Static Analysis Via Abstract Interpretation of the Happens-Before Memory Model

Pietro Ferrara

École Polytechnique
F-91128 Palaiseau, France
Pietro.Ferrara@polytechnique.edu
Dipartimento di Informatica
Università Ca' Foscari di Venezia
I-30170 Venezia, Italy

Abstract. Memory models define which executions of multithreaded programs are legal. This paper formalises in a fixpoint form the happens-before memory model, an over-approximation of the Java one, and it presents a static analysis using abstract interpretation. Our approach is completely independent of both the programming language and the analysed property. It also appears to be a promising framework to define, compare and statically analyse other memory models.

Keywords: Static Analysis, Abstract Interpretation, Memory Model, Multithreaded Programs.

1 Introduction

While the improvement of single-core architectures is slowing down, many multi-core processors, like the family of Intel® Core™, are appearing in a broad market [10]. The only way to take advantage of this technology is to develop multithreaded programs that perform many parallel tasks.

The semantics of a programming language supporting multithreading must be defined well enough that developers can fully understand which behaviours are allowed during an execution, and which are not. In the literature, a common approach has been to consider as incorrect all the programs containing data races [19], i.e. in which two parallel threads access without any synchronisation action the same area of shared memory, and not to provide any semantics in this case. In this way, many static analyses have been aimed at proving the absence of data races [16,20]. Leaving completely unspecified the semantics of these programs is unsatisfying for modern programming languages, particularly those that are focused on security issues.

The attention on this topic has increased during the last years: for instance, the first specification of the Java Virtual Machine [12] was flawed [18], and only a following work [13] provided a correct definition. Nowadays, the specification of the memory model appears to be the "lingua franca" that can fill this lack on the semantics specification. In this context, two main but opposing approaches are

B. Beckert and R. Hähnle (Eds.): TAP 2008, LNCS 4966, pp. 116–133, 2008.

considered: (i) to restrict the non-deterministic behaviours in order to provide a simple reference to the developers; (ii) to allow as many compiler optimisations as possible, but that introduce non-deterministic behaviours.

On this topic, the debate is still in progress [2], and different ideas and solutions have recently been proposed [23].

Most state-of-the-art static analyses do not support multithreading, or they deal only with the possible interleavings of instructions; this is why they are not sound w.r.t. the memory model, as it usually allows more behaviours than the ones exposed by sequentially consistent executions.

Contribution: In this context, a static analyser able to approximate all the possible runtime behaviours of a multithreaded program w.r.t. a memory model seems to be particularly appealing, as it would help developers reason about compiler optimisations and all the possible interleavings due to the parallel execution of multiple threads [24]. Moreover, since the threads communicate implicitly through the shared memory, really subtle and unwanted interactions may arise, and a static analysis may trace and provide information about all these potential bugs. Some examples about these situations, like the one depicted by our running example, are presented by [13].

The happens-before memory model is an over-approximation of the Java one; for this reason, tuning a static analysis at this level will allow us to obtain sound results for Java multithreaded programs.

Our approach follows the abstract interpretation framework [4,5]. We first define the concrete trace semantics in a fixpoint form, aimed at formalising the happens-before memory model. Then we abstract it, proving the soundness of our analysis. Our analysis is generic to the programming language, as the happens-before memory model is. Moreover, our framework shall be used to formalise, compare, and statically analyse other memory models. The proposed analysis approximates all the possible multithreaded behaviours w.r.t. the happens-before memory model, starting from a given intra-thread domain and semantics. Our approach allows that the intra-thread semantics follows the sequential consistency rule (as most static analyses do); the multithreaded executions that are sensitive to compiler optimisations and threads' interleavings are obtained through the computation of a fixpoint.

The rest of the paper is organised as follows. In this section we present a running example. Section 2 introduces the happens-before memory model, and analyses it from a static analysis point of view. The concrete domain and semantics are defined by Section 3, while Section 4 depicts the abstract ones. Section 5 presents and discusses some related work, and Section 6 concludes.

1.1 The Running Example

The concepts in the rest of the paper are explained in the context of a simple example.Figure 1 depicts a Java-style program composed of two threads where variables `i` and `j` are shared between them. Supposing that at the beginning

Thread 1	Thread 2
`i=1;`	`if(j==1 && i==0)`
`j=1;`	`  throw new Exception();`

Fig. 1. The running example

`i` and `j` are both equal to 0, which runtime behaviours are acceptable and consistent w.r.t. the memory model? And in particular: may the exception be thrown?

1.2 Abstract Interpretation

Abstract interpretation is a theory to define and soundly approximate the semantics of a program [4,5]. Roughly, a concrete semantics, aimed at specifying the runtime properties of interest, is defined; then it is approximated through one or more steps in order to finally obtain an abstract semantics that is computable, but still precise enough to capture the property of interest. In particular, the abstract semantics must be composed of an abstract domain, an abstract transfer function, and a widening operator in order to make the analysis convergent. Usually, each state of the concrete domain is composed of a set of elements (e.g. all the possible computational states), that is approximated by an unique trace in the abstract domain.

In our analysis, the concrete semantics is computed by a fixpoint that produces the set of all the possible finite executions of a multithreaded program; these executions respect the happens-before memory model. The abstract semantics computes, always through a fixpoint, an abstract element that approximates the concrete semantics. The soundness of the approach has been proved following the abstract interpretation framework.

2 The Happens-Before Memory Model

In the recent literature, the memory models have been aimed at formalising the behaviours that are allowed during the execution of a multithreaded program.

The Java Memory Model was presented by [18]. Its formalisation involves many different components, and all the run-time actions must be committed in a quite sophisticated way. In the same paper the happens-before memory model is formalised, as an over-approximation of the Java one, i.e. it allows a larger number of runtime behaviours. Its formalisation is simpler, and it allows us to reason in terms of static analysis.

The main components of this model are (we denote some rules with a specific name that will be used during the formalisation of the model in the fixpoint form):

– the program order, that, for each thread, totally orders the actions performed during its execution;

- a synchronises-with relation that relates two synchronised actions. For instance, the acquisition of a monitor synchronises-with all the previous releases of the same monitor. Moreover the first action of the execution of a thread is synchronised-with the action that launched it (rule IN);
- the happens-before order initially introduced by [11]. An action a_1 happens-before another action a_2 (rule HB) if (i) a_1 appears before a_2 in the program order; (ii) a_2 synchronises-with a_1; (iii) if you can reach a_2 by following happens-before edges starting from a_1 (i.e. the happens-before order is transitive).

Through the happens-before order, a consistency rule is defined. In particular, it states that a read r of a variable v is allowed to see a write w on v if: (i) r does not happens-before w (i.e. a read can not see a write that has to be executed after it); (ii) there is no write w' on v that happens-before r and w happens-before it (i.e. there is not any write on the same variable that has to be executed between the observed write and the read, overwriting it) (rule OW).

The happens-before memory model says nothing about what is a variable and its granularity (an object, a field, an array, a primitive value, ...).

2.1 Reasoning Statically

One point is not clear in these definitions: on one hand the definition of the happens-before consistency appears to be a static rule, but on the other hand the program order talks about a total order covering all the actions of an execution; this concept is clearly dynamic. Since our approach is parameterized on the abstract intra-thread transition relation, we suppose that it approximates this program order; in this way if a state appears before another one in the trace produced through this relation, it means that it will always be executed before it.

About the synchronises-with relation, threads generically synchronise on some elements (for instance in Java they synchronise on monitors defined on objects), and the mutual exclusion is guaranteed on them. In this way, they acquire a synchronisable element, keep it during some actions, and finally release it. In a static context, we do not know which thread acquires the synchronisable element. For instance, imagine the multithreaded program of Figure 2.

Thread 1	Thread 2
`acquire(o)`	`acquire(o)`
`var=v1`	`temp=var`
`var=v2`	`release(o)`
`release(o)`	

Fig. 2. An example

Which values may thread 2 read? The read action is synchronised on the same element of both writes in thread 1. It may read the initial value stored in `var`,

or `v2`, but not `v1`, as its acquisition synchronises-with the release of thread 1, or vice versa its release is synchronised-with the acquisition of thread 1.

This consideration allows us to conclude that statically a read action is allowed to see all the values written by parallel threads, except the ones that are overwritten by a successive action and such that all the actions between them are synchronised at least on an element locked also in the state that is going to perform the read action. This is a straight consequence of the mutual exclusion principle.

All these concepts are formalised by the concrete semantics.

2.2 The Running Example

Let us apply these concepts to the running example depicted by section 1.1, and in particular to state if it is consistent that the exception may be thrown under the happens-before memory model. To answer this question, we evaluate which values may be read by the condition of the `if` statement of thread 2.

First of all, since there are no synchronisation actions, the synchronise-with order is empty, and all the actions of thread 1 do not happen-before the evaluation of the condition. So this instruction is allowed to see the initial value of variable `i` equal to zero, and the value written by the second instruction of thread 2 that assigns 1 to variable `j`. Therefore, it is consistent to evaluate this condition to `true`, and to throw the exception.

For instance, the exception is thrown if the two statements of thread 1 are switched by the compiler (since they are independent, this is allowed), a single-core processor executes `j=1`, and then the control switches to thread 2, in which the condition of the `if` evaluates to `true` .

2.3 Notation

We denote the sets of functions by capital Greek letters, the elements by a single lower-case letter, and the identifiers of sets always begin with a capital letter.

Concrete and abstract: The concrete sets and elements are denoted as just defined, while the abstract ones are over-lined; for instance, if S is a concrete set, the respective abstract set is denoted by $\overline{S}$. About the functions, if *fun* is the concrete one, its respective abstract version is denoted by $fun^{\#}$.

Trace semantics: Our concrete and abstract semantics are based on partial finite traces. Roughly, the execution of a thread is represented by a trace of states. ϵ denotes the empty trace, while $\sigma_0 \rightarrow \sigma_1 \rightarrow \cdots$ denotes a trace that begins with a state σ_0 followed by σ_1, and then there is an arbitrary number of successive states, denoted by $\cdots$. Given a transition function $\rightarrow$: $St \times St \mapsto \{\texttt{true}, \texttt{false}\}$, with an abuse of notation we denote the fact that $\rightarrow (\sigma_1, \sigma_2) = \texttt{true}$ by $\sigma_1 \rightarrow \sigma_2$. We denote by $St_{\rightarrow}$ the set of blocking states following the transition relation $\rightarrow$, i.e. such that $\forall \sigma_1 \in St_{\rightarrow} : \nexists \sigma_2 \in St :\rightarrow (\sigma_1, \sigma_2) = \texttt{true}$. Finally, let be S a generic set of elements, we denote by $S^{\dot{+}}$ the set of all the finite traces composed of elements in S.

3 Multithreaded Concrete Semantics

In this section we present the multithreaded concrete semantics. This semantics is aimed at formalising the happens-before memory model in a fixpoint form; it is completely parameterized by the concrete operational semantics that defines the behaviour of intra-thread atomic computational steps, and on some functions that given a state returns a part of it. In this way we completely separate the semantics of the language from its memory model.

Since the happens-before memory model refers only to finite executions, we consider only finite traces. Our multithreaded concrete semantics produces all the complete executions, i.e. in which the executions of all the threads end with a blocking state.

3.1 Required Elements

In order to define the happens-before memory model on the concrete semantics we need some sets and functions that extract information from the states.

For the sets, we denote by TId the set of the threads' identifiers, by Sh all the possible shared memories, by Loc the shared memory locations, by Val the values, by $Sync$ all the shared elements on which a thread can synchronise, and by St the states that contain the memory and control state of a single thread. Moreover, as the happens-before memory model talks about the values read and written on the shared memory, we suppose that the shared memory relates each location to a value ($Sh : Loc \mapsto Val$).

We suppose that a transition function $\overset{\circ}{\rightarrow}$: $St \times St \mapsto \{\texttt{true}, \texttt{false}\}$ is provided, and that it defines the single step behaviours of the computation. We require that these steps are atomic at thread level, i.e. it is not possible for another thread to see an intermediate state during a single intra-thread transition.

We also require that the following functions are provided:

- $shared : St \mapsto Sh$, given a state it returns the shared memory contained in it;
- $action : St \mapsto \bot_a \cup (\{\texttt{r}, \texttt{w}\} \times Loc \times Val)$, given a state it returns the operation it is going to perform (reading from or writing on the shared memory), the shared location on which it operates and eventually the written value, or $\bot_a$ if it has performed another type of operation;
- $synchronised : St \mapsto \wp(Sync)$, given a state it returns all the elements of the memory state on which it is synchronised (for instance the set of all the monitors previously locked and not yet released); we do not specify what these elements are, since many different ways of synchronisation exist, and we are generic with respect to the programming language.

Finally, we require the function $set\_shared : St \times Sh \rightarrow St$, that given a state and a shared memory returns a state equal to the given one but in which the shared memory is replaced by the given one.

3.2 Thread Partitioned Concrete Domain

Our concrete domain is aimed at collecting information about the parallel execution of different threads. In this way we partition the trace of the execution relating each active thread to the trace of its execution. Moreover, we collect for each thread the one that has launched it, and the number of the state of its execution trace that is produced after this operation; for the main thread, that is launched by the system, we use a special value $\bot_\Omega$. In this way our concrete domain is composed by two functions, where the second one is just aimed at maintaining some information on the relations between threads. We collect the number of the state in order to restrict the execution trace only on the states successive to the launch of the thread, and so to respect the rule IN.

$$\Psi : TId \to St^{\vec{+}}$$
$$\Omega : TId \to ((TId \times Integer) \cup \bot_\Omega)$$

3.3 Single Step Definition

We define a *step* function that performs a single intra-thread step, consistent with respect to the happens-before memory model, and returns the set of the possible states obtained after it.

Definition 1 (*step* function). *Starting from the active thread, a multithreaded state containing the traces of the executions of all the threads, and an element of Ω, the step function returns the set of all the possible following states.*

In particular if the thread is not going to read from the shared memory, it computes the step while observing the sequential consistency rule (point (1)). Otherwise it may: (i) perform the step following the sequential consistency (point (2a)); (ii) select one visible value following the happens-before consistency and perform the step injecting this value in the shared memory (point (2b)).

Formally,

$$step : TId \times \Psi \times \Omega \mapsto \wp(St)$$
$$step\ (t, f, s) = \{\sigma\}\ where\ f(t) = \sigma_0 \to \cdots \to \sigma_i\ and$$
$$(1)\quad \sigma_i \overset{\circ}{\to} \sigma \qquad if\ \pi_1(action(\sigma_i)) \neq \boldsymbol{r}$$
$$(2a)\quad \sigma_i \overset{\circ}{\to} \sigma \vee \qquad if\ \pi_1(action(\sigma_i)) = \boldsymbol{r}$$
$$(2b)\quad \exists v \in visible(t, \pi_2(action(\sigma_i)), synchronised(\sigma_i), f, s(t)) :$$
$$\sigma' = set\_shared(\sigma_i, shared(\sigma_i)[l \mapsto v]), \sigma' \overset{\circ}{\to} \sigma$$

Definition 2 (*visible* function). *The visible function returns the values that are visible by the given thread. This set is built up by the values produced by the thread that launched the one that is reading, restricting it only on the part of the trace executed after the launch (point (1), rules IN), and the values produced by other threads (point (2)).*

$$visible : TId \times Loc \times \wp(Sync) \times \Psi \times ((TId \times Integer) \cup \bot_\Omega) \mapsto \wp(Val)$$
$$visible\ (t, l, S, f, (t', i')) =$$
$$(1)\quad = project(l, suffix(f(t'), i'), S) \cup$$
$$(2)\quad \{v : v \in project(l, f(t''), S) : t'' \in dom(f) \setminus \{t, t'\}\}$$

Definition 3 (*suffix* function). *The suffix function, given a trace and an index, cuts the trace at the i-th element and returns the suffix of the trace. It supposes that the given index is between 0 and the length of the trace.*

$$\begin{aligned} &\mathit{suffix} : \mathit{St}^{\vec{+}} \times \mathit{Integer} \mapsto \mathit{St}^{\vec{+}} \\ &\mathit{suffix}\ (\sigma_0 \rightarrow \cdots \rightarrow \sigma_j, i) = \begin{cases} \sigma_i \rightarrow \cdots \rightarrow \sigma_j & \text{if } i \geq 0 \wedge i < j \\ \epsilon & \text{if } i = j \end{cases} \end{aligned}$$

Definition 4 (*project* function). *The project function, given a location, a trace, a set of owned synchronisable elements, and the thread that is currently analysed, returns the set of visible values following the happens-before consistency in the given trace.*

$$\begin{aligned} &\mathit{project} : \mathit{Loc} \times \mathit{St}^{\vec{+}} \times \wp(\mathit{Sync}) \mapsto \wp(\mathit{Val}) \\ &\mathit{project}\ (l, \sigma_0 \rightarrow \cdots \rightarrow \sigma_i, S) = \{v : \exists j \in [0..i] : \mathit{action}(\sigma_j) = (w, l, v) \wedge \\ &\qquad\qquad \mathit{not\_synchronised}(\sigma_j \rightarrow \cdots \rightarrow \sigma_i, S)\} \end{aligned}$$

The first part of the condition of sh_j, i.e.,

$$\mathit{action}(\sigma_j) = (\mathtt{w}, l, v)$$

excludes the transitions that do not write on the shared memory, and the second part, i.e.,

$$\mathit{not\_synchronised}(\sigma_j \rightarrow \cdots \rightarrow \sigma_i, S)$$

the ones whose values are overwritten by a successive action following the happens-before order (rule OW).

Definition 5 (*not_synchronised* function). *The not_synchronised function, given a trace and a set of synchronisable elements, returns* `true` *if and only if the first state of the trace is not synchronised on an element in the given set (case (1)), or if there is no write action that writes on the same location of the first action of the given trace and that is synchronised-with it (case (2)).*

$$\begin{aligned} &\mathit{not\_synchronised} : \mathit{St}^{\vec{+}} \times \wp(\mathit{Sync}) \mapsto \{\mathtt{true}, \mathtt{false}\} \\ &\mathit{not\_synchronised}(\sigma_0 \rightarrow \cdots \rightarrow \sigma_i, S) = \mathtt{true} \text{ if and only if} \\ &(1) S \cap \mathit{synchronised}(\sigma_0) = \emptyset \vee \\ &(2) \nexists \sigma_j \in \mathit{cut}(\sigma_0 \rightarrow \cdots \rightarrow \sigma_i, S) : \mathit{action}(\sigma_j) = (w, l, v), \mathit{action}(\sigma_0) = (w, l_0, v_0), \\ &\quad l = l_0 \end{aligned}$$

Definition 6 (*cut* function). *The cut function, given a trace and a set of synchronisable elements, returns the trace cut to the first states that are all synchronised-with at least one of the given elements.*

$$\begin{aligned} &\mathit{cut} : \mathit{St}^{\vec{+}} \times \wp(\mathit{Sync}) \mapsto \mathit{St}^{\vec{+}} \\ &\mathit{cut}\ (\sigma_0 \rightarrow \cdots \rightarrow \sigma_i, S) = \begin{cases} \epsilon \text{ if } \mathit{synchronised}(\sigma_0) \cap S = \emptyset \\ \sigma_0 \rightarrow \mathit{cut}(\sigma_1 \rightarrow \cdots \rightarrow \sigma_i, S) \text{ otherwise} \end{cases} \end{aligned}$$

3.4 Fixpoint Semantics

Through the *step* function we define the fixpoint concrete semantics in order to compute all the possible finite traces of a given multithreaded program.

Single-Thread Semantics. Given a thread and an element of the thread-partitioned domain, this semantics returns the traces of its possible partial finite executions, following the happens-before memory model, when the parallel executions of other threads are the ones represented by the given element of the thread-partitioned domain. It is the basic step that will be used to define the multithreaded semantics. This approach is classical in literature, as for instance the example 7.2.0.6.3 of [5].

Definition 7 ($\mathbb{S}^\circ$)

$$\mathbb{S}^\circ : \Psi \times \Omega \times TId \mapsto \wp(St^{\vec{+}})$$
$$\mathbb{S}^\circ\ [\![f, r, t]\!] = lfp^{\subseteq}_{\emptyset} \lambda T.\{\sigma_0\} \cup \{\ \sigma_0 \to \cdots \to \sigma_{i-1} \to \sigma_i : \sigma_0 \to \cdots \to \sigma_{i-1} \in T \wedge \sigma_i \in step(t, f, r)\}$$

Multithreaded Semantics. The multithreaded fixpoint semantics computes all the possible executions of a multithreaded program following the happens-before memory model.

It starts from an element of the thread-partitioned domain that relates each thread that is active at the beginning of the computation to an empty trace ϵ ($f_0 = \{[t \mapsto \epsilon : t \text{ is the identifier of an active thread}]\}$), and in the second component each active thread to $\bot_\Omega$, where

$$r_0 = \{[t \mapsto \bot_\Omega : t \text{ is the identifier of an active thread}]\}.$$

At each iteration it computes the semantics using the traces of the execution of different threads obtained at the previous step. The set of finite traces is restricted only to the traces that end with a blocking state; it is necessary in order to compute which values are visible through the *not_synchronised* function. In particular we want to discard all the elements that are overwritten during a set of transitions all synchronised-with the analysed read action; to do that, we need to consider only the traces that are complete, i.e. that end with a blocking state.

Definition 8 ($\mathbb{S}^{\|}$)

$$\mathbb{S}^{\|} : \Psi \times \Omega \mapsto \wp(\Psi \times \Omega)$$
$$\mathbb{S}^{\|}\ [\![f_0, r_0]\!] = lfp^{\subseteq}_{\emptyset} \lambda\Phi.\{(f_0, r_0)\} \cup \{(f_i, r) : \exists (f_{i-1}, r) \in \Phi : \forall t \in dom(f_{i-1}) : f_i(t) \in S^\circ[\![f_{i-1}, r, t]\!], f_i(t) = \sigma_0 \to \cdots \to \sigma_i, \sigma_i \in St_{\to}^{\circ}\}$$

The intuition behinds this fixpoint definition is the following:

- at the first iteration it computes the complete semantics of each thread "in isolation" since the trace of the other threads is empty, and then the *step* function performs a step using the last state of the given thread and following the sequential consistency;

– at the second (or i-th) iteration it computes the complete semantics of each thread in which the visible values have been modified at most one (or i-1) times by other threads.

For instance to compute the multithreaded semantics of the following example when `x`, `y`, and `z` are equal to 0 at the beginning of computation:

Thread 1	Thread 2	Thread 3
`y=z;`	`z=x;`	`x=1;`

Informally, at the first iteration we obtained that in thread 1 `j=0`, in thread 2 `z=0`, and in thread 3 `x=1`.

At the second iteration we still obtain that in thread 1 `j=0` and in thread 3 `x=1`, while in thread 2 we may write the value 0 or 1 (as it may see the write action performed by thread 3 in the previous iteration) to variable `z`.

During the third iteration, thread 1 may write the value 0 or 1 to the variable `y`, as it may or may not see the value written by thread 2. The other two threads behave as in the previous iteration. Moreover we have reached a fixpoint and our computation ends.

In this simple example it is clear that we need to compute a fixpoint between the semantics of different threads in order to propagate the values written and read by threads. In a more complex situation, as for instance when a value written by a parallel thread may change the control flow of the thread, this interaction may be repeated many times, requiring a fixpoint computation.

3.5 Launching a Thread

The *step* function is not in position to launch a new thread, as it operates only intra-thread steps. So the multithreaded semantics must be extended to support this action. Since we are generic with respect to the programming language, we do not present the details; on the other hand, it is important to define it in order to make evidence of how the relations between threads are traced by the second component of the multithreaded domain.

In this context, we suppose that a function $launch : St \mapsto (TId \times St \times St) \cup \bot_l$ is provided; given a state, if its next action is the launch of a thread, it returns the identifier of the new thread, its initial state and the next state of the execution. Informally, the computational multithreaded step may be defined in the following way, where (f, r) is the previous state:

$$(f', r') : t \in dom(f), f(t) = \sigma_0 \to \cdots \to \sigma_i, launch(\sigma_i) = (t', \sigma'_0, \sigma_{i+1}),$$
$$f' = f[t \mapsto (\sigma_0 \to \cdots \to \sigma_i \to \sigma_{i+1}), t' \mapsto (\sigma'_0)], r' = r[t' \mapsto (t, i)]$$

3.6 The Running Example

We apply all these definitions to the example presented in section 1.1. We focus only on the analysis of the condition of thread 2.

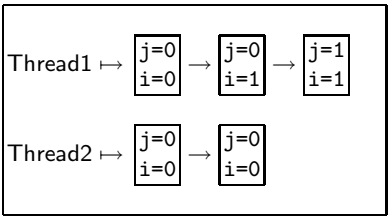

Fig. 3. The result of the first iteration of the multithreaded semantics computation

The result obtained by the first iteration of the computation of $\mathbb{S}^{\|}$ is depicted by figure 3 (we represent only the state of shared memory, ignoring the control state and the private memory). Note that, since the choice of which shared memory is visible is deterministic, it is composed only of an element of the concrete domain.

Which are the states of the shared memory returned by the *visible* function when we are evaluating the condition of thread 2 at the second iteration? In order to compute them, we need to consider which values are returned by the *project* function. We ignore the first use of this function ($project(suffix(f(t')), i)$, where t' is the thread that launched the current one), as we suppose there were two parallel threads at the beginning of the execution, and so that both are launched by the system. In the second case, we use the *project* function only with the execution trace of thread 1, as it is the only thread in the domain of our multithreaded state that is not the current thread. Since there is no synchronisation action, S is empty. In this situation, the read actions of variables `j` and `i` are both able to see the values 0 and 1, as 0 is sequentially consistent, while 1 has been written by thread 1 and may be returned by the *visible* function.

Finally, we are in position to check if, in this situation, the condition may be evaluated to `true`. The condition to be evaluated is $j == 1 \&\& i == 0$. If the read action on `i` sees the value sequentially consistent, and the one on `j` the value written by the second instruction of thread 1 (and returned by the *visible* function), the condition would be evaluated to `true`. This behaviour is sound w.r.t. the happens-before memory model, as pointed out by section 2.2.

4 Multithreaded Abstract Semantics

In order to develop a static analysis via abstract interpretation [4,5], we define the abstract semantics aimed at computing an approximation of the concrete one.

4.1 Required Elements

As we have done for the concrete semantics, in order to be generic with respect to the programming language we need some sets and functions.

In particular, the required sets are the following, with the same meaning as the ones introduced by the concrete semantics, but applied to abstract elements: $\overline{TId}$, $\overline{Sh}$, $\overline{Loc}$, $\overline{Sync}$, and $\overline{St}$. Moreover, we suppose that $\overline{Sh} : \overline{Loc} \mapsto \overline{Val}$.

In the same way the function $\overset{\circ}{\rightarrow}^{\#}$: $\overline{St} \times \overline{St} \mapsto \{\mathtt{true}, \mathtt{false}\}$ defines the abstract single step behaviours of the computation.

About the functions, we require the following, with the same meaning as the concrete semantics: $shared^{\#} : \overline{St} \mapsto \overline{Sh}$, $action^{\#} : \overline{St} \mapsto \overline{\bot}_a \cup (\{\mathtt{r}, \mathtt{w}\} \times \overline{Loc} \times \overline{Val})$, $synchronised^{\#} : \overline{St} \mapsto \wp(\overline{Sync})$, and $set\_shared^{\#} : \overline{St} \times \overline{Sh} \to \overline{St}$.

4.2 Trace Partitioned Abstract Domain

The abstract domain is similar to the concrete one: the only difference is that it deals with abstract sets, while the meaning is exactly the same.

$$\overline{\Psi} : \overline{TId} \to \overline{St}^{\vec{+}}$$
$$\overline{\Omega} : \overline{TId} \to ((\overline{TId} \times Integer) \cup \overline{\bot}_{\Omega})$$

4.3 Upper Bound Operator and Abstraction Function

We define the upper bound operator and the abstraction function on the domain just presented. We need the upper bound operator between two single-thread states ($\sqcup_{St}$) and between two values ($\sqcup_{Val}$), the abstraction functions $\alpha'_{ST} : St \mapsto \overline{St}$ (that given a concrete single-thread state returns its abstraction), and $\alpha'_{Tid} : TId \mapsto \overline{TId}$ (that given a concrete thread identifier returns its abstraction), are provided.

Note that in these definitions (and in the soundness proofs) we focus only on the first component of the domain (Ψ), as the second part just traces some relations between threads, and it can be linearly abstracted applying the α'_{TId} function.

Definition 9 (Upper bound operator on $\overline{\Psi}$)

$$\begin{aligned}
&\sqcup_f : && \overline{\Psi} \times \overline{\Psi} \mapsto \overline{\Psi} \\
&\overline{f}_1 \sqcup_f \overline{f}_2 = && \{[\overline{t} \mapsto \overline{\tau}] : \overline{t} \in dom(\overline{f}_1) \cup dom(\overline{f}_2), \\
& && \overline{\tau} = \begin{cases} \overline{f}_1(\overline{t}) \sqcup_\tau \overline{f}_2(\overline{t}) & \text{if } \overline{t} \in dom(\overline{f}_1) \cap dom(\overline{f}_2) \\ \overline{f}_1(\overline{t}) & \text{if } \overline{t} \in dom(\overline{f}_1) \setminus dom(\overline{f}_2) \\ \overline{f}_2(\overline{t}) & \text{if } \overline{t} \in dom(\overline{f}_2) \setminus dom(\overline{f}_1) \end{cases} \}
\end{aligned}$$

Definition 10 (Upper bound operator on $\overline{St}^{\vec{+}}$)

$$\begin{aligned}
&\sqcup_\tau : \overline{St}^{\vec{+}} \times \overline{St}^{\vec{+}} \mapsto \overline{St}^{\vec{+}} \\
&(\overline{\sigma}_0 \to \cdots \to \overline{\sigma}_j) \sqcup_\tau (\overline{\sigma}'_0 \to \cdots \to \overline{\sigma}'_i) = \\
&\qquad = (\overline{\sigma}_0 \sqcup_{St} \overline{\sigma}'_0) \to \cdots \to (\overline{\sigma}_j \sqcup_{St} \overline{\sigma}'_j) \to (\overline{\sigma}'_{j+1}) \to (\overline{\sigma}'_i)
\end{aligned}$$

supposing that $j \leq i$. Otherwise, $\sqcup_{St}$ is commutative and it is sufficient to commute the elements.

Definition 11 (Abstraction function of $\wp(\Psi)$)

$$\alpha_f : \wp(\Psi) \mapsto \overline{\Psi}$$
$$\alpha_f(\Phi) = \bigsqcup_{f \in \Phi}^{f} \alpha'_f(f)$$

$$\alpha'_f : \Psi \mapsto \overline{\Psi}$$
$$\alpha'_f(f) = \{[\overline{t} \mapsto \overline{\tau}] : \exists t \in dom(f) : \overline{t} = \alpha'_{TId}(t) \wedge \overline{\tau} = \alpha'_\tau(f(t))\}$$

Definition 12 (Abstraction function of $\wp(St^{\vec{+}})$)

$$\alpha_\tau : \wp(St^{\vec{+}}) \mapsto \overline{St}^{\vec{+}}$$
$$\alpha_\tau(T) = \bigsqcup_{\tau \in T}^{\tau} \alpha'_\tau(\tau)$$

$$\alpha'_\tau : St^{\vec{+}} \mapsto \overline{St}^{\vec{+}}$$
$$\alpha'_\tau(\sigma_0 \to \cdots \to \sigma_i) = \alpha'_{ST}(\sigma_0) \to \cdots \to \alpha'_{ST}(\sigma_i)$$

4.4 *step*$^\#$ Function

The *step*$^\#$ function is quite similar to the concrete one. If the action is not a read it just performs the step through the $\overset{\circ}{\to}^\#$ function. Otherwise it computes the step injecting into the read value the least upper bound of all the values returned by the *visible*$^\#$ function and of the sequential consistent value. The *visible*$^\#$ function is obtained as the abstraction of the *visible* function.

Definition 13 (*step*$^\#$ function)

$$step^\# : \overline{TId} \times \overline{\Psi} \times \overline{\Omega} \mapsto \overline{St}$$
$$step^\# (\overline{t}, \overline{f}, \overline{s}) = \sigma \text{ where } \overline{f}(\overline{t}) = \overline{\sigma}_0 \to \cdots \to \overline{\sigma}_i \text{ and}$$
$$\overline{\sigma}_i \overset{\circ}{\to}^\# \overline{\sigma} \qquad \text{if } \pi_1(action^\#(\overline{\sigma}_i)) \neq \boldsymbol{r}$$

$$\overline{\sigma}'_i \overset{\circ}{\to}^\# \overline{\sigma} : \qquad \text{if } \pi_1(action^\#(\overline{\sigma}_i)) = \boldsymbol{r}$$
$$\overline{V} = visible^\#(\overline{t}, \pi_2(action^\#(\overline{\sigma}_i)), synchronised^\#(\overline{\sigma}_i), \overline{f}, \overline{s}(\overline{t}))$$
$$\overline{v} = \bigsqcup_{\overline{v'} \in \overline{V}}^{Val} \overline{v'}$$
$$\overline{sh} = shared^\#(\overline{\sigma}_i), \overline{sh}' = \overline{sh}[\overline{l} \mapsto \overline{v} \sqcup_{Val} \overline{sh}(\overline{l})]$$
$$\overline{\sigma}'_i = set\_shared^\#(\overline{\sigma}, \overline{sh}')$$

In this definition, we focused on a situation in which the abstract domain for primitive values is non-relational; in this way, we do not support a relational domain as for instance octagons [15].

4.5 Fixpoint Semantics

We proceed as in section 3.4: we define the single trace semantics in fixpoint form basing it on the $step^{\#}$ function just presented, and then we present the multithreaded semantics.

Definition 14 (Single-thread semantics $\overline{\mathbb{S}}^{\circ}$)

$$\overline{\mathbb{S}}^{\circ} : (\overline{\Psi} \times \overline{\Omega} \times \overline{TId}) \mapsto \overline{St}^{\vec{+}}$$
$$\overline{\mathbb{S}}^{\circ}\ [\![\overline{f}, \overline{r}, \overline{t}]\!] = lfp^{\sqsubseteq}_{\emptyset} \lambda\overline{\tau}.\{\overline{\sigma}_0\} \sqcup_{\tau} \{\overline{\sigma}_0 \rightarrow \cdots \rightarrow \overline{\sigma}_{i-1} \rightarrow \overline{\sigma}_i : \overline{\sigma}_0 \rightarrow \cdots \rightarrow \overline{\sigma}_{i-1} = \overline{\tau} \wedge \overline{\sigma}_i = step^{\#}(\overline{t}, \overline{f}, \overline{r})\}$$

Definition 15 (Multithreaded semantics $\overline{\mathbb{S}}^{\|}$)

$$\overline{\mathbb{S}}^{\|} : \overline{\Psi} \times \overline{\Omega} \mapsto \overline{\Psi} \times \overline{\Omega}$$
$$\overline{\mathbb{S}}^{\|}\ [\![\overline{f}_0, \overline{r}_0]\!] = lfp^{\sqsubseteq}_{\emptyset} \lambda(\overline{f}, \overline{r}).\{(\overline{f}_0, \overline{r}_0)\} \sqcup_f \{(\overline{f}_i, \overline{r}) : \forall \overline{t} \in dom(\overline{f}) : \overline{f}_i(\overline{t}) = \overline{\mathbb{S}}^{\circ}[\![\overline{f}, \overline{t}]\!]\}$$

The intuition of these definitions is exactly the same of the concrete semantics: $\overline{\mathbb{S}}^{\circ}$ computes the semantics of a single thread given a multithreaded state (from which the $step^{\#}$ function extrapolates the visible values of the shared memory through the $visible^{\#}$ function), while $\overline{\mathbb{S}}^{\|}$ iterates this computation using the previous multithreaded state for each thread until a fixpoint is reached.

The definition of the multithreaded semantics may be straightforwardly extended in order to support widening and narrowing operators [4], which are required to guarantee the convergence of the analysis when the abstract domain is of infinite height.

Theorem 1 (Soundness of $\mathbb{S}^{\|}$). *The multithreaded semantics is sound, i.e. let $\overline{\Psi}_{pre}$ be the set of all the prefixpoints of $\overline{F}^{\|}$, then $\forall \overline{f} \in \overline{\Psi}_{pre} : \alpha_f(\mathbb{S}^{\|})[\![\overline{f}]\!] \sqsubseteq_f \overline{\mathbb{S}}^{\|}[\![\overline{f}]\!]$.*

4.6 Launching a Thread

The launch of a thread may be abstracted from the composition of the concrete definition presented by section 3.5 with the abstraction function.

4.7 Complexity

The proposed analysis requires the computation of two nested fixpoints. The complexity of this approach might appear too heavy in order to apply it to real programs, as the computation of a fixpoint is known to be an expensive operation. On the other hand, the multithreaded semantics may execute in parallel the single-thread semantics of different active threads; supposing that there is a number of processors at least equal to the number of active threads (looking at the current trend of the CPU market, with the appearance of multicore architectures, this supposition is not unreasonable), the complexity of each iteration of $\overline{\mathbb{S}}^{\|}$ corresponds to the most expensive computation of the active threads'

semantics. In addition, there are interesting results in optimising the fixpoint computation [14].

A preliminary implementation of two nested fixpoints has been presented by [8]; note that this work does not implement the happens-before memory model, but it relies on two nested fixpoints in order to compute a sound approximation of a multithreaded program. Even if the computation of the fixpoint is sequential, the experimental results are quite promising: the analysis of 24 threads and more than 2.000 bytecode statements requires 1'07". Moreover, the execution time seems to grow linearly w.r.t. the number of analysed threads and statements.

In this context, our approach may be able to scale up; indeed, model checkers seem to be inadequate to analyse multithreaded programs because of the state explosion problem, which is particularly relevant when dealing with all the possible interleavings of threads' executions. In addition, partial order reduction techniques [17] do not improve significantly their performance when there are many interactions between threads.

Finally, since our approach is parametrised on the abstract intra-thread domain and semantics, we can also tune them in order to obtain a faster but less precise or a slower but more precise analysis.

4.8 The Running Example

We analyse the running example presented in section 1.1 supposing that we use the interval domain in order to catch information about integer variables.

At the first iteration we obtain the same results as the concrete semantics, completely described in section 3.6. The only difference is that now we deal with abstract values, and so we relate each integer variable to an interval value instead of an integer.

Then we analyse which abstract values are returned by the $visible^{\#}$ function when reading `i` and `j`. In particular, both the initial values (j and i equal to 0) and the values written by thread 1 (1 written both on j and on i) are visible. For both the variables the least upper bound of these elements returns the interval [0..1]. Then the condition of thread 2 ($j == 1 \&\& i == 0$) may be evaluated to `true`, and we conclude that the exception may be thrown. This result is sound w.r.t. the concrete semantics, and so to the happens-before memory model.

5 Related Works

Many approaches have been developed in order to statically analyse multithreaded programs; most of them deal with deadlock and data race detection [20]. In the last few years other approaches, analysing other and more generic properties, have been proposed [22,3,25,7]. Usually these approaches suppose that the execution is sequentially consistent, but this assumption is not legal under, for instance, the Java Memory Model.

[21] presents a semantics for Java multithreaded programs that respects an earlier version of the Java Memory Model. In particular it presents an executable

semantics that is sound and complete with respect to the Java Memory Model, and it verifies programs on it through model checking techniques. It is specific for the Java programming language, it deals with the memory model and also the semantics of the programming language, and it is affected by the state space explosion problem.

In a similar way, [9] develops a model checker sensitive to the .NET memory model [6]. It is specific for the C# language, and also in this case the experimental results show the effects of the state explosion problem.

[2] proposes a formalisation of the Java Memory Model through a semantics that combines the operational, denotational, and axiomatic approaches. It builds up a subset of the legal executions under the Java Memory Model. In this way this approach is similar to ours, since in order to obtain a sound static analysis we compute a superset of these executions. However it is specific to the Java programming language, and it does not propose any static analysis.

[23] presents a framework in order to formalise and study a memory model. This approach is generic w.r.t. the programming language, and it allows the comparison of different memory models. Indeed, it does not propose any static analysis.

In this research context, as far as we know our work appears to be the first one that combines a generic definition of a memory model and its static analysis.

6 Conclusion and Future Work

In this paper we present the formalisation of the happens-before memory model in a fixpoint form, and we build on top of it an abstraction in order to statically analyse a program w.r.t. this memory model. It is completely generic both to the programming language and to the analysed property, as it is parameterized by the concrete and abstract single-thread state, and semantics. In this way we completely split the formalisation of the memory model from the programming language. Moreover, at the abstract level the core of the happens-before memory model (i.e. the $visible^{\#}$ function) is obtained by linearly abstracting its concrete definition. In this way, we are in position to automatically build up a static analysis starting from the concrete specification of the core of a memory model.

Our approach may be easily applied to define and analyse other memory models, and also as an unifying framework in order to compare them: if we prove that a memory model is an abstraction of another one, we prove that all the analyses developed on the second one are sound in the first analysis.

We think that the idea of separating the memory model from the programming language on which it is applied is very promising, as it allows the reuse of analyses on different languages with different memory models. Until now, the only memory model that appears to have been studied deeply is the Java Memory Model, but it is not unrealistic to suppose that in the near future different models, like [6], will appear, to which our approach seems to be easily extendible.

On the other hand, the idea of reusing a memory model for a different programming language appears already near to reality. Behind the specification of

a memory model there often is much work (sometimes many years of deep study, in order to understand the problems and how solve them), and so its reuse is not only a possibility, but sometimes a need. For instance, there is a draft [1] that depicts how to apply the Java Memory Model to the C++ programming language.

6.1 Future Works

Our aim is to develop a static analysis on Java bytecode for multithreaded programs. In this way, we are going to define the concrete and abstract intra-thread domain and semantics of this language, and some properties on it. We want also to extend our definitions in order to introduce volatile variables and the causality requirement of the Java Memory Model.

Acknowledgements. We would like to thank Mike Barnett, Agostino Cortesi, Radhia Cousot, Francesco Logozzo and the anonymous referees.

References

1. Alexandrescu, A., Boehm, H., Henney, K., Lea, D., Pugh, B.: Memory model for multithreaded c++. C++ standards committee paper WG21/N1680 (September 2004)
2. Cenciarelli, P., Knapp, A., Sibilio, E.: The java memory model: Operationally, denotationally, axiomatically. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 331–346. Springer, Heidelberg (2007)
3. Chaumette, S., Ugarte, A.: A formal model of the java multi-threading system and its validation on a known problem. In: Proceedings of IPDPS 2001, IEEE Computer Society, Los Alamitos (2001)
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of POPL 1977, pp. 238–252. ACM Press, New York (1977)
5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of POPL 1979, pp. 269–282. ACM Press, New York (1979)
6. Standard ECMA-335. Common Language Infrastructure (CLI). ECMA, 4th edn. (June 2006)
7. Farzan, A., Madhusudan, P.: Causal dataflow analysis for concurrent programs. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 102–116. Springer, Heidelberg (2007)
8. Ferrara, P.: A fast and precise analysis for data race detection. In: Proceedings of Bytecode 2008, vol. ENTCS, Elsevier, Amsterdam (2008)
9. Huynh, T.Q., Roychoudhury, A.: A memory model sensitive checker for c#. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, Springer, Heidelberg (2006)
10. Koch, G.: Discovering multi-core: extending the benefits of Moore's law. In: Technology Intel Magazine, July 2005, Intel (2005)
11. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. In: Commun. ACM, vol. 21-7, pp. 558–565. ACM Press, New York (1978)

12. Lindholm, T., Yellin, F.: Java Virtual Machine Specification. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
13. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: Proceedings of POPL 2005, pp. 378–391. ACM Press, New York (2005)
14. Méndez-Lojo, M., Navas, J., Hermenegildo, M.: Efficient, parametric fixpoint algorithm for analysis of java bytecode. In: Proceedings of Bytecode 2007, vol. ENTCS, Elsevier, Amsterdam (2007)
15. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19, 31–100 (2006)
16. Netzer, R.H.B., Miller, B.P.: What are race conditions?: Some issues and formalizations. ACM Lett. Program. Lang. Syst. 1, 74–88 (1992)
17. Peled, D.: Ten years of partial order reduction. In: Klette, R., Peleg, S., Sommer, G. (eds.) RobVis 2001. LNCS, vol. 1998, Springer, Heidelberg (2001)
18. Pugh, W.: The Java memory model is fatally flawed. Concurrency - Practice and Experience 12(6), 445–455 (2000)
19. Reynolds, J.C.: Towards a grainless semantics for shared-variable concurrency. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, Springer, Heidelberg (2004)
20. Rinard, M.C.: Analysis of multithreaded programs. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, Springer, Heidelberg (2001)
21. Roychoudhury, A., Mitra, T.: Specifying multithreaded java semantics for program verification. In: Proceedings of ICSE 2002, May 2002, ACM Press, New York (2002)
22. Ruys, T.C., Aan de Brugh, N.H.M.: Mmc: the mono model checker. In: Proceedings of Bytecode 2007, vol. ENTCS, Elsevier, Amsterdam (2007)
23. Saraswat, V.A., Jagadeesan, R., Michael, M., von Praun, C.: A theory of memory models. In: Proceedings of PPoPP 2007, pp. 161–172. ACM Press, New York (2007)
24. Sutter, H., Larus, J.: Software and the concurrency revolution. ACM Queue 3(7), 54–62 (2005)
25. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: Rigorous concurrency analysis of multithreaded programs. In: Proceedings of CSJP 2004 (2004)