A Proposal for a Semantics of a Subset of Multi-Threaded "Good" Java Programs

Eva Coscia and Gianna Reggio
Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Via Dodecaneso, 35 – Genova 16146 – Italy
{coscia,reggio}@disi.unige.it
http:\\www.disi.unige.it

1 Introduction

We have developed a formal semantics (see [1]) for a very substantial subset of the Java language as a part of a larger project that would leads to a development method for some classes of reactive concurrent systems using Java. It is worthwhile noting that, because of the classes of systems addressed by our method, we are mainly interested in multi-threaded Java.

The method would cover every phase of the development: from a requirement description to a Java encoding. This last phase should be, given a detailed design specification, automatic or, at least, quasi-automatic; this implies that we must propose "standard" implementations for the most common communication mechanisms, concurrency control strategies and so on, established in the design phase.

So, we need to give a formal semantics of Java essentially for:

- deeply understanding Java, even if w.r.t. other languages it is much easier to use and has a clear informal description (see [3]); the activity of giving a semantics to Java obviously leads us to grasp its features;
- having a formal base for checking the correctness of the final step of the method, transforming a design specification into Java code.

A structural operational semantics ([7, 4]) seems to be the correct way to have a formal but simple and natural description of Java.

In our approach, multi-threaded Java programs are modeled by a labeled transition system (lts), that is two sets describing, respectively, the intermediate interesting states of the program execution and the information exchanged with the external environment during such execution, plus a transition relation. Such relation defines triple of the form (s, l, s') where s and s' are, respectively, the initial and the final state of a program transition, while label l describes the interaction that the program has with the external world during the transition. Any transition of the model describes the execution of an atomic step in the computation.

A program has an internal structure inside which we can identify components having an independent activity. Such components are modeled by an lts too.

A set of inductive rules describe the transitions of the model and give the operational, structured semantics of the language in a simple and intuitive way.

Since it is not possible to simply associate a result with a multi-threaded Java program, it is absolutely necessary to have a correct observational semantics and, consequently, an equivalence notion to test if two programs are equivalent.

We are currently working on this equivalence notion, taking advantage of the model defined in the operational semantics (see [1]): a labeled transition system not only gives an operational semantics but also describes what information about program execution is observable from outside. The observations over a program are simply determined by the labels associated with the lts. Roughly speaking, two programs are equal if they allow the same observations.

In [1] we have given an operational semantics to multi-threaded Java (leaving aside, for the moment, mechanism for distributed computing) by following the informal specification given in [3]. This work had pointed out some questionable properties of the Java programs. Such properties concern the way concurrent threads can interact with shared objects. Java semantics allows the access to shared objects to occur in a very loose way and this sometimes has very strange effects and makes it impossible or, at least, very hard to reason about concurrent programs.

To give an idea of such properties, we want to stress that if two threads share an object and if one of them assigns one of the object's field, the second one is not guaranteed to read the newly assigned value, even if it is forced to use the field value after the first one had performed the assignment (Java gives the mechanism for coordinating the activity of concurrent threads, e.g., to force that a thread waits until another one has terminated).

So, we want to look for subclasses of Java programs having acceptable properties guaranteeing a reliable use of shared variables (in the following we will talk of shared variables instead of fields of shared objects). We have formally characterized one of such classes, based on the idea that the mechanism ensuring such properties is the use of *synchronization*, that forces the updating of the thread working memory before using shared variables and the flushing of the assigned value into the main memory.

Unfortunately, it is not possible to statically check whether a program belongs to such a subclass.

The good point, on the contrary, is that the restriction to the class of Java programs allows the use of a simpler operational semantics, and, consequently, simpler observations and equivalence notion. Out method will only consider programs in such subclass, to avoid to take care of unreliable uses of shared memory.

If we compare our semantics with the work of other authors, we can say that we essentially focus on the *dynamic* semantics of concurrent Java. In the case of pure sequential programs and for the static part of the semantics, we just apply the ideas given in [8] and other works, even if in [1] we describe an extension of the type system and of the function that enriches syntactical terms with type information, as introduced in [8], to cover the whole Java sub-language we have chosen.

Our pragmatic approach to Java semantics is also different from the one adopted in [5, 6] where the use of *event structures* allows the semantics to get rid of many details about the execution order of certain operations, while we chose to be as descriptive as possible. While in [5, 6], semantics of operations on working and main memories is given

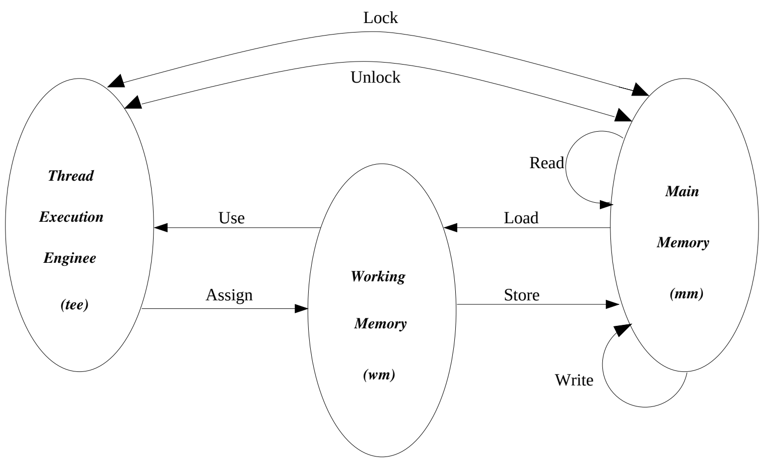


Figure 1: Working and main memory interaction

by axiomatizing the ordering rules given in [3], we explicitly describe such interaction in a detailed way, subsequently proving [1] that such rules are satisfied.

Besides, in [5, 6], the operational semantics does not give any information about what can be observed from program execution and, consequently, which could be a sensible notion of semantic equivalence between Java programs, that is essential from our point of view.

2 Questionable properties of Java programs

2.1 Main memory and working memory

Java supports concurrent programming by means of threads that independently execute code. Such code operates on Java values and objects residing in a shared *main memory*. In order to make memory access more efficient, [3] says that every thread owns a private *working memory* in which it keeps *working copies* of all the variables it uses or assigns. When a thread uses a local copy of a variable, it performs a sequence of actions, some of which, for example when it *assigns* or *uses* a value, directly operate on variables. These actions are *low level* actions that do not correspond to Java statements (in the following, *low-level actions* refers to this kind of actions). The *master copies* of variables are kept into *main memory*. Some rules (in [3]) describe when a thread is permitted to update its own local copy of a variable.

Moreover, Java implements *monitors* that allow one single thread at a time to execute protected parts of code. For that purpose, for each object and class there exists a *lock* kept into the main memory. Every thread has to compete with the other ones in order to acquire a lock.

Different actions (see fig.1) are performed by the *thread execution engine*, by the *working memory* and by the *main memory* (just *tee*, *wm* and *mm*, respectively, in the following).

A *tee* can `use` and `assign` a variable. A `use` transfers the content of a variable from the *wm* into the *tee*; an `assign` transfers a value of a variable from the *tee* into the *wm*; the *wm* can `load` a value from *mm* or `store` it into the *mm*, that, conversely, has the capability to send a value to the *wm* by a `read` action or to receive a value from *wm* by a `write` action. Every `read` by the *mm* is followed some time later by a corresponding `load` performed by the *wm*. Every `store` performed by *wm* is followed some time later by the corresponding `read` performed by the *mm*. Moreover, both *tee* and *mm* can perform (in a tightly coupled way) `lock` and `unlock` actions.

2.2 A too loose semantics

From our point of view, Java semantics is too loose w.r.t. shared memory management. Constraints ruling the asynchronous interaction in [3] between *mm* and *wm* leave a certain degree of freedom to the implementor. If, from one side, this relaxation allows designers to improve speed and throughput, by the other side "strange" and undesired behaviors can come from the free interactions of threads on shared variables.

Such a situation is described in the following **Example 1** (presented in [6]):

(t_1) $synchronized(p)\{p.y = 2;\}a = p.x; b = p.y; c = p.y;$

(t_2) $synchronized(p)\{p.y = 3; p.y = 100;\}p.x = 1;$

where t_1 and t_2 are the names of two threads executing the two pieces of code.

The key point, intuitively described by the example, is that from the moment a read operation for a value of an object field is performed by *mm*, to the moment such value is actually loaded on the thread *wm* and used, some other thread can change it.

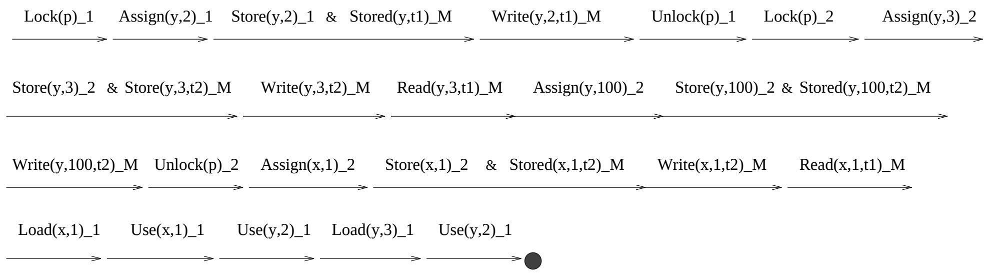
When a value is loaded into the *wm*, it could have been read by the *mm* in any moment within the past, so that it could not be the most recent one. This can sound normal in a usual concurrent programming context: if we do not ensure some ordering in the access to a field object, then it can be modified by any concurrent thread in any order and, in particular, it is possible that a thread reads a value after another one had assigned it but the *mm* has not been already changed.

What sounds a bit more strange is that if we know the execution order of the reading and writing operations on an object field, i.e., we use a "naive" coordination mechanism, for example by imposing that a thread can start only after another one has terminated the assignment execution, we can have undesired results the same.

In the previous example, even if we force the thread t_1 to execute after the thread t_2 had terminated, t_1 may read the old value 3 for y, even if t_2 assigned 100 to y before terminating. This situation is represented in fig.2, depicting one execution of the whole program. A label a_i, where $i = \{1, 2\}$ describes the occurrence of action a performed by the thread i, while a_M describes the occurrence of action a performed by *mm*. In fig.2, even if transition Load(y,3)_1 occurs after t_2 has terminated, the corresponding `read` occurrence Read(y,3,t1)_M precedes Assign(y,100)_2).

What seems worthwhile noting, from our point of view, is that the asynchrony of communication between *mm* and *wm* can cause "unexpected" and unpredictable behaviors that need to be avoided in truly concurrent programming.

If, in the previous example, t_1 assigns value 1 to a, and we are sure that t_1 executes after t_2 has completed its computation, then we reasonably expect that t_1 will assign to b and c the same value, since no other thread is modifying the value of y. From our point of view, it seems hard to predict that one possible computation assigns 100 to a but two different values to b and c.



A label l_i says that component i partecipates to the whole program transaction by performing the activity described by l

Figure 2: An execution for Example 1

Situations like the classical producer-consumer schema as well as the case when a variable is used for example as a counter, that can be updated and read by several components and must ensure that it always contains the correct values, are very frequent cases, where the "strange" executions can cause problems.

We want to stress that we are considering Java as the implementation language for systems in which concurrency is one of the most important features. Threads are heavily used to implement all the active parts of the systems, that communicate using shared memory also. However, it is hard to figure out a situation, in which the use of such a loose politic in the access of shared memory can be fruitful. In order to manage these complex systems, it is better to have stronger constraints on the implementation language that ensure a reliable communication among active components.

As we will see in the following, we aim to formally characterize a sensible subset of Java programs in which we are guaranteed to have banished computations like the one depicted in fig.2.

A very meaningful consequence, discussed in the following, is that we can use a simpler semantics for the programs in this class.

We describe the subset of good programs by means of a property, informally described below(see [1] for a formal presentation, using temporal logic). In the following `x` denotes a field of a shared object.

P1) If sometime in the past a thread tc_1 assigned a value `v` *to* `x`*, no other thread subsequently assigned a value to* `x` *and another thread tc_2 ($tc_1 \neq tc_2$) reads* `x` *from main memory, then tc_2 reads* `v`*.*

A graphical presentation of property *P1* is in fig.3.

GoodP is defined to be the set of all Java programs satisfying *P1.* It is obvious that **GoodP** do not coincide with the whole set of Java programs there exists at least one program satisfying the first part of the implication, but not the second one e.g., the program of the Ex.1. The condition $tc_1 \neq tc_2$ is justified since a thread cannot lose its most recent assignment to a variable and it is forced to write it into the *mm* before loading it again.

It is easy to prove, considering Ex.1 again, that Java programs do not satisfy even a less restrictive reformulation of the previous property requiring that if one thread is forced to read from *mm* after the other one has completed its execution, the value transmission

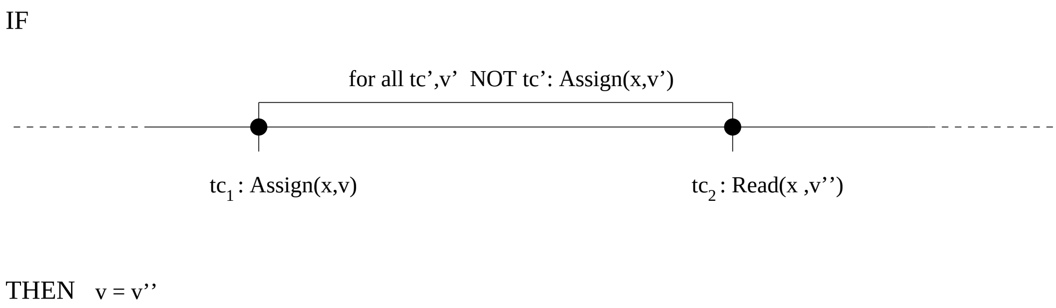


Figure 3: Property *P1*

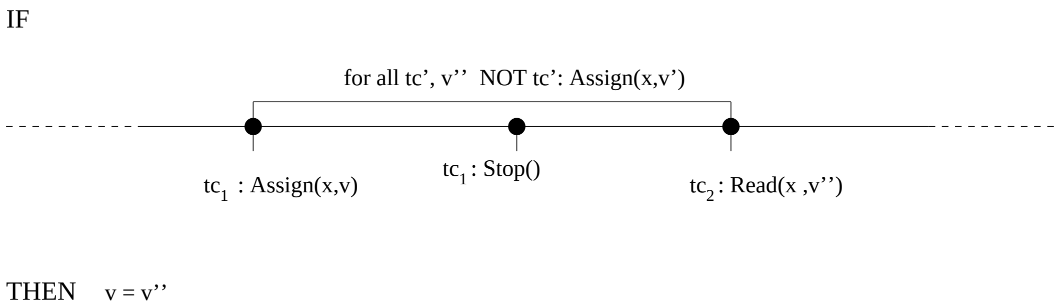


Figure 4: Property *P1'*

is correct. A formulation for this property is the following one, where we require that one thread stops before the other one can start its execution.

P1') If sometime in the past a thread tc_1 assigned a value `v` *to* `x`*, no other thread assigned subsequently a value to* `x`*, tc_1 has terminated and another thread tc_2 reads* `x`*, then tc_2 reads* `v`*.*

Again, we have a graphical presentation in fig.4.

2.3 Good programs

Now it is interesting to identify what programs, conversely, satisfy the requirement expressed by *P1* and, then, ensure a predictable use of shared memory.

Shared Objects

We make some considerations about the communication between threads by means of *shared objects*. Sadly enough, it is not so simple to determine if an object is shared by two or more threads, since it can be referred by using different variables at different times.

A thread tc shares an object o_{id} with a thread tc' if the two threads eventually in their computation read or modify the value of o_{id}. Informally we can say that:

S) tc and tc' share o_{id} iff the execution of tc eventually performs an assignment which requires to evaluate a reference to a field of o_{id} and the execution of tc' eventually performs an assignment which requires to evaluate a reference to a field of o_{id}.

Again, in [1] there is the corresponding formalization in temporal logic.

Java gives some synchronization mechanisms to avoid that threads interfere when accessing shared resources (see A). So, the easiest solution to have completely reliable communications by using shared objects is to synchronize every access (*SS* property). Moreover, it is obvious that not all the shared variables have to be used in a synchronized block, since variables that are read only or write only by all threads sharing them can be accessed without restriction. The problem is only to guarantee that a thread reads correctly a variable that could be modified by some other thread.

It is worthwhile noting that the good property *SS* above is trivially satisfied by single thread Java programsand by distributed Java applications consisting of several threads running into separate Java programs, on different hosts. This is particularly important if we consider that among the classes of systems to which our method is addressed there are distributed systems too.

Starting from these observations, the restriction that forces any access to a shared object to happen within a synchronized block, seems to be natural for having concurrent programs satisfying *P1*.

Part of our work is also to find ad hoc programming strategies that makes the use of synchronization not too expensive for performance, for example by optimizing the code in order to make the synchronized part of code as short as possible and to allow the concurrent execution of all the code not involving shared variables.

Example 2 below gives an example of a program in **GoodP** (the set of programs satisfying *P1*) obtained by Ex.1 by extending the use of synchronization to all the accesses to shared variable y (we use the short notation $\{p.y = 2;\}_l$ for $synchronized(l)\{p.y = 2;\}$):

(t_1) $\{p.y = 2;\}_{ly}\{a = p.x;\}_{lx}\{b = p.y;\}_{ly}\{c = p.y;\}_{ly}$

(t_2) $\{p.y = 3;\}_{ly}\{p.y = 100;\}_{ly}\{p.x = 1;\}_{lx}$

Informally, the proof that Ex.2 satisfies *P1* is based on two considerations: an unlock operation forces any assignment occurring inside the synchronized block to be copied in the *mm* and any read operation to occur after the locking operation.

A more detailed reasoning is: y in $\{b = p.y;\}_{ly}$ is synchronized and if $\{b = p.y;\}_{ly}$ is performed after $\{p.y = 100;\}_{ly}$, then the reading of y for t_1 that is coupled with the loading action is forced to occur after the lock on l_y by t_1 and, consequently, after the unlock on l_y of t_2 and then, after the writing of value 100 for y in *wm*. In fig. 5 we can see that if an `assign` occurs inside a block labeled by Si, Si also contains the corresponding `write`. Analogously, if a `load` occurs inside a block labeled by Si, Si contains the corresponding `read`.

We can observe that forcing any access to a shared variable to be synchronized on the same lock can be too strong a constraint.

If we only impose that an assign or a load operation on a variable referencing a shared object occurs inside a synchronized block, no matter which is the lock, we have a larger set **GoodP**$'\supset$ **GoodP**, whose elements do not satisfies property *P1* but the less restrictive *P1'*.

Example 3 is:

(t_1) $\{p.y = 2;\}_{q1}\{a = p.x;\}_{q2}\{b = p.y;\}_{q3}\{c = p.y;\}_{q4}$

(t_2) $\{p.y = 3;\}_{q5}\{p.y = 100;\}_{q6}\{p.x = 1;\}_{q7}$

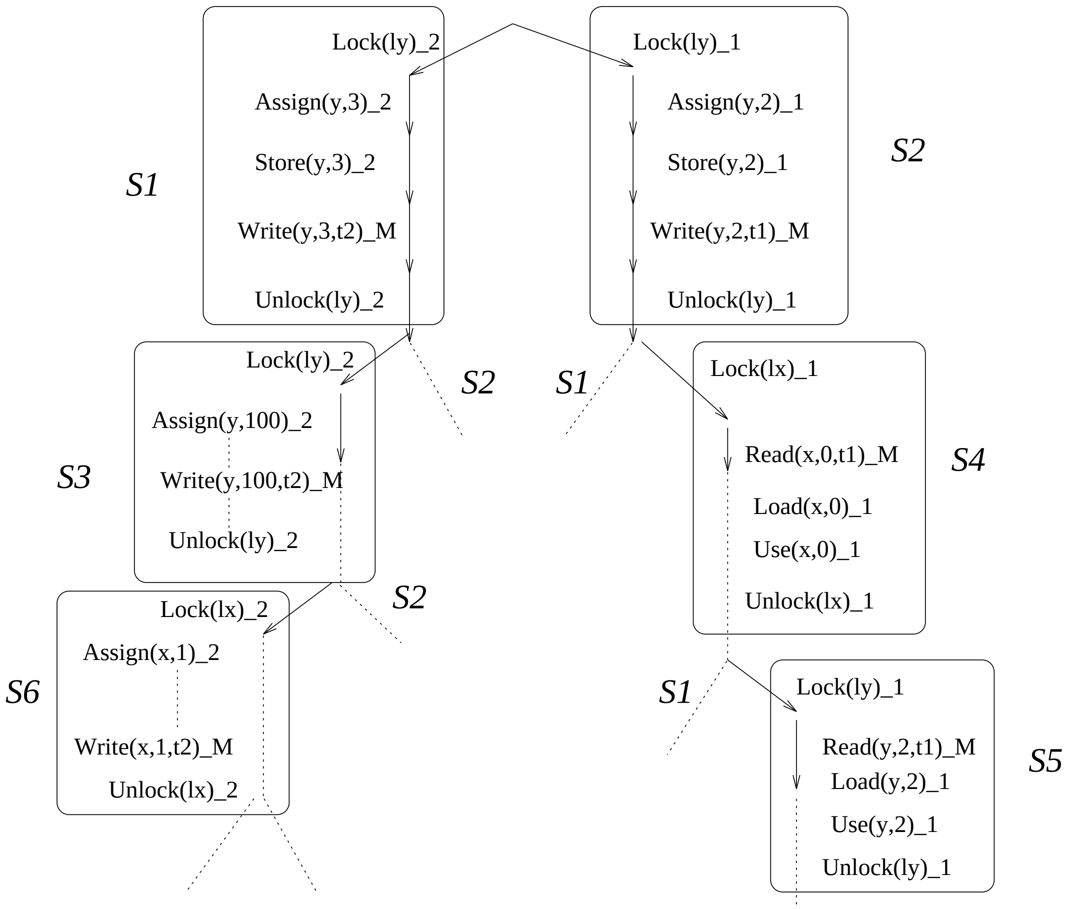


Figure 5: The execution model of Example 2

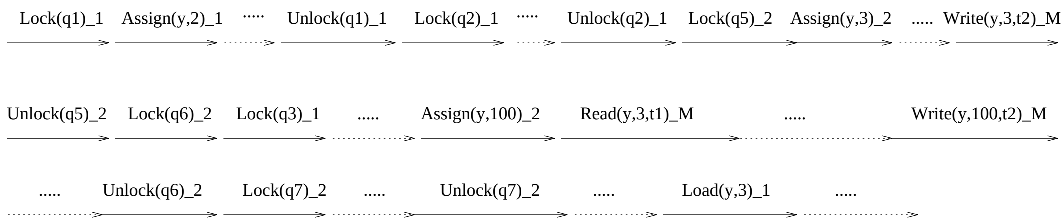


Figure 6: A computation for Ex.3 not satisfying *P1*

Fig.6 presents a computation of Ex.3 that does not satisfy *P1* since Read(y,3,t1)_M occurs after Assign(y,100)_2 but before Write(y,100,t2)_M.

2.3.1 Construction of programs in GoodP

Unfortunately, we cannot establish if a program is in **GoodP** with a static analysis, since there is no way to calculate if two variables in different threads will share, at run time, the same object field. There are only a few trivial cases of goods programs: single-threaded ones, multi-threaded ones if types of fields and methods variables of each thread are different and not assignment-compatible.

But let us stress that our work on Java semantics starts from a pragmatic point of view: we want to analyse the language and how to use it in the final implementation step within a development method. In such context it is important to develop strategies that result in "safe" implementations, preventing undesired interferences among threads.

The coding phase of our development method must guarantee to produce good programs. Part of our work will be to develop guidelines and implementation schema for most common mechanisms used in the classes of concurrent-reactive systems (e.g. communication by a channel, broadcasting, etc), proving that such implementation ensure a trustable use of shared memory, in the sense described above.

Given a shared object, there are different techniques to ensure that changes to its state occur only in consistent ways. The simplest one is to synchronize any method that modifies the object: i.e., all the object methods plus all methods containing assignments and uses of the object.

A similar technique is to create another object that contains the shared one. The outer object is fully synchronized and never returns references to the inner one. The simplest implementation is by one-slot buffers as outer objects. The slot contains the shared object that is accessible only by synchronized methods. This ensures that every thread uses that object in a secure way.

For performance considerations this technique could be unacceptable. Sometimes is better to synchronize just sections of methods or only the statements involving the shared variable. This strategy allows the other parts of code to be executed concurrently.

Again for performance considerations, if the right part of an assignment is a complex expression (i.e., the invocation of a time-consuming method) that does not contain any reference to the shared object, it is better to use a temporary variable and then synchronize just its assignment to the shared variable. For example $\{\texttt{y} = \texttt{AVeryComplexMethod}();\}_p$ becomes $\texttt{y}' = \texttt{AVeryComplexMethod}(); \{\texttt{y} = \texttt{y}'\}_p$, with y' a local variable.

3 The Operational Semantics

3.1 Generalities

We need a semantics that is simple and intuitive and gives a formal counterpart to the Java specification in [3], by using the simplest mathematical foundations and tools.

The reason is that we want to give a formal description of Java that must be easily understood and discussed by many people; we also need a formal base for proving the correctness of the coding step of the method and perhaps, as a formal support to the use of tools within the method.

Our semantics covers a wide subset of multi-threaded Java: class declarations, class and instance variables and methods, threads synchronization, I-O operations, etc. The main aspect we have not considered so far are exception handling, object serialization and remote method invocations.

Our main interest is on multi-threading in Java and we focus on the *dynamic* semantics of concurrent Java; while for *static* semantics and sequential Java, we completely refer to [8].

We have chosen an operational approach that models a Java program by a labeled transition system (*LTS*), that is a triple (*States*,*Labels*,$\rightarrow$) where *States* is the set of the meaningful intermediate states of the computations of the program, *Labels* is the set of the labels describing information exchanged with the external environment during the computation and $\rightarrow \subseteq$ *States* $\times$ *Labels* $\times$ *States* is the transition relation describing how the computation proceeds: $s \xrightarrow{l} s'$ means that the program in the state s may perform a transition with the information exchange to the external environment described by l ending in the state s'.

We need to have a very precise semantics that gives a fully detailed description of program execution. The transitions of the *LTS* model not only the execution of each single statement, but also the low level actions that are involved in the execution of complex statements. For example, entering a synchronized block implies the execution of locking operations that do not correspond to Java statements.

The choice of the information put on the labels of the system is crucial for the semantics, since it also determines what is observable from outside the program and consequently, whether two programs are equivalent. The observation of data exchange involved in input-output operations performed over files and sockets seems to be a good choice ([1]).

In a multi-threaded Java program we can identify several computation flows, each one associated with a "particular" object, that is an instance of a *thread class*. These objects have a particular `run` method whose execution runs concurrently with the other ones, so we can call these objects *active components* of the program in contrast with the *passive components*, (classes and instances of non-thread classes), that do not perform an independent activity and whose state is modified only by the actions of the active components.

Our semantics reflects this structure. The active components can be modeled by labeled transition systems. The passive ones are modeled simply by describing their states, that are, essentially, the values of their fields. Classes have a semantics too, since classes have states (given by the value of class fields) and methods, as well as objects. Moreover, an explicit description of class states is useful to record the identities of the instances created for a class.

The interaction of threads with mm (described in Sect. 2) involves the modeling of another entity (the main memory manager, MmMng) that may asynchronously read a value for a thread from an object state and, later, send it to the wm of a thread or write a value received from the thread by modifying the corresponding object: it is not just a passive component of the program and needs to be modeled by an LTS.

The state of a program is a set that results by combining the states of all the active and passive components, while the activity of the whole program can be defined by suitably combining the activities of active ones.

3.2 Semantic rules

For reasons of space, the complete semantics is in [1]; here we report only the rules related to low-level action execution. In Appendix A are reported the formal definitions of the lts' for threads, MmMng and the programs, plus the definitions of variables, functions and predicates used in the rules below.

An object state is a couple (o_{id}, m), where o_{id} is the object identifier and m memorizes the values of the object fields. A class state is a 4-tuple $(c_{id}, S_{id}, lm_i, m)$ where c_{id} is the class identifier, S_{id} the set of identifiers of previously created instances, lm_i the initial value for the fields of an instance of the class and m memorizes the values of the class fields. A thread state is a tuple $< t_{id}, \mathtt{stm}, tSt, wm, m, lmS >$ where t_{id} is the thread identifier, `stm` the code to be executed, tSt the status of the thread, lmS and wm the local and working memories and m contains the values of the thread fields. A MmMng state is a stack of elements recording some occurrences of low-level actions.

All non-meaningful components will be omitted from configurations, to make more readable the rules in the following. From now on, when the identifier and the status are omitted, we assume that their values are, respectively, t_{id} and Run; the working memory and other components will also be omitted in some states, if they are not meaningful. For example, $< \mathtt{stm}, lmS >$ stands for $< t_{id}, \mathtt{stm}, tSt, m, lmS >$.

For program states we use the notation $s|S$ for $\{s\} \cup S$ to enlighten the element s.

Inductive rules are written:

$$\frac{\phi_1 \ldots \phi_n}{\phi} \quad \textbf{if } \phi_c$$

where ϕ_c is the side condition; axioms are simply written $s \stackrel{l}{\longrightarrow} s'$ **if** ϕ_c.

Assignment of an object field

Thread rule: the assignment of a field `f` of an object o_{id}, modifies its value into the wm. It is described by the axiom below:

$< o_{id}.\mathtt{f} = \mathsf{v}, wm > \longrightarrow_T < \mathtt{skip}, wm[(o_{id}, \mathtt{f}) \to (\mathsf{v}, A)] >$

Program rule: this thread transition causes a transition of the whole program with null label (internal transition) since there is no information exchange with the external environment.

We give one inductive rule to describe that every time a thread performs a transition with null label (also, see below, when it uses a value in wm, for example), the whole program performs an internal transition.

$$\frac{tc \longrightarrow_T tc'}{tc|S \rightarrow_P tc'|S}$$

Low level actions

Here we give the rules related to low-level interactions between threads and *mm*, using names of actions as in [3]. A `use` or `assign` action is a tightly coupled interaction between a thread's execution engine and the thread's *wm*. But the transfer of data between the *mm* and a thread's *wm* is loosely coupled. When data is copied from the *mm* to a *wm*, two actions must occur: a `read` action performed by the *mm* followed some time later by a corresponding `load` action performed by the *wm*. When data is copied from the *wm* to the *mm*, two actions must occur: a `store` action performed by the *wm* followed some time later by a corresponding `write` action performed by the *mm*.

- use

Thread rule: the thread looks for the value of field `f` of o_{id} in the *wm*; the value cannot be undefined, since a `load` or an `assign` must have occurred previously.

$< o_{id}.\mathtt{f}, Run, wm > \longrightarrow_T < \mathsf{v}, Run, wm >$

if $\exists x. wm(o_{id}, \mathtt{f}) = (\mathsf{v}, x)$

- read

Main memory rule: a `read` is allowed only if there is no previously stored value for the same variable and thread that has not been written into MmMng yet;

$Mm \xrightarrow{Read(o_{id},\mathtt{f},\mathsf{v},t_{id})}_M Push(R(o_{id}, \mathtt{f}, \mathsf{v}, t_{id}), Mm)$

if $\nexists \mathsf{v}'. In(S(o_{id}, \mathtt{f}, \mathsf{v}', t_{id}), Mm)$

Program rule: a `read` adds a stack element recording the value of the object field.

$$\frac{Mm \xrightarrow{Read(o_{id},\mathtt{f},\mathsf{v},t_{id})}_M Mm'}{Mm|oc|S \rightarrow_P Mm'|oc|S}$$

if $Mem(oc)(\mathtt{f}) = \mathsf{v}$

- load

Thread rule: the `load` of a value modifies the thread *wm*; before the variable was undefined in the *wm* otherwise it had been assigned and already stored;

$< t_{id}, wm > \xrightarrow{Load(o_{id},\mathtt{f},\mathsf{v},t_{id})}_T < t_{id}, wm[(o_{id}, \mathtt{f}) \rightarrow (\mathsf{v}, L)] >$

if $\exists \mathsf{v}'. wm(o_{id}, \mathtt{f}) = (\mathsf{v}', S)$ *or* $wm(o_{id}, \mathtt{f}) = \mathtt{undef}$.

Main memory rule: a `load` that sends a value `v` for field `f` of object o_{id} into some *wm* of thread t_{id} cancels the stack elements representing previous occurrences of the corresponding read action;

$Mm \xrightarrow{Loaded(o_{id},\mathtt{f},\mathsf{v},t_{id})}_M RecCancel(R(o_{id}, \mathtt{f}, \mathsf{v}, t_{id}), Mm)$

if $In(R(o_{id}, \mathtt{f}, \mathsf{v}, t_{id}), Mm)$.

Program rule: a thread loads into its *wm* a value that MmMng has previously read; the `read` is canceled from the MmMng.

$$\frac{tc \xrightarrow{Load(o_{id},\mathtt{f},\mathsf{v},t_{id})}_T tc' \quad Mm \xrightarrow{Loaded(o_{id},\mathtt{f},\mathsf{v},t_{id})}_M Mm'}{tc|Mm|S \rightarrow_P tc'|Mm'|S}$$

- store

Thread rule: the `store` modifies the wm; it is checked that the variable has been assigned before.

$$< t_{id}, wm > \xrightarrow{Store(o_{id},\texttt{f},\mathsf{v},t_{id})}_T < t_{id}, wm[(o_{id}, \texttt{f}) \rightarrow (\mathsf{v}, S)] >$$

if $wm(o_{id}, \texttt{f}) = (\mathsf{v}, A)$.

Main memory rule: a value for a given variable and thread is stored into the stack and all the `read` actions for the same variable and thread are canceled.

$$Mm \xrightarrow{Stored(o_{id},\texttt{f},\mathsf{v},t_{id})}_M Mm''$$

if $Mm' = RecCancel(R(o_{id}, \texttt{f}, \mathsf{v}, t_{id}), Mm)$, $Mm'' = Push(S(o_{id}, \texttt{f}, \mathsf{v}, t_{id}), Mm')$

Program rule: a thread stores a value into MmMng; the `store` is added to the MmMng stack.

$$\frac{tc \xrightarrow{Store(o_{id},\texttt{f},\mathsf{v},t_{id})}_T tc' \quad Mm \xrightarrow{Stored(o_{id},\texttt{f},\mathsf{v},t_{id})}_M Mm'}{tc|Mm|S \rightarrow_P tc'|Mm'|S}$$

- write

Main memory rule: a `write` cancels the oldest store action for the same variable and thread;

$$Mm \xrightarrow{Write(o_{id},\texttt{f},\mathsf{v},t_{id})}_M Mm'$$

if $Mm' = Cancel(S(o_{id}, \texttt{f}, \mathsf{v}, t_{id}), Mm)$

Program rule: a `write` action modifies the field value of an object;

$$\frac{Mm \xrightarrow{Write(o_{id},\texttt{f},\mathsf{v},t_{id})}_T Mm'}{< o_{id}, m > |Mm|S \rightarrow_P < o_{id}, m[\texttt{f} \rightarrow \mathsf{v}] > |Mm'|S}$$

Synchronized statement

In Java there are essentially two ways to synchronize the execution of a piece of code: to declare the entire body of a method to be synchronized or to put the code into the block of a synchronized statement. For simplicity here we report only the semantics of the last mechanism.

The execution of a synchronized statement first evaluates the expression representing the lock to be taken; then the thread executing the code tries to perform a `lock` action. When it obtains the lock, it starts the execution of the statement block.

Moreover, notice that a `load` action for a variable `x`, when performed inside the synchronized lock, has a different label: the MmMng transfers to the wm a value for `x` that has been read subsequently to the `lock` action.

- synchronized

Thread rule: a synchronized statement is executed by first evaluating the expression;

$$\frac{< \mathsf{e} > \xrightarrow{l}_T < \mathsf{e}' >}{< \texttt{synchronized(e)}\{[L]\texttt{stm}\} > \xrightarrow{l}_T < \texttt{synchronized(e}'\texttt{)}\{[L]\texttt{stm}\} >}$$

We omit rules describing expression evaluation.

- synchronized load

Thread rule: a load action within a synchronized block, need to be represented by a new label.

$$\frac{< \texttt{stm} > \xrightarrow{Load(g_{id},\texttt{f},\texttt{v},t_{id})}_T < \texttt{stm}' >}{< \{\texttt{stm}\}^{g'_{id}} > \xrightarrow{SLoad(g_{id},\texttt{f},\texttt{v},t_{id})}_T < \{\texttt{stm}'\}^{g'_{id}} >}$$

Main memory rule : if a thread requires to load a value and the thread is executing a synchronized block (and, consequently, there exists a `lock` action for that thread), then the corresponding read action must have been performed after the lock action.

$$Mm \xrightarrow{SLoaded(g_{id},\texttt{f},\texttt{v},t_{id})}_T RecCancel(R(g_{id},\texttt{f},t_{id}), Mm$$

if $\exists g'_{id}, Mm_1, Mm_2, Mm_3.Mm = Mm_1 : L(t_{id}, g'_{id}) : Mm_2 R(g_{id},\texttt{f},\texttt{v},t_{id}) Mm_3$

Program rule: a thread performs a load operation while in a synchronized block and it is returned a value from wm;

$$\frac{tc \xrightarrow{SLoad(g_{id},\texttt{f},\texttt{v},t_{id})}_T tc' \;\; Mm \xrightarrow{SLoaded(g_{id},\texttt{f},\texttt{v},t_{id})}_T Mm'}{tc|Mm|S \rightarrow_P tc'|Mm'|S}$$

- lock

Thread rule: the executing thread takes the lock associated with the value of the expression and then executes the block; a new environment, local to the block, is pushed into the stack;

$$< \texttt{synchronized}(g_{id})\{[L]\texttt{stm}\}, wm, lmS > \xrightarrow{Lock(t_{id},g_{id})}_T < \{\texttt{stm}\}^{g_{id}}, \mathsf{emp\_wm}, Push(L, lmS) >$$

Main memory rule : the `lock` action is recorded. There exists no other `lock` action for the same object.

$$Mm \xrightarrow{Locked(t_{id},g_{id})}_T Push(L(t_{id}, g_{id}), Mm)$$

if $\forall t'_{id}.t'_{id} \neq t_{id} \Rightarrow \neg In(L(t'_{id}, g_{id}), Mm)$

Program rule : a thread performs a lock operation on a lock that is not assigned or that it already owns;

$$\frac{tc \xrightarrow{Lock(t_{id},g_{id})}_T tc' \;\; Mm \xrightarrow{Locked(t_{id},g_{id})}_T Mm'}{tc|Mm|S \rightarrow_P tc'|Mm'|S}$$

- unlock

Thread rule: a thread relinquishes the locks it owns;

$$< \{\texttt{skip}\}^{g_{id}}, wm > \xrightarrow{Unlock(t_{id},g_{id})}_T < \texttt{skip}, wm >$$

if $\forall g_{id}, \texttt{f}, \texttt{v}.wm(g_{id}, \texttt{f}) = (\texttt{v}, ss)$ then $ss \neq A$

Main memory rule : the corresponding `lock` action is deleted if there is no `store` action following the `lock` actions that is still in the stack.

$$Mm \xrightarrow{Unlocked(t_{id},g_{id})}_T Cancel(L(t_{id}, g_{id}), Mm)$$

if $\nexists \texttt{v}, \texttt{f}, g'_{id}, Mm_1, Mm_2.Mm = Mm_1, L(g_{id}, t_{id}), Mm_2 \wedge In(S(g'_{id}, \texttt{f}, \texttt{v}, t_{id}), Mm_2)$

Program rule : a thread performs an unlock operation; the MmMng deletes the corresponding element

$$\frac{tc \xrightarrow{Unlock(t_{id},g_{id})}_T tc' \;\; Mm \xrightarrow{Unlocked(t_{id},g_{id})}_T Mm'}{tc|Mm|S \rightarrow_P tc'|Mm'|S}$$

4 A simpler semantics

4.1 Generalities

One of the goals of our work on Java is to make the operational semantics as simple as possible. We used some simplifications on the language (i.e., the access to class variable is allowed only by using class references and so on, see [1]) and we also tried to abstract from all the aspects of the language that seems to be not relevant, i.e., not observable.

As we have seen, one of the most tricky aspects of Java semantics is to describe the interaction among threads, *wm* and *mm*. What does it happen if we abstract from the distinction among a *mm* and several *wms*, if all the interactions involved by the reading or writing of a variable are performed synchronously, if an assigned value is assumed to be written immediately into *mm*, if a variable is directly read from the *mm* every time it is used? The semantics becomes fairly simple and easy to understand.

First of all, the program state has a less complex representation since it loses the MmMng component. Thread states lose working memories and there is no need of an lts that models the memory manager. Moreover, the number of transitions both for thread and program lts is reduced, since the low level actions on memories disappear.

These considerations moved us to define a new, more abstract semantics (see [2]), called *abstract* (*Abs*) and to compare it with the original *concrete* (*Con*) described in [1].

Fig.7 shows how much simpler is the model given by *Abs* w.r.t. the one given by *Con*.

Let us consider the rules describing reading and writing of object values (see Appendix B): for both the thread's and the program's transition predicate, we have just one rule for the reading and one for the writing of a value.

Again, we briefly report the rules defining the execution of low-level actions and synchronized statements in the new semantics. See Appendix B for the complete definition of the lts' of threads and programs, while the complete definition of the semantics is in [2].

4.2 Semantic rules

Use of an object field

- get

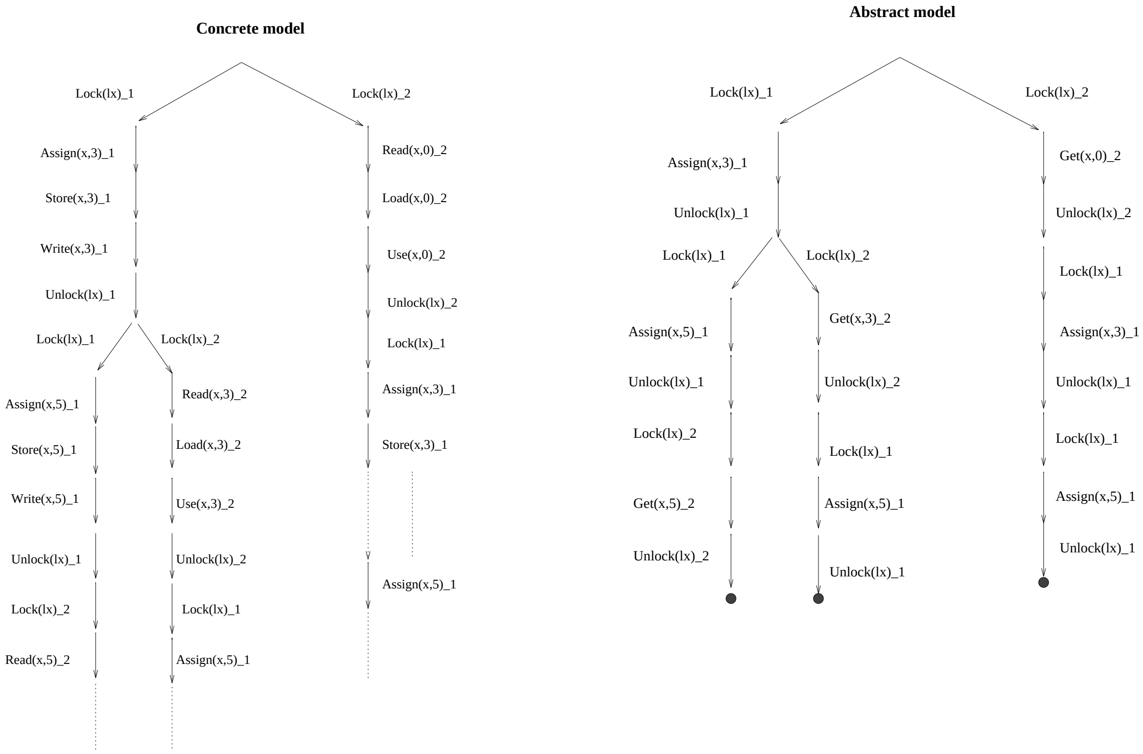
Thread rule: evaluation a field of a different object: the thread looks for its value in the corresponding class or object;

$$< o_{id}.\mathtt{f}, Run, > \xrightarrow{GetV(o_{id},\mathtt{f},\mathsf{v})}_T < \mathsf{v}, Run, >$$

Program rule: a thread accesses the instance field of a component;

$$\frac{tc \xrightarrow{GetV(o_{id},\mathtt{f},Mem(nc)(\mathtt{f}))}_T tc'}{tc|nc|S \rightarrow_P tc'|nc|S}$$

if $o_{id}(nc) = o_{id}$



Example 4: t1: $\{x = 3;\}_{lx}$ $\{x = 5;\}_{lx}$
t2: $\{a = x;\}_{lx}$

Figure 7: The concrete and the abstract models of a simple example

Assignment of an object field

- assign

Thread rule: a thread assigns a new value to an object field;

$$< t_{id}, o_{id}.\mathtt{f} = \mathsf{v}, Run > \xrightarrow{Assign(o_{id},\mathtt{f},\mathsf{v})}_T < t_{id}, \mathtt{skip}, Run, m, lmS >$$

if $o_{id} \neq t_{id}$

Program rule: an assignment modifies the field value of an object;

$$\frac{tc \xrightarrow{Assign(o_{id},\mathtt{f},\mathsf{v})}_T tc'}{tc| < o_{id}, lock, m > |S \rightarrow_P tc'| < o_{id}, lock, m[\mathtt{f} \rightarrow \mathsf{v}] > |S}$$

Synchronized statement

- lock

Thread rule: analogous to *Con* semantics;

$$< \mathtt{synchronized}(g_{id})\{[L]\mathtt{stm}\}, wm, lmS > \xrightarrow{Lock(t_{id},g_{id})}_T < \{\mathtt{stm}\}^{g_{id}}, \mathsf{emp\_wm}, Push(L, lmS) >$$

Program rule: a thread performs a lock operation on a lock that is not assigned or that it already owns;

$$\frac{tc \xrightarrow{Lock(o_{id})}_T tc'}{tc| < o_{id}, lock, m > |S \rightarrow_P tc'| < o_{id}, SetLock(tc, lock), m > |S}$$

if $LockOwner(tc, lock)$

Analogous rules for locks of threads and class objects.

- unlock

Thread rule: when the execution of the statements of a synchronized block is completed, the thread relinquishes the lock;

$$< \{\mathtt{skip}\}^{g_{id}}, wm > \xrightarrow{Unlock(t_{id},g_{id})}_T < \mathtt{skip}, wm >$$

Program rule: a thread performs an unlock operation;

$$\frac{tc \xrightarrow{Unlock(g_{id})}_T tc'}{tc| < o_{id}, lock, m > |S \longrightarrow_{\mathsf{P}} tc'| < o_{id}, DecrementLock(lock), m > |S}$$

if $LockOwner(tc, cc)$

Analogous rules for locks of threads and class objects.

Unfortunately, *Abs* cannot be considered *a semantics of Java*: it has been proved that on some examples the models associated by *Abs* and *Con* are different. An example is the one proposed also in [6]; in fact, computation depicted in fig.2 is not modeled by *Abs*.

But in the following we want to stress that, even if the two models are different, such a difference is due to the fact that the concrete semantics allows computations resulting in undesired behaviours that is reasonable, for the kind of application we consider in our work, to banish.

It is provable that, if restricted to programs that makes interaction of *wm* and *mm* not observable from outside, the *Abs* semantics is equivalent to *Con* w.r.t. the observational semantics we are investigating (see [2]).

So, if we limit our interest to class **GoodP**, *Abs* and *Con* are equivalent.

This observation is very important from our point of view. *Abs* provides a semantics that simplifies the model of programs and, by banishing a lot of internal actions (corresponding to low-level actions on memories), also eliminates transitions with no information exchange with the external environment. As a consequence, observations on models becomes simpler and the proof of equivalence among programs is easier.

It is simple to prove that *Abs* semantics satisfy properties that characterize **GoodP**. Unfortunately the two semantics do not share the same signature, so the temporal formula is slightly modified w.r.t. those for *P1*.

An equivalent reformulation of *P1* (in the *Abs* semantics a thread directly modifies the object state since there is no entity modeling the main memory manager) for the models of the abstract semantics is:

$P1^{Abs}$) *If sometime in the past a thread tc assigned a value* `v` *to a field* `f` *of an object* o_{id}*and no other thread assigned subsequently a value to* `f` *and if another thread tc" reads* `f` *in* o_{id} *state, it reads* `v`.

Just another observation. We have shown that the two semantics restricted to a meaningful subclass of programs satisfy the same properties that make more predictable the use of shared variables. But there still exists a difference in the two semantics: while rules on interactions among *mm* and *wm* do not impose the writing into *mm* the value assigned to an object if such variable will not be used again, the abstract semantics always changes the value of the object. However, this difference is not observable outside.

Intuitively, communications with the external environment (files, sockets and so on) determine the value of a program. Let's suppose that a thread assigns a value to a not shared variable, for example by {`x = 5;`}

In the model associated by *Con*, the assignment could not be written in the *mm*. The only way to observe it is to write its value outside: {`x = 5; System.out.println(x);`} but this requires a use operation that reads the *wm* and finds the newly assigned value, i.e.,`5`.

5 Conclusions

In [1] we give a formal semantics to Java in a very clear and simple way, as it has to be used within a method for system development that we hope will be used by a very heterogeneous class of people.

Investigating the way multi-threaded Java manages shared memory, we found out that some properties, that we consider necessary when reasoning over concurrent programs, are not ensured by Java programs.

Since we are persuaded that these properties cannot be disregarded in truly concurrent programming, we gave a formal characterization for subclasses of "good" (i.e., having such properties) Java programs.

A very interesting point is that, if we focus on that subclasses, we can give a simplified semantics to Java programs.

Even if, given any Java program, it is not possible to check if it belongs to these subclasses, we are interested in standard Java implementations of main mechanisms and strategies given in the design specification of concurrent-reactive systems, that produce "good" Java programs. The correctness of the implementation can be proved by using the simplified semantics.

A The Operational Concrete Semantics

In the following we formally describe the lts for the semantics of threads, MmMng and the program; the semantic rules defining the transition predicates for every lts (restricted to the execution of low-level actions and synchronized statement) are reported in Sect. 3.2, while the rules for the complete semantics are in [1].

Let us remember that a synchronized statement is : $\texttt{synchronized(}e\texttt{)}\,B$ where e is the expression denoting the object whose lock must be acquired by a thread before executing the statements in block B.

Moreover, we leave apart class fields and type of fields. For the complete semantics, see [1].

Semantic sets
We define the following syntactic and semantics sets for the metavariable we will use in the type system and in the following:

- FIELD: is the set of field names; ranged over by metavariable `f`;
- $\mathsf{STATEMENT}$ is the set of Java$_1$ statements as defined in the appendix; ranged over by metavariable `stm`; `skip`; `stm` and `stm`; `skip` are simply denoted by `stm`; {`skip`} is abbreviated in `skip`
- $\mathsf{INTEGER}$ is the set of integer values; ranged over by metavariable i;
- $\mathsf{BOOL}=$ {true,false} is the set of boolean values; ranged over by metavariable b;
- CHAR is the set of char values; ranged over by metavariable c;
- $\mathsf{PRIMITIVE} = \mathsf{INTEGER} \cup \mathsf{BOOL} \cup \mathsf{CHAR}$ is the set of primitive type values; ranged over by metavariable pv;

We use the following sets of semantic values for (active and passive) components identifiers:

- $\mathsf{TH\_ids}$: is the set of thread *identifiers*; ranged over by metavariable t_{id};
- $\mathsf{OB\_ids}$: is the set of (non-thread) object identifier; ranged over by the metavariable o_{id};
- $\mathsf{COMP\_ids} = \mathsf{TH\_ids} \cup \mathsf{OB\_ids} \cup \mathsf{CL\_ids}$: is the set of generic component identifiers; ranged over by the metavariable g_{id};
- $\mathsf{NonCL\_ids} = \mathsf{TH\_ids} \cup \mathsf{OB\_ids}$: is the set of non-class identifiers; ranged over by the metavariable nc_{id};

Finally, the set representing expression values is:
$\mathsf{VALUE} = \mathsf{COMP\_ids} \cup \mathsf{PRIMITIVE} \cup \{\texttt{null}\}$
It is ranged over by metavariable v. `null` denotes a reference to a special "null" object.

A.1 Thread Semantics

The semantics of threads is given by the lts T_LTS=(T_states, T_labels, $\longrightarrow_T$).

The state of a thread in each instant, during program execution, can be modeled by the code it has to execute and by the value of its local and global variables. Furthermore, it is necessary to know the *status* of the thread, that can be started, stopped or suspended. Each thread state contains information to describe an intermediate state: the identifier and the class, the code to be executed, the values of local variables, the contents of its working memory and the values of its instance fields.

T_states $=$ TH_ids $\times$ STATEMENT $\times$ Status $\times$ W_MEMORIES $\times$ MEMORIES $\times$ Stacks, where :

- W_MEMORIES= (COMP_ids $\times$ FIELD) $\rightarrow_{\mathcal{T}}$ Wm_Values; [1]
  Wm_Values $= \{$`undef`$\} \cup$ (VALUE $\times$ Ss) where $Ss = \{A, S, L\}$ is the set of elements returned by evaluating a variable in the local working memory; each element contains the value of the variable and the status of the value, that could be L if the variable has been loaded but not stored, A if it has been assigned and not still stored, S if it has been stored;

  emp_wm, denoting the empty working memory, is the constant function always returning `undef`.

- MEMORIES= FIELD $\rightarrow_{\mathcal{P}}$ VALUE; by $m[$`f` $\rightarrow$ v$]$ it is denoted the updating of m that associates v to `f` and behaves as m on the other elements.

- Stacks is the set of stacks of local memories, where a local memory is an element of the set Locals and denotes a partial function lm : IDENTIFIER $\rightarrow$ VALUE; ranged over by metavariable lmS; em$_{\text{l}}$mS denotes the empty stack of local memories.

- Status $= \{New, PreStop, Run, Stop, Susp\}$ is the set of all possible status a thread can assume:

  - *New*: not yet started; it has been created but not started by dome other thread.
  - *PreStop*: stopped but not started; it has been stopped while it was waiting to be started.
  - *Run*: runnable; it has already been started and it is executing its run method.
  - *Stop*: it has terminated the execution of its run method or another thread stopped it;
  - *Susp*: suspended; it has been suspended by some other thread while it was executing; it will remain in such state until another thread resume it.

T_labels $=$ LowLevelLabs $\cup$ SynchronizationLabs

Here we just consider labels describing low level actions (LowLevelLabs) and synchronized statement (SynchronizationLabs).

The labels are n-tuples, but we write $label(x_1, \ldots, x_n)$ instead of $(label, x_1, \ldots, x_n)$.

We briefly analyse every label (in the following $VarArgs \equiv$ COMP_ids $\times$ FIELD):

[1]In the following, we use $f : D \rightarrow_{\mathcal{T}} C$ to denote a total function from D in C, and $f : D \rightarrow_{\mathcal{P}} C$ to denote a partial one.

- $\mathsf{LowLevelLabs} = \{Load, SLoad, Store\} \times VarArgs \times \mathsf{VALUE}$;

  `assign` and `use` actions are internal action performed by the *tee* (thread execution engine), so there is no information to be sent outside; `load` and `store`, on the contrary, have the corresponding labels:

  - $Load(g_{id}, \mathtt{f}, \mathsf{v})$: a thread loads the value v of a field $\mathtt{f}$ from an object g_{id} in the *wm*.
  - $SLoad(g_{id}, \mathtt{f}, \mathsf{v})$: a load action within a synchronized block, need to be represented by a new label.
  - $Store(g_{id}, \mathtt{f}, \mathsf{v})$: a thread stores a value v of the field $\mathtt{f}$ for an object g_{id} in the MmMng.

- SynchronizationLabs$= \{Lock, Unlock\} \times \mathsf{TH_ids} \times \mathsf{COMP_ids}\}$

  - $Lock(t_{id}, g_{id})$: t_{id} performs a lock action on the lock of g_{id};
  - $Unlock(t_{id}, g_{id})$: t_{id} performs an unlock action on the lock of g_{id};

Transition relation
$\longrightarrow_T\; : T\_states \times T\_labels \rightarrow T\_states$ is defined by the inductive rules in Sect. 3.2.

A.2 Main Memory Manager semantics

The Main Memory Manager (MmMng) models the behaviour of the main memory that can perform low-level actions on variables, as depicted in fig.1. Since MmMng may asynchronously read a value for a thread and send it to *wm* or write a value received from *wm*, it is not just a passive component of the program. It owns some capabilities that need to be modeled by an *LTS*. Essentially, the MmMng internal state is a stack of elements describing the occurrences of `read` actions performed by the MmMng and of `store` actions performed by threads. The MmMng, depending on the elements in the stack may:

- modify the value of an object, by a `write` action, if there is the corresponding `store` in its state;
- send a value to a thread if there is the corresponding `read` in its state.

The semantics of the MmMng is given by:
$\mathsf{MmMng}\_LTS = (\mathsf{MmMng}\_states, \mathsf{MmMng}\_labels, \longrightarrow_M)$.

MmMng_states is set of stacks of Actions elements that are n-tuple in:

- $(\{R, S\} \times ActArgs)$ describing the occurrences of actions `read` and `store`;
- $(\{L\} \times \mathsf{COMP\_ids} \times \mathsf{TH\_ids})$ describing the occurrences of the action `lock`.

where we use the short notation $ActArgs = (VarArgs \times \mathsf{VALUE} \times \mathsf{TH\_ids})$. We use the notation $a_1 : a_2 \ldots$ instead of $a_1 a_2 \ldots$ to make more clear the separation of the elements in the stack.
We will use the following operations on MmMng_states:

- In : $\mathsf{Actions} \times \mathsf{MmMng\_}states$ checks if an action is in the stack;
- $Cancel$: $\mathsf{Actions} \times \mathsf{MmMng\_}states \to \mathsf{MmMng\_}states$ cancels the oldest occurrence of an action in the stack;

  $Cancel(a, Mm) = Mm_1 : Mm_2$ if $Mm = Mm_1 : a : Mm_2$ and $\neg In(a, Mm_1)$;

  $Cancel(a, Mm) = \texttt{undef}$ otherwise.
- $RecCancel$: $ActArgs \times \mathsf{MmMng\_}states \to \mathsf{MmMng\_}states$ cancels any element (if there exists) of the stack differing from the action parameter at most in the value of the variable;

  If $Mm = a' : Mm_1$ then

  $RecCancel(a, Mm) = RecCancel(a, Mm_1)$ if $Similar(a, a')$

  $RecCancel(a, Mm) = a' : RecCancel(a, Mm_1)$ otherwise
  where $Similar((a\_name, var\_args, \mathsf{v}, t_{id}), (a\_name, var\_args, \mathsf{v}', t_{id}))$ if a_name ranges over $\{R, S\}$, var_args ranges over $VarArgs$.
- $Push$: $\mathsf{Actions} \times \mathsf{MmMng\_}states \to \mathsf{MmMng\_}states$: to push an element on the top of the stack: $Push(a, Mm) = Mma$

$\mathsf{MmMng\_}labels \quad = \mathsf{LowLevelActionLabs} \cup \mathsf{LockingLabs}$, where:

- $\mathsf{LowLevelActionLabs} = (\{Write, Read, Loaded, SLoaded, Stored\} \times ActArgs)$: $Write$ and $Read$ describe the writing and the reading of a value, while $Loaded$, $SLoaded$, $Stored$ describe operations to cancel or introduce elements into the stack.
- $\mathsf{LockingLabs}$= $(\{Locked, Unlocked\} \times \mathsf{COMP\_ids} \times \mathsf{TH\_ids}\})$
  - $Locked(t_{id}, g_{id})$: a `lock` action $L(g_{id}, t_{id})$ is added to MmMng;
  - $Unlocked(t_{id}, g_{id})$: it is checked the presence of the corresponding `lock` action $L(g_{id}, t_{id})$ that is canceled from the MmMng.

Transition relation
$\longrightarrow_M$: $\mathsf{MmMng\_}states \times \mathsf{MmMng\_}labels \to \mathsf{MmMng\_}states$ is defined by the inductive rules in Sect. 3.2.

A.3 Program Semantics

The semantics of programs is given by means of P_LTS=(P_states, P_labels, $\to_P$).

$P\_states \quad = Set(T\_states \cup \mathsf{Pass\_states}) \times \mathsf{MmMng\_}states$
$\mathsf{Pass\_states} = \mathsf{O\_states} \cup \mathsf{C\_states}$ is the set of all the passive component states.
$\mathsf{O\_states}$, $\mathsf{C\_states}$ denote, respectively, the set of states for objects and classes;

- $\mathsf{O\_states} = \mathsf{OB\_ids} \times \mathsf{MEMORIES}$.

  An object state $< o_{id}, m >$ consists of:
  - o_{id} : the object identifier;

- m : the memory;

- $\mathsf{C\_states} = \mathsf{CL\_ids} \times Set(\mathsf{OB\_ids}) \times \mathsf{MEMORIES} \times \mathsf{MEMORIES}$.

  A class state $< c_{id}, S_{id}, lm_i, m >$ consists of:

  - c_{id} : the class identifier;
  - S_{id}: a set of identifiers of previously created objects of this class.
  - lm_i : the initial memory for an instance of the class, used for creating new instances.
  - m : the memory;

We use the short notation $gc_1| \ldots |gc_n|Mm$ for $< \{gc_1, \ldots, gc_n\}, Mm >$.

Auxiliary operations Auxiliary operations on program states are:

- $IdInfo : \mathsf{O\_states} \rightarrow \mathsf{OB\_ids}$ characterized by $IdInfo(< o_{id}, m >) = o_{id}$; returns the identifier of the component (analogous for class and thread states);

- $Mem : \mathsf{O\_states} \rightarrow \mathsf{MEMORIES}$ s.t. $Mem(< o_{id}, m >) = m$ returns the memory of the component (analogous for class and thread states);

$P\_labels \quad = \{\tau\}$
The labels describe information made available to an external observer at each step of the computation, i.e., every kind of interaction a Java program can have with the external environment. Since the subset of Java considered in this appendix does not contain I-O operations, we can have only the τ label describing transitions without information exchange with the environment (usually omitted).

Transition relation
$\rightarrow_P \; : P\_states \times P\_labels \rightarrow P\_states$ is defined by the inductive rules in Sect. 3.2.

B The Abstract Operational Semantics

In this Appendix we present the simpler semantics *Abs*.

Again, we formally define the lts for threads and the lts for programs. Notice that we do not need to model an entity that manages the main memory: field values are directly read and write acting on the private memory of each object. As a positive consequence, the number of rules modeling the use of the shared memory is sensibly decreased.

Let us remember that we report only the semantics related to memory management and of the synchronized statement; we also forget class variables and field types.

B.1 Thread Semantics

With regard to the *Con* semantics, the lts modeling threads in *Abs* has simpler states, lacking in the working memory component. The number of labels describing the thread activity is decreased too.

The semantics of threads is given by an lts
$AbsT\_LTS$=($AbsT\_states$,$AbsT\_labels$, $\longrightarrow_T$).

Configurations are slightly modified, since synchronization requires to add to each components (also to non-thread objects) informations about the status of the lock. These information, in *Con*, were recorded in the MmMng.

The status of a lock is a couple $lock = (t_{id}, n)$ where t_{id} is the thread that has the lock of the object and n is the number of locking operations performed by the thread on the lock. $\mathsf{Locks} = \mathsf{TH\_ids} \times Nat$ is the set of all possible locks; it is ranged over by metavariable $lock$;

$AbsT\_states \quad = \mathsf{TH\_ids} \times \mathsf{STATEMENT} \times \mathsf{Status} \times \mathsf{MEMORIES} \times \mathsf{Locks} \times \mathsf{Stacks}$
where the sets $\mathsf{TH\_ids}, \mathsf{STATEMENT}, \mathsf{Status}, \mathsf{MEMORIES}, \mathsf{Stacks}$ are the same defined for *Con* semantics in Appendix A.

$AbsT\_labels \quad = (\{Assign, GetV\} \times \mathsf{NonCL\_ids} \times \mathsf{FIELD} \times \mathsf{VALUE})$
Only two labels are used to model the assignment and the reading of an object value.

Transition relation The transition relation $\longrightarrow_T$ is defined by the rules in Sect. 4.2.

B.2 Program Semantics

The semantics of programs is given by means of
$AbsP\_LTS$=($AbsP\_states$, $AbsP\_labels$, $\rightarrow_P$).

The state of a program is simply a set of passive and active components, where every component is enriched with the status of the associated lock.

$AbsP\_states \quad = Set(T\_states \cup \mathsf{Pass\_states})$
where $\mathsf{Pass\_states} = \mathsf{O\_states} \cup \mathsf{C\_states}$ and

- $\mathsf{O\_states} = \mathsf{OB\_ids} \times \mathsf{Locks} \times \mathsf{MEMORIES}$.

  An object state $< o_{id}, lock, m >$ consists of:

- o_{id} : the object identifier (plus the class information);
- $lock$: describes the status of the lock associated with the object;
- m : the memory;

- $\mathsf{C\_states} = \mathsf{CL\_ids} \times Set(\mathsf{OB\_ids}) \times \mathsf{MEMORIES} \times \mathsf{Locks} \times \mathsf{MEMORIES}$.

  A class state $< c_{id}, S_{id}, lm_i, lock, m >$ consists of:

  - c_{id} : the class identifier;
  - S_{id}: a set of identifiers of previously created objects of this class.
  - lm_i : the initial memory for an instance of the class, used for creating new instances.
  - $lock$: describes the status of the lock associated with the class;
  - m : the memory;

Auxiliary operations on program states are the same defined in *Con*.

$P\_labels \quad = \{\tau\}$
Again, since the subset of Java considered in this appendix does not contain I-O operations, we can have only the τ label describing transitions without information exchange with the environment (usually omitted).

Transition relation
The transition relation $\rightarrow_P$: $P\_states \times P\_labels \times P\_states$ is defined by the rules in Sect. 4.2.

B.3 Auxiliary functions

In this section we report the description of the auxiliary functions used in Sect. 4.2.

$$DecrementLock(lock) = \begin{cases} (t_{id}, n-1) & \text{if } \exists n.lock = (t_{id}, n), n > 1 \\ (t_{id}, 1) & \text{if } lock = (t_{id}, 1) \end{cases}$$

$$LockOwner(tc, lock) \equiv \exists n.lock = (IdInfo(tc), n)$$

$$SetLock(tc, lock) = \begin{cases} (t_{id}, n+1) & \text{if } IdInfo(tc) = t_{id} \text{ and } \exists n.lock = (t_{id}, n) \\ (t_{id}, 1) & \text{if } IdInfo(tc) = t_{id} \text{ and } lock = u\_lock \end{cases}$$

References

[1] E.Coscia and G.Reggio. *An Operational Semantics for Java.* Technical report, DISI, July 1998.

[2] E.Coscia and G. Reggio. *An Abstract Operational Semantics for a Subset of Java Programs.* Technical report, DISI, 1998. in preparation.

[3] G. Steele J.Gosling, B. Joy. *The Java Language Specification.* The Java Series. Addison Wesley, 1996.

[4] R. Milner. A Calculus of Communicating Systems, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1980.

[5] A. Knapp P.Cenciarelli, B.Reus, and M.Wirsing. *From Sequential to Multi-Threaded Java: An event Based Operational Semantics.* In M.Johnson, editor, *Algebraic Methodology and Software Technology.* AMAST '97, Springer, July 1997.

[6] A.Knapp P.Cenciarelli, B. Reus, and M.Wirsing. *An Event-Based Structural Operational Semantics of Multi-Threaded Java.* In Formal Syntax and Semantics of Java. Springer, 1998.

[7] G. Plotkin. *An Operational Semantics for CSP.* In D. Bjorner, editor, *Proc. IFIP TC 2-Working conference: Formal description of programming concepts*, pages 199–223, Amsterdam, 1983. North-Holland.

[8] S. Eisenbach S.Drossopoulou. *Java is Type Safe- Probably. 11th European Conference on Object Oriented Programming*, June 1997.