The KeY Approach on Hagrid

VerifyThis Long-Term Challenge 2020

Stijn de Gouw, Mattias Ulbrich, and Alexander Weigl

[1] Open University
[2] Karlsruhe Institute of Technology

1 Introduction

We present the results of the application of the KeY verification approach to the VerifyThis Long Term Challenge 2020.

KeY [1] is a deductive program verification engine to show the conformance of Java Programs to their specification in the Java Modeling Language (JML). It supports sequential Java 1.4 and the full JavaCard 3.0 standard. The deductive engine of KeY is based on a sequent calculus for a dynamic logic for Java and supports both interactive and automatised verification.

2 Verification of the Subject

The verification target of the challenge is the HAGRID key server, a new implementation of the PGP key server written in Rust that makes the key server conform to data protection regulations and increases resilience against denial of service attacks.

Since KeY operates on programs written in Java, it cannot directly be used to verify HAGRID's Rust source code. Hence, a simplified re-implementation of the of the core functionality of the HAGRID key server in Java had to be written. We came up with two different Java implementations of different complexity. Both adhere to the natural language specifications in [2] The first version implements a single class that only makes use of primitive data types and arrays. The second version modularizes the first version and uses an implementation of a map data structure. Both versions are abstractions of HAGRID's implementation and actual behaviour. We focus on the database logic and leave network connection, and en- and de-coding of HTTP messages aside for this project. Moreover, in the implementation, we assume that e-mail addresses and keys are "atomic" in the sense that they are used as keys and values in the database, but are never analysed for their contents. In particular, we avoid the use of objects for the data and represent them by primitive integer values. This is of course a severe simplification, but since strings are objects in the Java programming language, they produce significantly more difficult verification conditions due to additional heap framing conditions which need to be shown.

We were able to specify and verify both implementations successfully.

Table 1. Verification in numbers of lines of code, lines of specification, applied rules, interactions, and proof obligations.

Version	lines of code	lines of spec	rule applications	interactive rule appl.	proof obligations
Plain	69	82	30.119	0	10
Map-based	146	262	77.663	89	40

A simple email-key map. The first version bases upon five integer arrays. These arrays store:

- the email (identification) of the user
- one array for confirmed and one array for unconfirmed keys
- an array that stores confirmation codes, and
- an array that stores which operation was most recently requested.

The maximum number of users is fixed to 1024, as the arrays are never resized. The implementation only allows to confirm last requested action, e.g. if a deletion is requested, a pending addition is abandoned. We avoid the use of any objects to avoid dealing with a changes of the heap, resulting in a version that is verifiable without interactions in KeY. Table 1 shows the aggregated metrics of the proofs.

We also attempted to add a 'time-out' mechanism, to cover the following aspect of the challenge:

> If the provided code *is one recently issued*, then the corresponding operation (addition/removal) is finalised.

This is easy to add in the implementation: first store the time that the user requests the operation (in an additional array), and when confirming, only approve the operation if that time was sufficiently recent. But it is problematic for specification and verification: the time limit may not yet have elapsed when the precondition (i.e. the specification) is evaluated, but it may have when the JVM determines the current time in the `confirm` method body. So we dropped the time-out aspect.

The map-based approach. The second version follows the same design principles as the first one, but aims to achieve a more object-oriented, modular architecture. To this end, the key server now contains four map data structures for the stored keys, pending additions and pending deletions.

Fig. 1 gives an overview of the class layout. The interface `KIMap` (Key Integer Map) represents a map of from `int` to `int`. Its functionality is specified (by JML contracts) using the abstract map theory built into KeY. `KIMapImpl` is a simple implementation based upon two *int*-arrays (one for the keys and one for the values). `KeyServer` is the verifying key server providing the functionality to

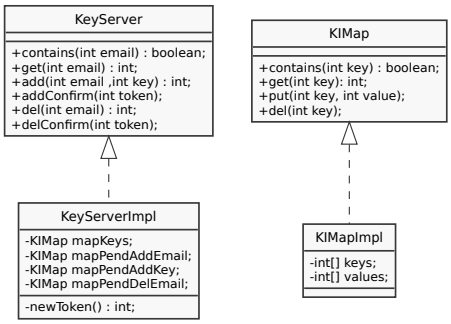

Fig. 1. UML class diagram of the Map version

answer queries for keys, to process requests for key addition and deletion and to perform these mutation operations upon confirmation. It is specified using a number of model fields containing (finite) logical maps.

3 Verification Results

We were able to verify strong functional method contracts for all methods of the implementations. This includes verifying that requested operations are confirmed by the right confirmation code, absence of runtime exceptions and a guaranteed termination of each request handler.

We noticed during the verification of the modular map approach that discharging the framing conditions brought the KeY on the edge of its capabilities. In the following, we devised a new technique to deal with framing conditions that combines dynamic frames with aspects from ownership. This allowed us to close all proofs successfully.

Both implementations used integers instead of Strings as an simplifying abstraction. The obvious next goal is the verification of an implementation which uses `String` values for e-mails and keys.

References

1. Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book: From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, December 2016.
2. Marieke Huisman, Raúl Monti, Mattias Ulbrich, and Alexander Weigl. VerifyThis collaborative long term challenge: The PGP key server. `https://verifythis.github.io/VerifyThisLongTerm.pdf`, 2019. [Online; accessed 2020-02-27].