Verifying Dijkstra’s algorithm in Jahob

Robin Mange, Jonathan Kuhn

Ecole Polytechnique Fédérale de Lausanne
{robin.mange, jonathan.kuhn}@epfl.ch

Abstract. This report presents how to efficiently prove correctness of JAVA implementations. To this end we used Jahob verification system which is one of the most evolved verification tool for Java-like programs nowadays. We chose to analyze Dijkstra’s algorithm because it is widely used in routing protocols. It is simple, yet allows us to cope with many types of software errors.
This project was motivated by today’s increasing need of bug-free programs, the main reason being that software development and maintenance are expensive tasks.

Keywords: Software verification, Jahob, Dijkstra’s algorithm

1. Introduction

Over the end of last century, new technologies such as computers and the Internet have become essential in several tasks. Software is at the core of such technologies, it is present in a broad range of tools: from fridges interfaces, through Internet browsers, as well airplane navigation systems. Until recently, software’s correctness wasn’t the main preoccupation in computer science since it was mostly used for mundane usage. Now there is an increasing demand of software in critical domains, such as in medical industry (e.g. with pacemakers), in space missions or security-related systems. That’s why it has become necessary to have correct programs that won’t fail at unexpected times.
Heeding this call, computer scientists have been developing verification tools to prove correctness of software. Jahob is one of those new developed systems that we will use in this project. It is oriented towards verifying JAVA-like programs.

2. Dijkstra's algorithm

Dijkstra's algorithm is a greedy algorithm that solves the shortest path problem from a given source to all the other nodes of a directed graph. It is widely used in Internet routing protocols such as OSPF (Open Shortest Path First). It can be used in every link-state routing protocol: that is when every node knows the complete topology of the network (and every time it changes, the whole map is updated at each node).

It takes a weighted directed graph $G = V \times E$[1] with non-negative edges and a source *s* as input and computes the shortest path to every other node in the graph.

It works as follows: Initialize all distances from s to v as infinity, and from s to s at zero; stores all the unvisited nodes. Then, as long as each node hasn't been reached, takes the shortest distance d[v]: if a direct neighbor allows a lower cost path than current cost, it is updated.
Here is the pseudo-code of the algorithm:

```
 1  function Dijkstra(Graph, source):
 2     for each vertex v in Graph:
 3         dist[v] := infinity
 4         previous[v] := undefined
 5     dist[source] := 0
 6     Q := copy(Graph)
 7     while Q is not empty:
 8         u := extract_min(Q)
 9         for each neighbor v of u:
10             alt = dist[u] + length(u, v)
11             if alt < dist[v]
12                 dist[v] := alt
13                 previous[v] := u
```

Fig1. Pseudo-code of Dijkstra's algorithm (Wikipedia)

Note that our implementation follows this one closely, except that we are dealing with undirected graphs only. Indeed when we add an edge between two nodes, the weight is assigned for both directions.

[1] V being the set of vertices and $E \subseteq V \times V$ being the set of edges in the graph

3. Overview of Jahob

Jahob verification system has been developed at the Massachusetts Institute of Technology in the framework of Viktor Kuncak PhD thesis (which happens to be our instructor). The interest was to focus on techniques for automatically proving formulas that arise in the verification context. It is developed in Objective Caml which is functional programming language, and is still under active development. It is aimed at a subset of Java as implementation language.

Let's briefly introduce the functioning of Jahob verification process.
In order to prove programs' correctness, developers have to provide specifications about its behavior in the form of annotations in high-order logic (HOL); they are expressed as JAVA comments followed by ":". They are then translated into a guarded command language to generate verification conditions. Jahob reduces the verification problem to the problem of validity of HOL formulas.

Here is a representation of Jahob's possibilities and links with several theorem provers and other verification tools.

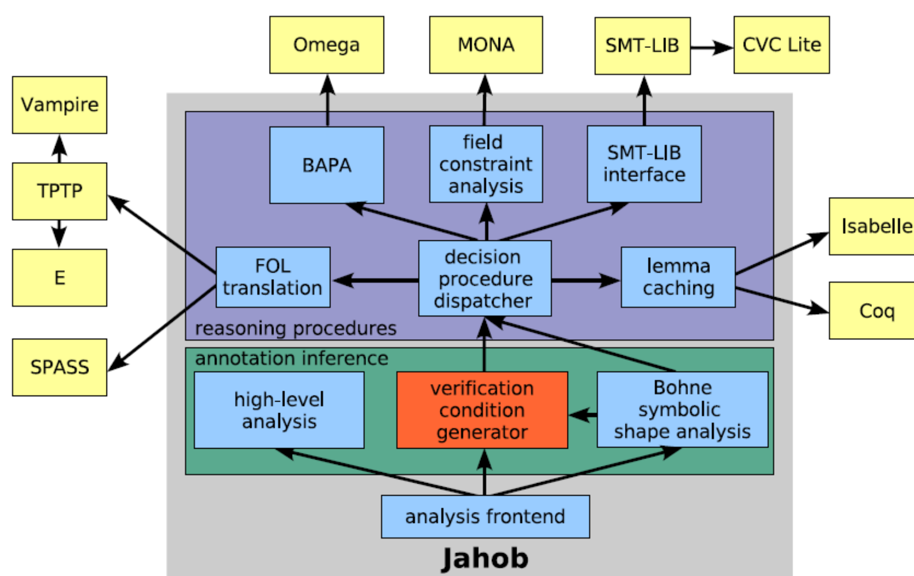
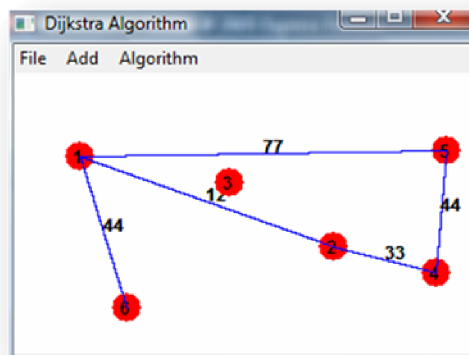


Fig 2. Jahob system architecture. see [3]

4. Implementation verification

We implemented a version of Dijkstra's algorithm in JAVA. We are not really interested in his functioning since it's not our main preoccupation, but in brief we created a graphical user interface with a menu giving us the possibility to create new nodes and edges with corresponding distances (see the figure on the right); but for the verification part using Jahob, we were only interested in the core algorithm. Then we simply extracted the data structures and methods that were directly implied in the core of the shortest-path computation algorithm.
In the next sections we will discuss the process of annotations we followed and some of the difficulties we encountered during this project.



4.1 Specifications

To correctly specify the behavior of a program a certain annotation scheme has to be followed. This scheme uses the standard concepts of preconditions, frame conditions, postconditions, invariants and specification variables.
Preconditions are expressed using a ***requires*** clauses, frame conditions as ***modifies*** clauses and postconditions as ***ensures*** clauses. Class invariants denote properties of a class that are available in each reachable state and are declared as ***invariant*** clauses in the beginning of a class. The specification variables don't affect the execution of the program but help the reasoning about the behavior. There exists also

ghost variable that changes only on explicit assignments and are independent of other variable.
For example, here is the global specification of the class Vector, followed by a method of this class, using the previously defined clauses:

```
class Vector
{
	private static Integer[] a;
	public /* readonly */ int size;

	/*:
	public static ghost specvar init :: bool;
	public static specvar content :: objset;
	vardefs
		"content == {n. EX j. n = a.[j] & 0 <= j & j <
	size}";
	invariant "init --> a ~= null & 0<a..Array.length & 0 <= 20
	& 20<=a..Array.length";
	*/


	…
}
```

In those statements, we declare a specification ghost variable *init* which will implies formulas defines in the invariant field as soon as it is set to *True*. This assignment will be done in an *initialize* method. Another specification variable *content* is defined and correspond to an existing object of the array *a*.
Those global specifications are quite small in this case, but can become very big when dealing with several arrays.

Here is a method using the above global specifications to define its {pre/frame/post} conditions. This method is simply designed to add an object to an array if the there is still som place.

```
public void add(Integer e)
/*:
	requires "init & e~=null"
	modifies content, "Array.arrayState", size
	ensures "((content = old content Un {e}) & (size =
	(old size) + 1)) | ((content = old content) & (size = (old
	size)))";
*/
{
	if ((a == null) || (e == null)) {
		//: noteThat "content = old content";
		//: noteThat "size = old size";
		return;
	}
```

```
        if ((size>=0) && (size < a.length)) {
                if (a[size] == null) a[size] = new Integer();
                        a[size] = e;
                        size = size + 1;
                        //: noteThat "content = old content Un {e}";
                        //: noteThat "size = old size + 1";
                }
                else {
                        //: noteThat "content = old content";
                        //: noteThat "size = old size";
                }
        }
}
```

Here the method requires that the specification variable *init* is true and that the object *e*, here an Integer, given in parameter is not null (***requires***). The fact that *init* must be True at the start of the method simply say that the object a needs to be initialized.
In the 2nd statement simply declare which objects and variables will be modified during the execution of the method (***modifies***).
We then specify the state at which the method should arrive after the execution of the method (***ensures***). Here we make sure that either the content is increased with the new object *e*, or that the content remains unchanged.
Inside the method, the "*noteThat*" statements are used to establish lemmas about current program state to help Jahob.

Jahob provides also some other features, such as allowing the definition of loop invariant which is very useful. We used them a lot since we have many loops in our main algorithm. The syntax is quite similar as what we saw above; here is an example of such a loop:

```
Integer tmp = new Integer();

int i = 0;
while  //: inv "0 <= i & i <= nb & (vec1~=null) & (tmp~=null) &
        theinvs";
(i < nb) {
        tmp.a = i;
        if (i != start) vec1.add(tmp); //Add all elements except
                                         start to the vector
        i = i + 1;
}
```

We need to define what is happening exactly inside the loop. This is done with the *inv* statement, which makes sure that at each iteration the given formulas are respected. In this example, we express that *i* must always be between *0* and *nb* and that the objects *vec1* and *tmp*

must never be *null*. We will explain *theinvs* in the next section, but we can already say that it was necessary to use it each time that methods were used inside a loop.

The actual prover we used in Jahob was CVC Lite. Here is the actual command used:

./bin/jahob.opt ../Dijkstra.java –class Dijkstra –sastvc –usedp cvcl

The option -sastvc generate verification conditions from simplified Ast (abstract syntax tree) and we used to automatically desugar procedure calls. The usedp option selects which decision procedure to use. On some occasions we also used –failfast: that makes Jahob stops as soon as one obligation fails. We sometimes used –isa as theorem prover with a timeout to be able to check pending logs, which was really useful to debug some problems we got during the annotation phase.

4.2 Problems encountered

Jahob is a very powerful modular software checker, but unfortunately supports only a subset of JAVA. In consequence some features of JAVA are not accepted, like concurrency, exceptions, generics and libraries importations. The last one was quite a problem for us since we were using the Vector object and the Integer object of JAVA. To compensate this we had to simulate those classes with faked objects in replacement. For example the class Vector discussed in the previous chapter is one of those objects we had to create.
We also had some problems with function calls inside loops: the class invariants weren't taken into account in the loop invariants. To solve this problem we used a small trick given by the Jahob creator himself; that is to give *theinvs* to the loop invariant.
Another thing is that Jahob doesn't support integer-valued arrays, only object-valued arrays. They are treated differently in the decision procedure. To overcome this difficulty, we created the faked Integer object, which was actually already needed by our implementation; but it would have been nice if we could have used int-valued arrays at some places.

5. Results

5.1 Verification Succeeded!

After a lot of effort invested (since we started as beginners in Jahob), we managed to successfully verify our implementation of Dijkstra's algorithm under our specification design.
Our program consists of about 200 lines of codes and about 80 lines of annotations were introduced for Jahob. We first proceeded to write the implementation in JAVA, and then it took us a month to fully annotate and verify the program. Until now, Jahob was mainly used to prove data structures such as linked lists, but since we had to deal with many runtime error checks, then the time we spent to make our verification scheme working was quite long. It would have been more benefic and less time-consuming as a whole to write specifications during the development whenever it's possible.

The complete execution time of Jahob to verify our program is about 1 minute (around 40s for the main algorithm); this was after having rewritten our code in a more modularized way. Before that, it was taking easily more than 20 minutes. Let's recall that verifying a program is exponential in the code size (due to generating verification conditions). Thus it is important to write the code with as many structured modules as possible to reduce the explosion of time effectively.

5.2 Benchmarks

To make sure that our program followed our design specifications correctly, we intentionally introduced bugs in it such as non initialization of objects, wrong array bounds, etc. For example if we don't check bounds for array usage, then we will have a failure in the verification of the type: *'ObjNullCheck'*.
In the majority of the tests we tried we observed that the intentional errors were catch by Jahob. When they were not detected, this was because of some specification errors or missing. This is quite interesting to test a program like this, and it could also help to see if no specifications are missing, and then can help to precise our behavior design.

6. Conclusion

This project allowed us to see that programs (in certain cases) are very tough beasts to tame. Jahob proved us to be a very efficient software checker. It can be very important to check the correctness of programs in several today's applications. We gained some precious insight about software verification. This was extremely interesting to deal with such correctness prover since verification system could become (and probably will become) more and more used in a near future.

7. Related work

What we would have liked to do would have been to compare the results between Jahob and ESC/Java, which is another (and older) compile-time checker. While they don't support the same features, as shown on the following table, the annotations are quite similar and translating those from Jahob to ESC/Java would have been straightforward.

	ESC/Java	Jahob
Goal	find bugs	prove correctness
Spec. language	JML	based on Isabelle/HOL
Java support	aims at full Java	subset of Java (no exceptions, no concurrency, no generics, no dyn. dispatch, ...)
Loop invariants	none	provided by user or automatically derived
Completeness	only linear arithmetic with free function symbols	general purpose theorem provers and decision procedures for specialized theories

Fig.3 Comparison of ESC/Java and Jahob. See [3]

One particularly important difference is that Jahob's objective is to prove soundness while ESC/Java makes more trade-offs on it. Unfortunately we didn't manage to install ESC/Java on the EPFL machines for rights reasons (root rights required).

Jahob in its current state is already very powerful, but could be improved in some ways (increase JAVA support, e.g. libraries importation, etc).

8. Bibliography

[1] V. Kuncak. Modular Data Structure Verification. PhD Thesis, MIT CSAIL, USA, February 2007

[2] V. Kuncak, M. Rinard. An Overview of the Jahob Analysis System. MIT CSAIL, USA, 2006

[3] T. Wies. Formal Methods for Java (Software engineering lecture notes), Albert-Ludwigs-University Freiburg, May 2007

[4] C. Flanagan, K. Rustan M. Leino, M. Lillibridge
G. Nelson, J. B. Saxe, R. Stata. Extended Static Checking for Java. In PLDI'02, Berlin, June 2002

[5] C. Bouillaget, V. Kuncak, T. Wies, K. Zee, M. Rinard. Using First-Order Theorem Provers in the Jahob Data Structure Verification System. 2007