal model of the Java threading system as it is defined at the level of the virtual machine. It is a crucial issue, because is it important for both network and parallel and distributed computing communities. Multithreading systems are used intensively, for instance, to achieve computation/communication overlapping. A chapter in the specifications of Java is dedicated to threads. Our goal is to serve four main purposes. First the model we build will be available for other projects. Second, this will serve for educational purpose. Third, we hope that it will help in the design of new – better – threading systems. Fourth, this model will be used for other research activities carried out in our team, especially the work that is being done on automatic distribution of objects\cite{16}. This last activity requires

\textsuperscript{*} Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. The authors are independent of Sun Microsystems, Inc.
the proof of some properties on objects making up the application, properties that involve threads.

The rest of this paper is organized as follows. In section 2 we present related work in the domain of Java modeling. We then describe Java threads from a practical point of view in section 3, still remaining at a high level of abstraction. In section 4 we comment on the specifications as given by Sun. In section 5 we explain and show, on some examples, how we translate these informal specifications to our model. We then use the MEC[3] transition system based software tool to check the validity of our model in section 6. We eventually conclude and consider future uses and evolutions of the research results presented in this paper.

2. Related Work

Many research projects in the domain are targeted to the definition of a formal specification of the Java Virtual Machine. In most cases the approaches are restricted to a formal model of a subset of the instructions or bytecode of the JVM, and these approaches focus on different features depending on the context in which they apply, i.e. the effective aim of the model. Many of them have for concrete result the identification of possible failures of systems that obey the specification of the Java Virtual Machine.

We have selected some of the projects that we consider the most representative of the work that is carried out in the domain. These are:

- the specification by Zhenyou Quian at the Kestrel Institute in California[25]. He has formalized a subset of the instructions of the Java Virtual Machine by defining it as a static and typed transition system. He focuses on the bytecode verifier and the class loader, his aim being to formalize the security model of Java 2[15];

- **The Defensive Java Virtual Machine[9]** project carried out by Richard Cohen at Computational Logic, Inc., Austin, Texas. It describes a formal model of the JVM and uses runtime verifications to ensure safety of the types being used. The model is build in a logical expression language called ACL2[22], based on a subset of Common Lisp[28];

- the project of Dan Lambright from the University of Danemark. He has been working on the semantics of the bytecode of Java[19]. He has formally described a subset of the JVM and of the instructions of the bytecode. In his model, the JVM is described as a fimited state machine, and each bytecode instruction is described as a transition rule that specifies the changes of states and the conditions when the execution succeeds;

- the work carried out by R. Stata and M. Abadi at Compaq[27], now being carried out by S. Freund and J. Mitchell, at Stanford University[13], the aim of which is to prove the robustness of the system and to make the semantics of some Java bytecodes more clear.

There are other research projects around the formalization of the JVM. They mainly focus on a specific part of the specifications and often do not consider high level features like the management of several threads, exceptions handling, class loading or garbage collection. We have not found any model that integrates the notion of multithreading therefore we began the research effort described in this paper.

At the same time, we find nowadays a lot of work being done in model checking Java programs. Most of them focus on providing a tool to translate Java language to the input languages of verification tools. We can cite JavaPathFinder [17] and JCAT [11] that translate Java programs to PROMELA programs, the input language of the SPIN [1] model checker. Other tools translate Java programs to their own intermediate language, as Bandera [10] and other works being done at the Stanford University [24]. Some of these projects target the Java multithreading system and they allow to detect errors due to multithreading, such as deadlocks. Nevertheless, these tools work directly upon the Java language or the Java bytecode and try to verify properties on Java programs and not on the Java Virtual Machine used to execute them.

3. High level view of Java threads

3.1. Implementation of the Java multithreading system

In this section we show a possible implementation of the Java multithreading system that satisfies the specifications that Sun Microsystems publishes for the Java Virtual Machine[21]. The first specification of the JVM was published in 1996. It has hardly changed since then and still supports the Java language[18] that is available today. Indeed, this specification supports the different releases that have been designed till now, i.e. 1.02 and 1.1, as well as the most recent Java 2 – formally called Java 1.2 or 1.3 –.

The effective implementation of threads in the Java Virtual Machine is not far from the implementations that have been adopted by other systems. Figure 1 illustrates the different data areas that take part in the execution of a thread in the Java Virtual Machine. Here is a short description of them:

**The program counter.** The Java Virtual Machine can manage several threads executing in a concurrent manner. Each thread has its own program counter. At
any given time, a thread executes the code of a single method. The program counter contains the address of the instruction of the method that is being executed. The Java language makes it possible to invoke methods written in different languages. These are called native methods. If the method that is being executed is native, then the value of the program counter is undefined.

The stack. Each thread has its own private stack that is created when the thread is created. This stack is structured in frames. Entering a new method causes a new frame to be pushed on the stack. The way the stack is used is basically equivalent to the way the stack of a classical process is used (pushing local variables, the return values of the methods, etc.).

The heap. This is a memory area that is shared by all the threads. It contains all the class instances and hence the variables shared by all the threads. This area is managed by the garbage collector.

Methods area. This memory area is shared by all the threads. It contains the definition of the different classes, as well as the code of the methods.

The native stack. This is the private stack used to execute the native methods of a thread.

3.2. Threads and the Java bytecode

To be executed by the Java Virtual Machine the Java code is compiled to a binary format into a .class file. This binary code is independent of the target platform. It is known as the Java bytecode. The structure of these binary files will not be detailed here (see [21] for further details). The body of the Java methods is translated to a sequential sequence of calls to Java bytecode instructions. These instructions define the instruction set of the Java Virtual Machine.

Among the many instructions of the Java Virtual Machine, some are closely related to the management of threads. An example – that we present here because it will be reused later in this paper – is the `getfield` instruction that copies the value of a variable from the heap to the stack (see figure 1).

4. Informal Java threads specification

Now that we have explained the basic operations of Java threads, we explain and clarify the specifications of the Java multithreading system as described in The Java Virtual Machine Specification by Tim Linholm and Franck Yellin[21]. The model that we propose in section 5 is based on these specifications.

4.1. Overall model of the Java threading system

In this section we describe the approach adopted in [21]. We illustrate the components involved in this informal model by means of the schema presented figure 2.
A set of **low level actions** is defined. These actions can be used to explain the interactions between threads and the main memory.

A set of **constraints** is defined. These constraints must be obeyed in order to ensure the integrity of data. They set up an order among the actions generated by threads.

The **operations that transfer a value (read or write)** between the main memory and the local memory of the thread are split in two phases. So that a read or write operation is effectively achieved, it must first have been validated by the thread and by the main memory. Therefore the operation takes place in two phases, with some sort of intermediate transfer area (see figure 2). This makes it possible to relax the synchronization constraints between the different components of the system. This could be used to implement optimizations in a virtual machine or in a compiler. Some research activities in the domain[14] have already shown the usefulness of this model in a distributed framework.

### 4.2. Constraints on the behavior of Java threads

The execution of threads is directed by a set of rules that are given in the specification provided by Sun in [21]. There are mainly three kinds of constraints that control:

1. the basic behavior of the system;
2. the relationships between instructions executed by a thread;
3. the relationship of threads with their environment, mainly the main memory.

From a formal point of view, although expressed in the English language, some of these constraints let the reader guess the underlying intrinsic automaton. This is unfortunately not the case for all of them. Nevertheless, this is one of the reasons why we chose the model of transition systems for our model.

### 4.3. Virtual instruction set

The behavior of individual Java threads is described by means of a set of virtual instructions that represent basic operations that threads can achieve. These instructions make it possible to describe the basic operations of the main memory, the interactions of threads with memory and the lock on variables. These operations are not necessarily those offered by a real Java Virtual Machine implementation but are used only for a descriptive purpose.

These operations are:

- **use**
  
  reads the contents of a local variable from local memory;

- **assign**
  
  assigns a value to a variable in local memory;

- **load**
  
  gets the value of a variable as transferred by the main memory and assigns it to its corresponding copy in local memory;

- **store**
  
  transfers the value of a variable in local memory to the global memory;

- **read**
  
  the main memory transfers the value of a variable to the thread;

- **write**
  
  stores the value of a variable transferred by the thread in main memory;

- **lock, unlock**
  
  although dealt with in our work, operations on locks will not be detailed here.

For instance, the bytecode instruction `getfield` – see section 3.2 – can be implemented using the `load` and the `read` virtual instructions.

### 5. Effective construction of the model

To achieve our goal we considered several different description languages: Petri nets[30], Milner’s CCS[23], finite transition systems[2]. For the reasons explained above, we eventually chose finite transition systems.

#### 5.1. Components of the system

The Java threading system basically contains within its specification a set of entities. Among these are the threads, local memories and global memory.

We have added components that do not directly appear in the specifications. We use these entities to carry the semantics of the synchronizations. This will be explained in section 5.4.

#### 5.2. Basic constraints

The basic constraints are those that are mandatory when describing the behavior of any system in terms of states and transitions. There are four of these constraints. Here is one of them as given in [21]:

→ "The actions performed by any one thread are totally ordered; that is, for any two actions performed by a thread, one action precedes the other."

These basic constraints are *de facto* contained within the model that we use since it is based on transition systems.
5.3. Local constraints

Local constraints are those that specify the order in which a thread can achieve its own operations. There are 8 of them. These constraints do not take relationships between threads into account. Form a practical point of view, the aim of local constraints is mainly to avoid threads useless work. For instance the aim of the constraint expressed as follows in [21]:

→ "A store operation by [a thread] T on [a variable] V must intervene between an assign by T of V and a subsequent load by T of V."

is mainly to ensure that there is no useless local memory assignment carried out by a thread. This is illustrated in figure 3.

Figure 3. Illustration of one of the constraints that enforce useful uses of variables by a thread

Such a constraint can directly be coded to a regular expression and then to an automaton. We model this constraint as the automaton shown figure 4. The notation \(^{\text{instruction}}\) means all but instruction.

5.4. Synchronization constraints

We call synchronization constraints, those constraints that define the relationships between operations carried out by different threads. Since all the information that are shared between threads are exchanged through the main memory, all of these synchronization constraints are expressed as relations between threads and the main memory. There are 7 of these constraints.

To illustrate our approach we consider one of these constraints as given in [21]:

→ "Each store action by a thread is uniquely paired with a write action by the main memory such that the write action follows the store action."

This constraint says that the main memory must be ready to write the value of a memory location when this is required by a thread, and that it cannot write any value without a thread requiring it.

To handle that kind of synchronization constraint, we introduce new entities that do not exist as such in the specifications, and that obey a set of rules expressed by means of a transition system. For instance, the constraint considered here is modeled by a process that we have introduced, and that we call a variable, since it is used to model the usage of memory locations, i.e. of variables. The resulting automaton is shown figure 5. It models not only the constraint given above but also all the constraints that deal with memory transfers.

This variable is the formalization of the transfer area that we introduced to explain the informal specifications in section 4.1.

Figure 4. Automaton modeling one of the constraints that enforce useful uses of variables by a thread

Figure 5. Behavior of a variable
5.5. Global system

We have set up a global model of the system by synchronizing all of the automaton previously defined. Figure 6 shows all the components of this global model and the synchronizations that have to be applied between them. The synchronizations are carried by the links between entities; these links represent the synchronization vectors between processes. In order to keep this model tractable, we have applied a set of simplifications regarding the number of threads, of variables and locks – that have not been detailed in this paper. Nevertheless, it remains significant regarding the behavior of a system of any size for the properties that we want to show.

Figure 6. Global system

6. Using the MEC software tool to verify the model

The aim of this section is to show how we prove expected properties or exhibit failures that are already known in order to check the model that we have set up.

We remind the reader that we intend to provide a formal model that first describes the informal specification of the Java Virtual Machine, and second offers a formal support that can be used to check behavioral properties. Therefore, what we want to check here, is the fact that our model effectively models the informal specifications, with their known properties and their possible problems.

Once the model is available as a transition system, we use a software tool called MEC[3] to automatically check the properties we are interested in.

6.1. Overview of MEC

MEC[3] is a model checker that implements the so-called Arnold-Nivat[4] model and Dicky’s logic[12]. It makes it possible to build and analyze a model of a system of processes. To do that, it computes the synchronized product of a transition system. The reader can refer to [3] for a complete description of MEC, of its operators and functions.

Once specified within MEC, the model we have set up leads to a transition system that is made up of 209,464 states and 4,608,048 transitions. The creation of the system inside MEC on a Sun Ultra Sparc with 384 Mo of RAM takes around 12 minutes.

6.2. Deadlock freeness

The first property to establish is either the absence of deadlocks, or if there are deadlocks, the delayed deadlock situations, i.e. states that inevitably lead to a deadlock.

This request is easily described using basic operators and functions that are provided by MEC. For instance, the states that represent a system that cannot evolve any longer are computed from the following formula:

$$\text{dead} := \ast - \text{src}(\ast)$$

where $\text{src}(T)$ are the source states of the set of transitions $T$. After this formula has been evaluated, the set $\text{dead}$ contains all the states of the system but those that are the source of at least one transition. In other words, $\text{dead}$ contains all the states from which the system cannot evolve any longer.

Applied to our model, the result is the empty set, showing there is no deadlock, as expected.

6.3. Checking of the model

In order to check in a more significant manner that the model corresponds to the specification as given by Sun in [21], we tried to exhibit a known problem of the Java threading system. The problem we consider is related to $\text{long}$ and $\text{double}$ variables. The instructions used to handle these variables work on 32 bits, although they are 64 bits long.

Figure 7 shows two threads executing concurrently. This execution exhibits a behavior that can be considered faulty...
in that although it obeys the specification, the result can be
different from what is basically expected. The first thread,
called ThreadZero clears all the bits of the _field variable. The second, called ThreadOne sets all the bits of the same variable to 1. The 64 bits long variable is shown as two 32 bits parts, that we call A and B. The operation on A and B are atomic. The figure shows the transitions that make up the shortest path from the initial state to a faulty state. This state is considered faulty – at least problematic – because, if the variable _field is read at that point, it neither contains the value 0 nor the value 1, i.e. none of the values written either by ThreadZero or ThreadOne.

![Figure 7. An example of the 64 bits variables problem](image)

The path shown figure 7 is a result provided by MEC based on the following call sequence, given here for illustrative purpose – for further details see [29] and [3] –:

```java
/* transitions identifying faulty situations */
errors:=!label[1]="verifyER";
/* sources of transitions in errors */
faulty_states:=src(errors);
/* transitions that make up a shortest */
/* path to a faulty state */
min_faulty_path:=trace(initial,*,faulty_states);
```

The fact that our model exhibits this behavior that is also contained within the informal specifications is an other validation of our results.

6.4. Additional properties

One of the other basic properties of the Java multithreading system is the statement that says that any two threads can communicate by means of a shared variable and that, in such a case, the consistency of the local copies and the integrity of the variable can be guaranteed by using the available system of locks.

This property which is more difficult to show and that requires space to explain will not be detailed here. We have shown in [29] that this property holds.

7. Conclusion

In the current state of the research described here, we have a formal model of the Java multithreading system. We have shown that it closely sticks to the specifications given by Sun in [21]. We have shown that it carries known problems that the informal specification also carries. This is a first validation of our work.

Our model also has some limitations in that it is purely a formalization of the informal specification of the multithreading system of the Java language. It does not describe an abstract, perfect, multithreading system, but describes a specific one, with known properties, either good or bad. The limitations of our model are those of the effective Java multithreading system.

To get back to the purpose of this research as we introduced it at the beginning of this paper, we can say that this model is now available both for research projects and for educational purpose. We also hope that it will help in the design and implementation of future multithreading systems.

Eventually, it will support other research carried out in our team, regarding automatic distribution of objects[16]. This work requires the verification of some complex properties on objects, and especially on objects that are shared by threads.

References