Verification of data-race-freedom of a Java chat server with VeriFast

Cedric Cuypers *Bart Jacobs* *Frank Piessens*

Report CW 550, June 2009

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Verification of data-race-freedom of a Java chat server with VeriFast

Cedric Cuypers *Bart Jacobs* *Frank Piessens*

Report CW 550, June 2009

Department of Computer Science, K.U.Leuven

Abstract

Even now, when computers have become a vital part of our society, software errors are still common, and their effects can be devastating. From the recent rise of multicores emerged the need for multi-threading software and a way to cope with its typical software errors such as data-races and deadlocks. This paper shows how VeriFast can be used to verify the data-race-freedom of a multi-threaded Java application, by means of a simple Java chat server example. We will cover the verification of jar files in general, how to deal with Java core classes and interfaces such as *ArrayList* and the specifics of verifying a multi-threaded Java application, using *Thread*, *Runnable* and *Semaphore* as building blocks. To achieve this, we need to take a closer look at VeriFast elements such as predicate families, predicate constructors and fractional permissions. This paper is intended as an experience report. We will conclude with some suggested improvements and possible future work.

Verification of data-race-freedom of a Java chat server with VeriFast

Cedric Cuypers Bart Jacobs Frank Piessens
Department of Computer Science, Katholieke Universiteit Leuven, Belgium

Abstract

Even now, when computers have become a vital part of our society, software errors are still common, and their effects can be devastating. From the recent rise of multicores emerged the need for multi-threading software and a way to cope with its typical software errors such as data-races and deadlocks. This paper shows how VeriFast can be used to verify the data-race-freedom of a multi-threaded Java application, by means of a simple Java chat server example. We will cover the verification of jar files in general, how to deal with Java core classes and interfaces such as *ArrayList* and the specifics of verifying a multi-threaded Java application, using *Thread*, *Runnable* and *Semaphore* as building blocks. To achieve this, we need to take a closer look at VeriFast elements such as predicate families, predicate constructors and fractional permissions. This paper is intended as an experience report. We will conclude with some suggested improvements and possible future work.

1. Introduction

In this paper we use the VeriFast tool [4] to verify a simple Java chat server. VeriFast is a tool that verifies safety properties, in the form of annotations in the source code, of some program in C or Java using an approach based on separation logic and symbolic execution. Users may add additional annotations such as inductive data types, abstract predicates etc. that can be used in the verification process. In order to understand this approach, readers are strongly advised to have read the technical report of VeriFast [3] before continuing.Although this technical report describes the verification of a linked list ADT in C, almost everything is equally applicable to Java programs. Any differences will be noted in the rest of this paper.

The basic idea behind multi cores is simple yet powerful: multiple CPUs working together in order to decrease the latency or increase the throughput. The advantages of multi cores may be clear, but in order to use these advantages, special multi-threaded software is required. This software is vulnerable to different security issues than single-threaded software, such as data-races, deadlock etc. For a good overview of multi-threaded software in Java, its pitfalls and solutions, see [6].

First we will show how to verify jar files with VeriFast in Section 2. Then we explain how the use of core Java classes is handled in VeriFast in Section 3. We consider classes and interfaces concerning collections (*List*, *Iterator* and *ArrayList*) in Section 3.1, multi-threading(*Thread*, *Runnable*) in Section 3.2 and locking (*Semaphore*) in Section 3.3. Other supported core classes and interfaces such as *Socket*, won't be discussed in detail. In Section 4 we will show how this support can be used to verify our simple Java chat server example. Finally we conclude with a list of problems we experienced and possibilities for future work that counter them in Section 5.

The binaries of VeriFast, the full sources of the examples discussed in this paper and more can be found in the latest release [4].

2. Verification of jar files

Usually a Java application will consist of a jar file, containing the different components of the application (class files, images, ...). To verify a jar file, we need information about the behavior of the different components. VeriFast supports the following construct to verify jar files: suppose we want to verify *lib.jar*,

- *lib.jarsrc* contains a list of all the Java files in this jar and the other jars used in the implementation of this jar. It can also state the class that contains the main method of this jar, if there is any.
- (optionally) *lib.jarspec* contains a list of *Javaspec* files that deliver the specification of this jar, and a list of other jars whose specification is used in the specification of this jar.

A jar can thus separate its implementation and specification by creating a *jarsrc* file containing a list of plain Java files that make up its implementation, and a *jarspec* file containing a list of *Javaspec* files to describe its specification. *Javaspec* is just an extension used here to indicate a special kind of Java-like file, with the intent of specifying behavior rather than implementing it. A *Javaspec* file can contain specification elements (inductive data types, predicates, lemmas, fixpoint functions) and classes or interfaces with constructors and methods with only their signature and their contract, not their body. This specification can be used in the verification process of this jar or other jars that use this jar. The verification process of a jar proceeds as follows:

1. First the specification of the jar file is type-checked, if the corresponding *jarspec* file is present. This means that the *Javaspec* files in the *jarspec* file are type-checked, together with the specifications of the other jars mentioned in the *jarspec* file.
2. Then the implementation, augmented with the specification, if any, is verified. This means that the Java files in the *jarsrc* file are verified together with their specifications in the *Javaspec* files of the *jarspec* file, and the specifications of any jars listed in the *jarsrc* file.
3. Next, VeriFast checks whether the elements in the *Javaspec* files of that jar that require a concrete implementation, are implemented. For instance if a method of constructor of a class is specified in a *Javaspec* file, it must be implemented accordingly in a Java-file. If a main method is specified in the *jarsrc* file, it must be present in the right class in the implementation.
4. Finally, a *jardeps* file is generated, containing all the jars used in the specification or implementation of this jar.

To illustrate this, consider A.jar, consisting of A1.java and A2.java, and B.jar, with B1.java and B2.java, that uses A.jar in its implementation. So, suppose we have:

- A.jarsrc: A1.java A2.java
- A.jarspec: A1.javaspec A2.javaspec
- B.jarsrc: B1.java B2.java A.jar [main-class: Program]

The verification of A.jar will go as follows:

1. First A1.javaspec and A2.javaspec are type-checked in isolation.
2. Then the specifications in A1.javaspec and A2.javaspec are combined with A1.java and A2.java and the result is verified.
3. The elements in A1.javaspec and A2.javaspec requiring implementation, are checked to be implemented in A1.java or A2.java.
4. Finally, an empty A.jardeps file is generated, because A.jar does not use any other jars.

The verification of B.jar will be similar:

1. There is no B.jarspec file, so there is no specification of the jar file that has to be type-checked.
2. Then the specification of A.jar, being A1.javaspec and A2.javaspec, is verified together with B1.java and B2.java.
3. B1.java or B2.java must contain a class called Program with a main method with true as precondition and postcondition.
4. Finally, B.jardeps file is generated, which consists of a single line "A.jar", because B.jar uses A.jar in its implementation.

A complete program can consist of multiple jars (at least *rt.jar* (see Section 3) and the jar with the main method). VeriFast offers the following **soundness statement**: if all jars verify, and there are no cycles in the jar dependency graph, than the program is **safe**, meaning it won't throw any assertion errors. In this context the *jardeps* are used to ensure that there are no mutually recursive lemmas.

3. Core Java classes in VeriFast

If we want to be able to verify programs that use core Java classes and interfaces, such as *Object* and *String* but also *Thread*, *Runnable* and *Socket*, then we need to create specifications for these core classes and interfaces so their behavior can be used in the verification process. In VeriFast every Java file or jar file implicitly uses the specifications listed in *rt.jarspec*. This *jarspec* file simply consists of a list of *Javaspec* files who specify the behavior of some core Java classes and interfaces. Notice that the implementation of this *jarspec* file is not considered, since it is provided by the Java API.

Currently there is support for the following core classes and interfaces: *Object*, *Class*, *String*, *Runnable*, *Thread*, *Semaphore*, *List*, *Iterator*, *ArrayList*, *StringBuffer*, *Socket*, *ServerSocket*, *InputStreamReader* and *OutputStreamWriter*. Not every method of these classes is present in the specifications, only the ones that are necessary in order to verify our Java chat server example. In the future, extra specifications or additional core classes or interfaces can be added very easily by adapting the current *Javaspec* files or adding a new one to *rt.jarspec*, though writing correct and useful specifications is far less trivial.

One of the encountered difficulties was that some of the methods of those core classes and interfaces can throw checked exceptions, such as *InterruptedException*, *IOException* etc. , that must be handled properly by *try/catch* statements or *throws* list at the call site. Currently there is no support in VeriFast for either of those, so we used a wrapper around the original core class, that catches all checked exceptions and converts them in a non-checked exception such as *RuntimeException*. The wrapper classes have the name of the core class, extended with an underscore and they are located in subpackages of a *wrapper* package. For simplicity reasons, we will use the names of the core classes or interfaces instead of the names of the wrapper classes in the rest of this paper. Support for checked exceptions in VeriFast is a must if we want to handle more realistic Java applications.

VeriFast has some support for *string literals* in Java. Every string literal is considered to be of type *String* and at the evaluation of a string literal, an assumption that the resulting *String* object is not null, is added to the assumptions. At the moment there is no support for specifications involving knowledge of the contents of strings.

```
package java.util;

inductive listval = nil | cons(Object, listval);
predicate list (List l, listval v);
predicate iter (Iterator i, List l, listval v, int i);
lemma void iter_dispose(Iterator iter);
  requires iter(iter, ?l, ?v, ?i);
  ensures list(l, v);
interface List {
  boolean add(Object element)
    requires list(this, ?v);
    ensures list(this, list_add(v, element)) * result;
  boolean remove(Object element)
    requires list(this, ?v);
    ensures contains(v, element)?list(this, remove(v, element))
      : list(this, v);
  Iterator iterator()
    requires list(this, ?v);
    ensures iter(result, this, v, 0);
}
interface Iterator {
  boolean hasNext()
    requires iter(this, ?l, ?v, ?i);
    ensures iter(this, l, v, i) * result = (i < length(v));
  Object next()
    requires iter(this, ?l, ?v, ?i) * i < length(v);
    ensures iter(this, l, v, i + 1) * result = ith(v, i);
}
public class ArrayList implements List {
  public ArrayList()
    requires true;
    ensures list(result, nil);
}
```

Figure 1. Specifications of List, Iterator and ArrayList (Note: annotations are shown on a gray background. Also, for readability, we typeset some operators differently from the implementation.)

```
package wrapper.lang;

predicate_family thread_run_pre(Class c)
(Runnable run, any info);
predicate_family thread_run_post(Class c)
(Runnable run, any info);
interface Runnable {
  void run()
    requires thread_run_pre(this.getClass())(this, ?info);
    ensures thread_run_post(this.getClass())(this, info);
}
predicate thread(Thread_ thread, Runnable run);
predicate thread_started(Thread_ thread, Runnable run,
  any info);
public class Thread_ {
  public Thread_(Runnable run)
    requires true;
    ensures thread(result, run);
  void start()
    requires thread(this, ?run)*
        thread_run_pre(run.getClass())(run, ?info);
    ensures thread_started(this, run, info);
  void join()
    requires thread_started(this, ?run, ?info);
    ensures thread_run_post(run.getClass())(run, info);
}
```

Figure 2. Specifications of Thread and Runnable (Note: the exact semantics of the *join* method were slightly changed in the *Thread_* wrapper because of a soundness issue.)

3.1 Collections: specifications of *List* and *Iterator*

We did achieve full functional correctness of *List*, *ArrayList* and *Iterator*. The most important elements are shown in Figure 1. The inductive data type *listval* is used to reason about the contents of a list. The *list* predicate binds a *List* object with its inductive value. The *iter* predicate does the same for an *Iterator* object and its *List* object, inductive value and current index. The contracts of the methods and the constructor are pretty straightforward and they will not be discussed in details here. We must state that the contract of the *remove* method is only sound for objects of classes that do not override the *equals* method of *Object*. For space reasons, the fixpoint functions operating on *listvals* (*ith*, *length*, *remove*, *list_add* and *contains*) were not added to Figure 1. A minor inconvenience is that VeriFast does not support Java *generics* (yet), so all elements in the list are of type *Object* and casts are sometimes necessary in the code.

3.2 Multi-threading: specifications of *Thread* and *Runnable*

We won't give the full specifications of *StringBuffer*, *Socket*, *ServerSocket*, *InputStreamReader* and *OutputStreamWriter*. Instead we will focus on the details of the contracts used in *Thread*, *Runnable* and *Semaphore*, since these are more interesting when considering the verification of multi-threaded Java applications [2]. The specifications of *Thread* and *Runnable* can be found in Figure 2.

Predicate families [5] such as *thread_run_pre* and *thread_run_post* are a special construct in VeriFast: they represent a collection of predicate family instances, with the same name and arguments, who are indexed by *Class* objects. When a predicate family is closed or opened, the correct instance is searched using the index argument (between the first pair of brackets). This construct is particularly useful in cases of inheritance: the super class or interface can use the predicate family in its contracts, with as index argument the actual class of the implicit *this* argument. When the overridden or implemented method of the subclass is called, the predicate family instance with that subclass as index, will be used in the contract of that method. Later in our chat server example (Section 4) we will show how to use this construct when we want to develop a class that implements *Runnable*. Notice the second argument of *thread_run_pre* and *thread_run_post*,called *info*. This is an inductive data type value that may contain some additional information. It has type *any*, which means that *info* can be of any type of inductive data type. It can be useful when the join method is used to synchronize threads, but in this paper, this is not the case, so at the call site it will always be *unit*, meaning nothing really.

The *Runnable* interface contains only one method: the *run* method, which is called automatically when a new thread, constructed with that *Runnable* object, is started. The contract of this method has the following effect: when called, the predicate family instance of *thread_run_pre* with the actual class of *this* as index, is consumed and the predicate family instance of *thread_run_post* with the same index is produced. When a class wants to implement *Runnable*, it just has to create predicate family instances of *thread_run_pre* and *thread_run_post* indexed with that class, and of course implement the *run* method so that it is consistent with the contract in the *Runnable* interface.

The *Thread* class contains one constructor, that takes a *Runnable*-argument and produces a *thread* predicate on the heap when called. This predicate is used to bind the newly constructed thread with its runnable and it is required in order to be able to successfully call the *start* method. The *start* method requires a *thread* predicate as mentioned above and a predicate family instance of *thread_run_pre* indexed with the actual class of the runnable of this thread, on the heap. In turn it produces a *thread_started* predicate on the heap, indicating that the thread has been started. The *join* method takes a *thread_started* predicate from the heap, and replaces it with the correct predicate family instance of *thread_run_post*, meaning that the thread has stopped running and that the postcondition of its runnable is now valid.

There was an soundness issue concerning the *join* method of *Thread*. Suppose thread *B* creates a new thread *A* with *Runnable* object *R*, starts the new thread and then *B* joins with *A*. If the execution of the *run* method of *R* would end with an unchecked exception before its body is fully executed, it might be that the postcondition of this method does not hold. This would mean that the postcondition of the *join* method of *A* would also not hold, but the joining thread *B* would not be able to detect this because the join would just succeed normally. This soundness issue is the reason why we changed the exact semantics of the *join* method in the wrapper implementation *Thread_*: the wrapper will remember if any exception was thrown during the execution of the *run* method of the *Runnable* object of the *Thread* object and when another thread joins with it, the *join* method will throw a new unchecked exception containing the original exception as its inner exception.

3.3 Locking: specifications of *Semaphore*

In order to prove data-race-freedom, some form of locking can be necessary. In VeriFast, the notion of locking is provided by the specification of *Semaphore* in Figure 3. Notice that VeriFast does not support reentrant locks (yet), so we had to choose *Semaphore*

```
package wrapper.util.concurrent;

predicate lock(Semaphore_ s; predicate ()inv);
predicate create_lock_ghost_arg(predicate ()inv)
  requires inv();
lemma void lock_dispose(Semaphore lock);
  requires lock(lock, ?a);
  ensures a();
public class Semaphore_ {
  public Semaphore_(int n)
    requires n = 1 * create_lock_ghost_arg(?a);
    ensures lock(result, a);
  void acquire()
    requires [?f]lock(this, ?a);
    ensures [f]lock(this, a) * a();
  void release()
    requires [?f]lock(this, ?a) * a();
    ensures [f]lock(this, a);
}
```

Figure 3. Specifications of Semaphore (Note: the actual class is called Semaphore_ because we had to use a wrapper to convert the checked exceptions to unchecked.)

instead of something like *ReentrantLock*. *Semaphore* has one constructor that takes an integer as argument that must be one to achieve a mutual exclusion lock, and produces a *lock* predicate to bind the new semaphore to its lock invariant. The contract is actually also sound for multiple permits (and it should be because *Semaphore* allows multiple releases), but this will only verify if the lock invariant can occur multiple times, which is not the case for e.g. fields, but it is the case for e.g. fractions of fields. The semaphore owns as many copies of the lock invariant as its number of permits. So a non-acquired lock owns the lock invariant, while an acquired lock does not own anything. When constructing a lock, ownership of the lock invariant is transfered to the lock. Acquiring the lock, transfers the ownership back to the thread, while releasing it does the opposite and transfers it back to the lock. This is precisely what the contracts of the constructor and the *acquire* and *release* methods state. If a thread wants to perform a sensitive operation that has the lock invariant as part of its precondition, it first has to acquire the lock in order to own the lock invariant so the precondition can be fulfilled. When it no longer needs the lock invariant, it can transfer ownership of it back to the lock by releasing it.

The *create_ghost_lock_arg* serves to pass the lock invariant as a ghost argument to the constructor. The square brackets before the *lock* predicates in the contracts of acquire and release, indicate that *fractional permissions* [1] are allowed. In VeriFast a predicate can be split in multiple fractions that can be used in the contracts of methods or constructors, afterwards those fractions can be joined together again. This construct allows sharing of predicates amongst for instance multiple threads. Applied here, this means that a thread can still acquire/release a lock, even when the *lock* predicate is shared by multiple threads, which is exactly the point of a lock.

4. Verification of the Java chat server example

We will now illustrate how to verify a multi-threaded Java application by a simple chat server example with the following classes:

```
public static void main(String []args)
  requires true;
  ensures true;
{
  Room room = new Room ();
  close room_ctor(room)();
  close create_lock_ghost_arg(room_ctor(room));
  Semaphore_ roomLock = new Semaphore_ (1);
  ServerSocket_ serverSocket = new ServerSocket_ (12345);

  while(true)
    invariant [_]lock(roomLock, room_ctor(room))*
      server_socket(serverSocket);
  {
    Socket_ socket = serverSocket.accept();
    split_fraction lock(roomLock, _);
    Session session = new Session (room, roomLock, socket);
    close thread_run_pre(Session.class)(session, unit);
    Thread_ t = new Thread_ (session);
    t.start();
  }
}
```

Figure 4. Main method in the *Program* class of the Java chat server example.

- *Member*: a member of a chat room, with a nickname and an outputstream.
- *Room*: a chat room, with a list of members present in that room.
- *Session*: a chat session per member connecting through a socket; implements *Runnable*.
- *Program*: contains main method that constructs one chat room and a corresponding lock and one server socket and waits forever for client connections on this server socket; creates a new thread and session for each incoming client connection.

Our focus will be on *Session* and *Program* because they render the application multi-threaded. The most important elements of *Session* can be found in Figure 5, the *Program* class consists of only the main method, which is given in Figure 4.

The main method simply creates a new chat room and a semaphore to lock it, and a server socket. Then it listens for client connections to this server socket in an endless loop, creating a new session for every incoming connection and a new thread to run this session. Notice that the *split_fraction* ghost statement splits the lock predicate in smaller parts, so the *lock* predicate can be shared by multiple threads. Finally, it starts the new thread and goes back to wait for new connections to the server socket. The loop invariant of the loop simply states that with every execution, there is always a certain fraction of the lock predicate preserved with each execution and the server socket continues to exist.

Session is the class that implements the *Runnable* interface. As said in Section 3.2, Session must create a predicate family instance of *thread_run_pre* and *thread_run_post*: the first will just consist of a session predicate, that binds the *Session* object with its fields, the latter is simply true because after the run method is finished, the session has no further use. The predicate constructor *room_ctor* is used to create a constructed predicate value *room_ctor*(room) to be used as a lock invariant for the semaphore of the chat room. The

run method is a rather long method, that does a lot of input and output using the in- and output stream of the socket of the session. The flow of the run method is that it first prints a welcome message to the new client, listing the members that are already present in the room. Next it prompts the client for its nick: if the there is already a member with the same nickname, it refuses the nick and sends an error message to the client and requests a different nick; otherwise the thread continues by calling another method, called *run_with_nick*. This method starts by adding the new member to the room. Secondly, it broadcasts any message said by the new member, to the other members in the chat room, until the new member ends the session by sending an empty string. Finally the member is removed from the room, and a goodbye message is sent to the members that are left in the room.

The *run* method in Figure 5 is a simplified version of the actual one, to illustrate how VeriFast handles the locking of the chat room. The sensitive operation [***] will have the *room* predicate in its pre- and postcondition. In order to satisfy these conditions, the lock of the room must first be acquired, since that will transfer ownership of the lock invariant to this thread. Afterwards, when the predicate is no longer required, the lock can be released and ownership of the invariant will be transfered back to the lock, so other threads can try to claim ownership of the lock invariant. Actually the lock does not really have to be released; not releasing it will just keep the lock locked longer than necessary, but it won't introduce any data races, so it is still valid.

5. Conclusion

The Java chat server example shows that VeriFast can be used to verify multi-threaded Java software, but there are still some problems or inconveniences. We will give a short overview of some possibilities for future work:

- In order to deal with more realistic Java applications, VeriFast needs to be able to handle checked exceptions, *try/catch* statements and *throws* lists. This might increase the annotation overhead, since one might have to specify the behavior separately for each kind of exception.
- Now there is only support for a list of *Object* elements, and thus casts are needed if we want to use for instance a list of *Member* elements. Adding support for Java generics would be extremely useful for this case.
- Only a small subset of core classes and interfaces and their methods are supported in VeriFast. Expanding this subset will probably happen example driven.
- There are some *assume* statements in the code of the chat server example. They are needed because the solver sometimes can't prove certain statements. For instance, in the *run_with_nick* method of *Session*, an *assume* statement is needed to assure the prover that the new member is still in the list of members in the chat room, after having added him and having sent some messages. The reasons why these statements are needed, are too complex to explain here; but the goal of course is to eliminate the need for any *assume* statements.

```
predicate_ctor room_ctor(Room room)()
  requires room(room);

predicate session(Session session)
  requires session.room↦?room*
  session.room_lock↦?roomLock*
  session.socket↦?socket*
  socket(socket, ?reader, ?writer)*
  [?f]lock(roomLock, room_ctor(room))*
  reader(reader) * writer(writer);

predicate_family_instance thread_run_pre
  (Session.class)(Session session, any info)
  requires session(session);

predicate_family_instance thread_run_post
  (Session.class)(Session session, any info)
  requires true;
public class Session implements Runnable{
  Room room;
  Semaphore room_lock;
  Socket socket;
  public void run()
    requires thread_run_pre(Session.class)(this, ?info);
    ensures thread_run_post(Session.class)(this, info);
  {
    open thread_run_pre(Session.class)(this, info);
    open session(this);
      ...
    roomLock.acquire();
    open room_ctor(room)();
    open room(room);
      [***]
    close room(room);
    close room_ctor(room)();
    roomLock.release();
      ...
    close thread_run_post(Session.class)(this, info);
  }
    ...
}
```

Figure 5. Some predicate elements and the run method of the *Session* class. [***] indicates a sensitive operation that accesses the chat room.

References

[1] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. *In Proc. POPL*, 2005.

[2] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS*, 2007.

[3] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, August 2008.

[4] Bart Jacobs, Frank Piessens, Cedric Cuypers, Lieven Desmet, and Jan Smans. VeriFast. Website, 2008. `http://www.cs.kuleuven.be/~bartj/verifast/`.

[5] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. *In Proc. POPL*, 2005.

[6] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005. ISBN 0321349601.