The VerifyThis Collaborative Long Term Challenge

Marieke Huisman[1], Raúl Monti[1], Mattias Ulbrich[2], and Alexander Weigl[2(✉)]

[1] University of Twente, Enschede, The Netherlands
{m.huisman,r.e.monti}@utwente.nl
[2] Karlsruhe Institute of Technology, Karlsruhe, Germany
{ulbrich,weigl}@kit.edu

Abstract. Over the last years, we have seen tremendous progress in the area of deductive program verification. To demonstrate this progress, and to bring the area of deductive program verification even further, we have proposed the VerifyThis Collaborative Long Term Challenge, which calls upon the program verification community to verify different aspects of a realistic software application over a period of several months. Goal of the challenge is to foster collaboration in order to verify a realistic and industrially-relevant software application. This paper outlines the considerations that we made when selecting the challenge, and discusses how we believe it will encourage collaboration. It presents the software application that was selected for the challenge in 2019–2020, discusses the practical set up of the challenge, and briefly reports on the received solutions and an online workshop where the different solutions were presented.

1 Introduction

Over the last 20 years, enormous progress has been made in the area of program verification [10]. This progress can be witnessed for example by the development of large, non-trivial case studies, such as the verification of the TimSort implementation in the shipped Java libraries [9], or the parallel nested depth first search [23]. Another evidence of the progress of program verification tools are the outcomes of program competitions, such as VerifyThis [12–16] and VSComp [8,18], where we see a steady increase in the complexity of challenges that have been posed (and solved). However, program verification competitions encourage competition rather than collaboration, and moreover they always impose a strict time constraint, ranging from 90 min per challenge (VerifyThis), to a time-span of 2 days for 4 challenges (VSComp). Thus, though program verification competitions are very useful, given the way that they currently are executed, they do not give a full account of what can be achieved with deductive program verification techniques in general. And in particular, they do not address the question whether deductive software verification techniques are suitable for realistic, industrial-size software.

W. Ahrendt et al. (Eds.): Deductive Software Verification, LNCS 12345, pp. 246–260, 2020.
https://doi.org/10.1007/978-3-030-64354-6_10

Moreover, we see that program verification tools are mainly developed in isolation, i.e., every tool implements its own techniques without benefitting from results obtained by other tools. We feel that this situation needs be changed: every program verification tool has its own strengths and weaknesses, and applying them on large real-world examples can be hampered by these weaknesses. Thus, these weaknesses need to be addressed, and rather than doing this for each verification tool individually, we believe that this should be done by collaboration and exchange.

Therefore, we asked ourselves what could be achieved in the area of program verification if (a) we as the program verification community collaborated and (b) the time constraints were removed? To answer this question, and in particular to encourage more collaboration within the program verification community, we have launched the *VerifyThis Collaborative Long Term Challenge*. The idea behind this challenge is to propose a single, large, industrially-relevant software application, which can benefit from formal verification, but that is too large and complex for a single verification tool. It is our hope that people contributing to this challenge all verify different aspects of the application, and that they exchange results, in order to verify together (almost) all relevant aspects of the application. We hope that the challenge will be of interest also to the larger program verification community, using techniques such as (bounded) model checking, static analysis and symbolic evaluation, and not just deductive software verification. Section 4.2 lists a variety of verification missions that show that the chosen challenge can be attractive to many communities. It is our belief that if the program verification community combines forces, we will be able to show that program verification can produce relevant results for real systems with acceptable effort.

This paper describes the set up of the VerifyThis Collaborative Long Term Challenge. In particular, we discuss the considerations that we took into account when selecting the challenge (for 2019–2020, the selected challenge is the verification of the OpenPGP Key Server), the overall set up of the challenge, and the actions and measurements that we have taken to actually foster collaboration. The main goal of this paper is to document and outline our considerations when setting up the challenge, and to help future challenge developers with their selection process.

Originally, it was planned that the long term challenge would end with a presentation session during the VerifyThis competition event. We briefly report on the five solutions that were handed in, on an online workshop that was held in April 2020 in which the different (partial) solutions to the challenges have been presented, and on lessons learnt.

The remainder of this paper is organized as follows. Section 2 discusses some earlier long term verification challenges, and compares their goals and set up with the current one. Section 3 then discusses the considerations that we took into account for selecting the challenge, and how this challenge can be used to encourage collaboration within the verification community. Section 4 discusses the particular challenge selected for 2019–2020, while Sect. 5 discusses the practical set up of

the challenge, and the measures we took to foster collaboration. It reports on the submitted solutions and experiences from the online workshop.

2 Related Community Challenges

We are not the first to propose a community challenge, or grand challenge. As stated by Bicarregui, Hoare and Woodcock [5]:

> Grand challenges have a long history. From the problem of longitude in the 18th century, through Hilbert's programme for 20th-century mathematics, to the space race of the 1960s, grand challenges have led to considerable innovation and have accelerated engineering and technological advancement towards their goal.

This section discusses some earlier grand challenges in the area of program verification.

The Steam Boiler Case Study [1,2] is the first competition of formal program specification and development methods that we are aware of. Originally, this challenge was proposed to the participants of a 1995 Dagstuhl seminar. The challenge had a competitive character: its intention was to challenge the formal specification community to apply their methods to a non-trivial non-academic case study of a steam boiler control system. The organizers gave a lot of freedom to the participants in how they wanted to address the challenge:

> We deliberately abstained from imposing any specific constraints on the expected solution. The idea was not to exclude any approach and to permit each method to be shown at its best, be it by providing a formal requirement specification, an architectural design, a sequence of stepwise refinements, an executable program or an analysis and proof of behavioural properties one wants to guarantee for the system.

Interestingly, the challenge proposers noticed that the internet helped tremendously to improve communication between the challenge participants, and to ensure that the competition is conducted on a universal scale. When preparing our challenge, we realized that enabling an efficient and lively communication between challenge participants is still critical for its success.

In 2000, the *Mondex case study* was presented [25]. This case study is a bit different: originally it was not designed as community challenge, but rather as a case study for the formal modelling, specification and verification of a smart card electronic cash system. The original case study was developed in Z, but quickly it was picked up as an interesting case study by other groups working on verification of smart card applications. The Mondex case study has solutions in Z, KeY, KIV, Alloy.

The *Verifying Compiler: A Grand Challenge for Computing Research* was proposed by Hoare [11] in 2003. This grand challenge eventually led to a repository of verified software [5]. In his original proposal, Hoare identified a set of criteria that distinguish a *grand challenge* in science or engineering from other research problems:

> A grand challenge represents a commitment by a significant section of the research community to work together towards a common goal, agreed to be valuable and achievable by a team effort within a predicted timescale.

We feel that much of the spirit of Hoare's grand challenge is similar to ours. However, Hoare's grand challenge was even larger than ours, and did not focus so much on one particular application.

In 2007, Joshi and Hoare proposed a challenge to build a *verifiable filesystem* [17]. They propose this smaller challenge as a stepping stone towards the original verifying compiler grand challenge. They identified this particular challenge because:

> [it was of] sufficient importance that successful completion of the mini challenge would have an impact beyond the verification community.

Interestingly, in their paper they also describe other challenge options that they considered, such as the verification of the Linux Kernel, but which they discarded because its complexity and size seemed too large for a reasonable challenge.

3 Challenge Selection

One of the most important aspects of the VerifyThis Collaborative Long Term Challenge is the selection of the software system serving as the challenge. We identified various criteria that we believe would attract the interest of the verification community, and also carefully considered how the challenge could pave the way for potential collaboration between researchers working on different formal verification techniques.

3.1 Criteria for Challenge Selection

We composed the following list with properties that we consider that a suitable target application for the proposed kind of challenge should possess:

- The application should be a *real piece* of software; possibly part of a relevant production system. This would make it attractive for people to participate: it will give them the satisfaction of contributing to a relevant problem given that their verification efforts will concern the properties of a real software system.
- The application should be *open source*: researchers must be able to publish adaptations, modifications, and analyzes without restrictions.
- Even though the application should be written in a real-world language (and be executable), the involved challenges should be of a *language-independent nature*, such that it can be convincingly transferred to other similar programming languages, and participation in the challenge is not restricted to people that work on a program verification tool for exactly the right language.

- The application should be easily *decomposable*, i.e., it should be possible for people to focus on only a part of the complete application, without the need to fully specify and verify the complete application. We believe that decomposability is also essential to enable collaboration.
- The application should be real, but *not over-complex*, i.e., it should be possible for participants to get a good overall understanding of the application without too much effort. After that, they can concentrate on those parts of the application that they wish to verify.
- The *core functionality* of (at least part of) the application should be simple and *well-understood*.
- It should be possible to attack the verification of the application at *different levels*: a participant might start with a highly simplified version of the code and then refine this into a more realistic version. It should also be possible to first concentrate on the key characteristics, and then later extend it in different directions, for example considering error handling or performance optimizations.
- It should be possible to point out a *varied collection of relevant and interesting aspects* in the project that people could try to verify, such as:
  - algorithmic properties, e.g., finding a crucial loop invariant;
  - optimization-related properties, e.g., preservation of correctness when a cache is used, the application is optimized for speed etc.;
  - heap shape specifications and suitable framing conditions;
  - runtime safety; i.e., absence of runtime exceptions (to attract the automatic verification community)
  - concurrency-related properties;
  - exception handling;
  - bounded loops and bit arithmetic (to attract bounded checkers); and
  - behavioural protocols (to attract modelling community).

  Importantly, the aspects to verify should not feel artificial, i.e., they should be related to real reliability aspects of the code.

Candidates. In our search we considered several software applications as candidate challenges. The *GNU Multiple Precision Arithmetic Library (GMP)* is a library for infinite-precision arithmetic. Together with the *GNU Scientific Library (GSL)* it provides a feature-rich set on mathematical algorithms, which makes it highly relevant and widely used. Both were not further investigated because they do not contain any concurrent algorithms. The same reasons are also valid for *Eigen* – a C++ library for linear algebra. Boost libraries, like *Graph* or *Parallel*, were declined because of their complexity and their heavy use of template programming, which renders them very C-specific. We also considered to propose a collection of algorithms as provided by a standard text book, like [24]. Such algorithms are practical and important, but we felt that they were not realistic enough. Probably such algorithms are better suited for the on-site VerifyThis program verification competition (i.e., as a 90 min challenge). Cryptographic algorithms are highly relevant in daily use. Therefore we looked at *Bouncy castle* – a library providing cryptographic implementations for the

Java Crypto API. However, cryptographic algorithms are not easy to specify, and their formal verification requires a substantial amount of mathematical reasoning. Finally, we also considered several libraries for distributed computation, such as Apache Hadoop and Thrill, but for these libraries we felt it was unclear what would actually be the desired properties to verify.

3.2 Encouraging Collaboration

As mentioned above, the VerifyThis Collaborative Long Term Challenge has been particularly designed with the goal in mind to incite collaboration between the participants. In fact, we feel that to further advance the field of formal program verification, more collaboration between different techniques and tools is essential.

Two unrelated tools can seldom make use of each other's results easily, mostly because combining fundamentally different techniques is inherently difficult. The common specification languages Java Modelling Language (JML) [19] and the ANSI/ISO-C specification Language (ACSL) [4] are designed to be applicable in different verification scenarios (e.g., deductive verification and runtime assertion checking). Even for these limited scopes, coming up with an indisputable, common language semantics is difficult (the runtime semantics of ACSL differs from that for deductive verification [21], and verification tools differ in how they verify JML specifications [6]).

This challenge has the potential for the program verification community to investigate ways of how results of one verification endeavour can be used in another. To achieve this goal, it is important that the challenge is formulated on a very general level that is not restricted to a particular sub-community. Section 4.2 reports on some relevant questions that exemplify that variety of different property types that can be specified and verified using different approaches. The more concrete situation should allow collaborating partners to identify what guarantees the results obtained in a different formal system imply in their formalism, and how these results can be encoded in their verification context.

We illustrate the potential for collaboration by a hypothetical example. An automatic static analysis may be able to infer that a module of the software only changes a number of memory locations. The analysis can produce results quickly without much specification overhead set since it answers a specialized question. This allows one to focus during the verification of heavyweight functional specifications on the already intricate interactive task to craft the relevant auxiliary specifications (contracts and loop specifications) that are usually required on such occasions. The results of the scalable framing analysis can be used as additional assumptions in the functional verification making it more precise.

4 The OpenPGP Key Server

As the target for the 2019–2020 challenge, we chose a modern public key server called HAGRID[1] This section introduces this application, and its verification missions.

When using public key encryption and signatures in e-mails, one challenge is to obtain the public key of recipients. To this end, public key servers have been installed that can be queried for public keys. The most popular[2] public key server OpenPGP was recently shown to have severe security flaws. There was no protection on who could publish a key for an e-mail address and no protection on the amount of data published. This opened the gate for a broad range of dangerous attacks such as the ones presented at CVE-2019-13050[3], or the ones described in the blog post of the HAGRID's developers. Moreover, the old key server software SKS did not conform to the General Data Protection Regulation (GDPR) and had performance issues.

As a consequence, the OpenPGP community decided to implement a new server framework that manages the access to public keys. The new official server is called HAGRID, it is open source[4], and it is already in production. HAGRID is written in the programming language Rust and comprises some 6,000 lines of code in total[5].

HAGRID represents a modern piece of code, with both an acceptable size and complexity, which makes it an excellent challenge application. Furthermore, it is currently in use by many pervasive applications such as *GPGTools, Enigmail, OpenKeychain, GPGSync, Debian* and *NixOS*, which implies that its verification will have an important impact towards security and efficiency of software that is in use daily. What is more, its architecture comprehends many interesting aspects for verification, such as database consistency, concurrency, efficiency and functional correctness, scoping a wide range of interests of the software verification community. This also makes it suitable to encourage interaction between members of the verification community, which can attack complementary verification problems over the single challenge program.

4.1 The Verifying Key Server

While HAGRID is the *reference implementation*, and the final goal is to verify it, we decided to define a more general *verifying* key server. This allows us to have a less restrictive starting point for verification by abstracting from HAGRID's particular implementation decisions. It also establishes clear bases for abstraction decisions to be made at the time of verifying with specific tailored tools.

[1] See https://sequoia-pgp.org/blog/2019/06/14/20190614-hagrid/.
[2] It is the default server used by the Thunderbird public-key engine *Enigmail* for instance.
[3] See https://access.redhat.com/articles/4264021.
[4] Available at https://gitlab.com/hagrid-keyserver/hagrid, (2020-04-29).
[5] Not including the underlying web framework or GPG library code.

The server is essentially a database that allows users to store their public key for their e-mail address, to query for keys for e-mail addresses and to tracelessly remove e-mail-key pairs from the database. To avoid illegal database entry and removal actions, confirmations are sent out to the e-mail addresses of issuing users upon an addition or removal request.

The server possesses a web frontend which accepts requests from users or via restful API. It additionally possesses a connection to a database from which it reads key-value pairs and writes to it, and a channel for sending e-mails. Figure 1 presents a schematic overview of the architecture. The key server can be separated into three components: the *webserver* (frontend), the *key manager* (backend), and the key *database*. At the core of the server, there are four operations that can be triggered from outside the server via HTTP-API-requests to the web frontend. The operations are:

Request adding a key. A user can issue a request for storing a key for a particular e-mail address. To avoid that anybody can store a key for someone else's e-mail address, the key is not directly stored into the database, but stored intermediately. The user retrieves a confirmation code via the given e-mail to verify the specified address. Only once the confirmation code is activated, will the address be actually added to the database.

Querying an e-mail address. Any user can issue a request for learning the key(s) stored with a concrete and verified e-mail address. Unlike on the old public server, queries for patterns are not allowed. Public keys that have been (verified) removed or have not yet been confirmed must not be returned in queries.

Request removing a key. The user can request the removal of the association between a key and an e-mail address. The process begins with the confirmation via the e-mail address: The user enters one of their previously confirmed addresses. The server sends an e-mail to this address containing a link. Behind this link, there is a website that allows the removal of the key's association.

Confirming a request. Additions and removals are indirect actions. Instead of modifying the database directly, they issue a (secret and random) confirmation code. Confirmation of the code is performed using this operation. If the provided code is one recently issued then the corresponding operation (addition/removal) is finalized.

The challenge we propose focuses on the key manager component of the server. This is a program that must provide implementations for the operations outlined above.

Nevertheless, multiple extensions to the *verifying server* can impose a bigger verification challenge for the participants. We encouraged participants to also look at this. One would maybe want to verify the database or the REST-API. One would usually want to abstract from the programming language unless the verification tool is prepared for it, but on the other hand may decide to be as faithful as possible to HAGRID's implementation, and thus as close as possible to the real code. Other verification possibilities go in the direction of improving the

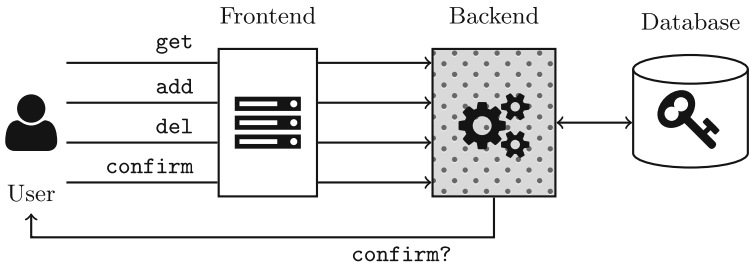

Fig. 1. Schematic of the key server architecture

reference implementation, for instance by increasing the server's performance by using more sophisticated search-friendly data structures.

4.2 Missions

The HAGRID server is a complex software system which has been chosen as the target for the long term challenge since it provides verification challenges on many different levels of *abstraction*, *complexity* and regarding many different *requirement aspects* (e.g., safety, security, performance properties). This section identifies different verification tasks (called missions) for the different abstraction levels and aspects.

One design decision of the challenge is to leave it to the participating teams on which actual artefact(s) they contribute a verification result. Depending on the addressed mission, different artefacts suggest themselves:

1. the existing production HAGRID reference implementation in Rust;
2. a new implementation (in any programming language) of the core functionality satisfying a basic set of natural language requirements available on the challenge website; or
3. a suitable abstraction from the code level to model a particular aspect (the protocol, memory, key handling, ...).

This also accommodates for the fact that contemporary verification approaches are heterogeneous in the languages and specification techniques they support. By opening the choice of language and implementation layout, the scope of the challenge has considerably broadened. Moreover, it allows also for an adjustable degree of algorithmic complexity: The data structures used in a re-implementation may range from rather simple ones (e.g., two arrays containing the email addresses and the associated keys), over more complex (e.g., a hash map with concurrent data access) up to an implementation that satisfies real-world requirements regarding, e.g., efficiency and memory consumption. This allows the contributing teams to choose their initial level and also to advance in the course of the challenge.

Moreover, an implementation may make use of underlying (provenly or assumedly correct) libraries and middleware layers (e.g., by assuming method contracts/function summaries and (object) invariants).

The first suggested mission of the VerifyThis Collaborative Long Term Challenge is the least specific task and allows many formal code analysis tools to participate. Verification tools that run fully automatically (in particular the participating tools in the SVCOMP[6] competition series) are particularly well suited to contribute solutions to this challenge and to show that results can be obtained without further user input. For this foundational verification question, no formal (full) specification is required.

Mission 1 (Safety). *Verify that the chosen implementation of the key server does not exhibit undesired runtime effects (e.g., no runtime exceptions in Java, or no undefined behaviour in C).*

Traditionally, the challenges in VerifyThis on-site competitions are more heavyweight, with concrete application-specific functional requirements on the used data structures that go beyond safety conditions and assertion checking. They strive to establish properties that require a logical formalisation against which the code needs to be verified. Unlike Mission 1, this *functional* verification requires knowledge about what the system has to compute, i.e., a specification must be provided. Depending on the complexity of the code and specification (and verification technique), the verification may then run automatically, or (in many cases) requires some form of user guidance (on top of the specification).

Mission 2 (Functionality). *Formalize the natural language functional requirements from the challenge description as formal specifications for the core operations. Verify that the implementation of the operations satisfies your formalisation.*

One example for a functional property of the key server is that if an e-mail address is queried, a key stored for this e-mail address is returned if there is one in the database.

Typically, especially for imperative (object-oriented) programming languages, functional verification is performed modularly by specifying *contracts* for each function (method) according to the design-by-contract paradigm and then proving them correct individually.

Other formal method traditions focus less on the operational effects of individual functions in form of contracts, but allow one to model the evolution of the entire system over time using a temporal or state-based formalism. This is particularly the case for model checking approaches where properties of the interaction protocol can then be analyzed on an implementation-independent level.

Mission 3 (Protocol). *Specify the temporal protocol behaviour of the key server. Identify relevant temporal properties for the key server and prove them satisfied by the protocol.*

[6] See https://sv-comp.sosy-lab.org.

Mission 3 deals with somewhat different properties than functional verification. It is a good candidate for a collaborative verification effort in the sense of Sect. 3.2: One team of participants may prove properties of the protocol using an approach designed for that purpose (e.g., model checking), whereas another team verifies that the implementations of the operations adhere to their abstractions. Tools have complementary strengths: Model checking cannot go into implementational details on the programming language level, and for deductive verification, encoding protocols is cumbersome, error-prone and little efficient.

There is a school of formal modelling techniques that allows one to model systems on a rather abstract level, but with mathematical rigour. These models can then be formally refined into more concrete detailed versions, and eventually into executable code. Prominent representatives are (Event-)B, Abstract State Machines or Z.

Mission 4 (Refinement). *Encode the natural language requirements from the challenge description as a mathematically rigour system model. Refine the abstract model in one or more steps into an executable program.*

Again this mission provides great potential for collaboration if abstract system verification meets program verification.

Not every system property can be formalised as functional property (Mission 2); for instance privacy properties require different verification techniques.

Mission 5 (Privacy). *Specify and prove that the key server adheres to privacy principles. In particular: (a) only exact query match results are ever returned to the user issuing a query, and (b) deleted information cannot be retrieved anymore from the server.*

One concrete example is that if an e-mail address has been deleted from the system, no information about the e-mail address is kept in the server. Classical non-interference analyses (from type-theoretical and dependency-graph-based statistical analyses to deductive analysis) allow formal methods to be applied to this mission.

There are more security-related properties for the key server since that has to produce and distribute random confirmation codes. A key server should be analysed w.r.t. cryptographic-related questions, too.

Mission 6 (Randomness). *Prove that any created confirmation code is (a) randomly chosen (i.e., that every string from the range is equally likely), (b) cannot easily be predicted, and (c) is never leaked, except as the return value of the issuing operation.*

Another field of interesting properties (that also have been addressed in VerifyThis onsite events recently) are questions around concurrency. They often are related to the actual implementation.

Mission 7 (Thread safety). *Specify and verify that your implementation is free of data races, where data races arise when concurrent processes try to access*

a common memory location simultaneously. For concurrent applications, also absence of deadlock and livelock are important requirements.

The key server as a software system providing a critical infrastructure also has relevant properties concerning its availability, its resource consumption or its performance. They are worth being formally analysed and their analysis can likely benefit from guarantees obtained during analyses of functional aspects.

Mission 8 (Non-functional properties). *Identify, specify and verify non-functional properties of the key server concerning worst case execution times and worst case memory consumption of the operations (or other relevant non-functional properties).*

5 Realisation

Finally, we discuss some practical aspects of how we have set up the challenge and report on the experiences that we gained after the submission of the solutions.

5.1 Practical Set-Up

The VerifyThis Collaborative Long Term Challenge was launched in August 2019, and submitted solutions were due end of March 2020. Participants were invited to present their results during the on-site VerifyThis program verification competition (co-located with ETAPS in April 2020). In addition, the plan to compose a special issue with (partial) solutions to the challenge was announced.

Two crucial aspects of the long-term challenge were (1) to get people started, and (2) to ensure that people continue working on the challenge. To achieve this, we had set up several means of communication.

First of all, we have the website https://www.verifythis.github.io, which serves as the entry point to the challenge and is the main source of up-to-date information. The website has several functions. First of all, it informs the community about the challenge and its current state by showing (preliminary) results and news. It also provides information to the participants, such as an overview of the other participants and artefacts that participants have made available for reuse. We decided for a website hosted by Github, as this makes it easy for participants to contribute to the website, either via the issue tracker or a via pull request. We tried to encourage the participants to use this public repository by themselves.

In addition, we also created a mailinglist[7], such that both organisers and participants could communicate with each other directly. We had considered using a collaborative development platform such as Slack, but decided not to do this, because we felt that most of our potential participants would prefer communication by email.

[7] verifythis-ltc@lists.kit.edu.

5.2 Submissions

At the end of the submission period, we had received five contributions from different teams using different verification approaches and tools. Three of them focused on functional specification (mostly missions 1 and 2), two considered security-related aspects (mission 5). Short papers describing the solutions can be found in the informal proceedings [20].

Submissions for functional properties were written in different programming (and specification) languages and verified with different deductive verification tools (SPARK [22], Why3 [7], and Java+JML with KeY [3]). All specifications followed similar ideas: They specified the functional key server interface using contracts that formally capture the effects of requests on the database (represented by some form of ADT). The Why3 solution targetted a file-based database backend, whereas the other two solutions modelled it as an in-memory database.

The other two submissions provided a security testing framework for the keyserver based on history traces (in Scala), and a formulation of information flow security properties with declassification in a variant of separation logic for security properties. In particular, revocation of key entries was identified here as an interesting challenge for non-interference approaches.

5.3 Experiences

We observed that the communication channels were used less frequently than we expected. We assume that the mailing list was perhaps not the best medium for the purpose. Moreover, new collaboration ideas did not spark between different approaches or tools during the offline period. We summise that a stronger incentive for collaboration could be given by additional meetings during the runtime of the challenge. These could either be real-world meetings or could be performed online.

Since the workshop originally planned in late April 2020 had to be postponed (due to the Covid-19 pandemic), we set up an online meeting – close to the original workshop date – at which the participants and interested parties could discuss the challenge, their solutions and perhaps identify synergies and common ideas.

The half-day online event received considerable attention (over 30 participants; more than would have been expected for the online event). Similarities and differences between the approaches were identified and discussed. The contributors agreed that the LTC should not be called terminated, but that further work and collaboration would improve their work. As a concrete idea for cooperation, it was identified that explicitly modelling the history of requests can be adapted in deductive specifications from the security testing approach.

The ETAPS conference (and with that its workshop) was postponed for several months. The organisers and participants agreed to meet again when ETAPS is held for an updated report on the collaboration and solutions to the challenges.

6 Conclusion

We received many positive reactions from the community since the start of this challenge, and we hope that this will also lead to interesting and unexpected verification outcomes. Ultimately, we hope that the challenge will bring the formal verification community a step closer to its ultimate goal: the usage of formal analysis in the daily software development process, by providing better insights into the obstacles and potentials for the use of formal techniques, and therewith helping the participants to further improve their approaches and tools.

In addition to tool improvements, we also hope that the VerifyThis Collaborative Long Term Challenge will diminish the gap between different formal verification approaches, and will foster more collaboration within the verification community.

References

1. Abrial, J.-R., Börger, E., Langmaack, H.: The steam boiler case study: competition of formal program specification and development methods. In: Abrial, J.-R., Börger, E., Langmaack, H. (eds.) Formal Methods for Industrial Applications. LNCS, vol. 1165, pp. 1–12. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0027228
2. Abrial, J.-R., Börger, E., Langmaack, H. (eds.): Formal Methods for Industrial Applications. LNCS, vol. 1165. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0027227
3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, LNCS, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
4. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: Acsl: Ansi/iso c specification language. Reference manual. http://www.frama-c.com/download/acsl.pdf
5. Bicarregui, J., Hoare, C.A.R., Woodcock, J.C.P.: The verified software repository: a step towards the verifying compiler. Formal Asp. Comput. **18**(2), 143–151 (2006). https://doi.org/10.1007/s00165-005-0079-4
6. Boerman, J., Huisman, M., Joosten, S.: Reasoning about JML: differences between KeY and OpenJML. In: Furia, C.A., Winter, K. (eds.) IFM 2018. LNCS, vol. 11023, pp. 30–46. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98938-9_3
7. Filliâtre, J.-C., Paskevich, A.: Why3 — Where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
8. Filliâtre, J., Paskevich, A., Stump, A.: The 2nd verified software competition: experience report. In: Klebanov, V., Beckert, B., Biere, A., Sutcliffe, G. (eds.) Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, Manchester, United Kingdom, June 30, 2012. CEUR Workshop Proceedings, vol. 873, pp. 36–49. CEUR-WS.org (2012)
9. de Gouw, S., de Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying openjdk's sort method for generic collections. J. Autom. Reasoning **62**(1), 93–126 (2019). https://doi.org/10.1007/s10817-017-9426-4

10. Hähnle, R., Huisman, M.: Deductive software verification: from pen-and-paper proofs to industrial tools. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 345–373. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_18
11. Hoare, C.A.R.: The verifying compiler: a grand challenge for computing research. J. ACM **50**(1), 63–69 (2003). https://doi.org/10.1145/602382.602403
12. Huisman, M., Monahan, R., Mostowski, W., Müller, P., Ulbrich, M.: VerifyThis 2017: A program verification competition. Technical Report, Karlsruhe Reports in Informatics (2017)
13. Huisman, M., Monahan, R., Müller, P., Paskevich, A., Ernst, G.: VerifyThis 2018: A program verification competition. Technical Report, Inria (2019)
14. Huisman, M., Monahan, R., Müller, P., Poll, E.: VerifyThis 2016: A program verification competition. Technical Report TR-CTIT-16-07, Centre for Telematics and Information Technology, University of Twente, Enschede (2016)
15. Huisman, M., Klebanov, V., Monahan, R.: VerifyThis 2012. Int. J. Softw. Tools Technol. Transf. **17**(6), 647–657 (2015)
16. Huisman, M., Klebanov, V., Monahan, R., Tautschnig, M.: VerifyThis 2015: a program verification competition. Int. J. Softw. Tools Technol. Transf. **19**(6), 763–771 (2017)
17. Joshi, R., Holzmann, G.J.: A mini challenge: build a verifiable filesystem. Formal Asp. Comput. **19**(2), 269–272 (2007). https://doi.org/10.1007/s00165-006-0022-3
18. Klebanov, V., et al.: The 1st verified software competition: experience report. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 154–168. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_14
19. Leavens, G.T., Baker, A.L., Ruby, C.: JML: a java modeling language. In: Formal Underpinnings of Java Workshop (at OOPSLA'1998), pp. 404–420. Citeseer (1998)
20. Huismann, M., Monti, R.E., Ulbrich, M., Weigl, A. (eds.): VerifyThis Long-term Challenge 2020. In: Proceedings of the Online-Event (Mai 2020). https://doi.org/10.5445/IR/1000119426
21. Maurica, F., Cok, D.R., Signoles, J.: Runtime assertion checking and static verification: collaborative partners. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11245, pp. 75–91. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03421-4_6
22. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press, Cambridge (2015). https://doi.org/10.1017/CBO9781139629294
23. Oortwijn, W., Huisman, M., Joosten, S., van de Pol, J.: Automated verification of parallel nested DFS (2019), submitted
24. Sedgewick, R., Wayne, K.: Algorithms, 4th edn. Addison-Wesley, Amsterdam (2011)
25. Stepney, S., Cooper, D., Woodcock, J.: An Electronic Purse: Specification, Refinement and Proof. Technical Report PRG-126, Oxford University Computing Laboratory, July 2000. http://www.cs.kent.ac.uk/pubs/2000/1527