

Methods of Formal Software Verification in the Context of Distributed Systems

Anna Fatkina, Oleg Iakushkin(✉), Dmitry Selivanov, and Vladimir Korkhov

Saint-Petersburg State University, St. Petersburg, Russia
o.yakushkin@spbu.ru

Abstract. This paper discusses several formal verification instruments and compares them. These tools are Isabelle/HOL, Coq, Verdi, and TLA+. All of them are developed for automatic verification of distributed systems. However, there are a number of differences in implementation and application. Verdi provides an effortless way of implementation to verify some distributed systems. Isabelle/HOL and Coq, on the other hand, can solve a wider range of tasks. These provide a low-level interface and require programming skills. TLA+ allows the user to communicate in pseudocode-like language as well as per algorithm implementation in TLA+/PlusCal. It is the most versatile tool for formal verification which is considered in this paper.

Keywords: Formal verification · Proof assistant · TLA+ · Isabelle/HOL · Coq · Verdi · Distributed systems

1 Introduction

In the research area of distributed ledger technologies, the inevitable question is the formalization and verification of developed systems. Any inaccuracy could be exploited as a vulnerability. Therefore, the formal verification of such systems requires special treatment. There are a number of tools for automatic software verification and formal description of models. In this work, a subset of these instruments are considered, that show effectiveness in the formalization of consensus algorithms, such as Casper in Isabelle and Coq, Paxos in TLA+ and Raft in Verdi [12,13,17–19].

This paper briefly describes how to use the considered tools and shows and shows several examples. It also presents a comparison of various tools and describes their suitable use-cases.

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11620, pp. 546–555, 2019.
https://doi.org/10.1007/978-3-030-24296-1_43

2 Formal Verification Tools

The following tools for formal proofs of mathematical statements are considered in this paper:

- Isabelle/HOL,
- Coq,
- Coq-based Verdi, and
- TLA+.

For Isabelle and Coq an Emacs interface called Proof General [1] is provided, which is distributed under GNU General Public License. The system TLA+ is delivered with the IDE called TLA+ Toolbox [2]. Both IDEs have user-friendly

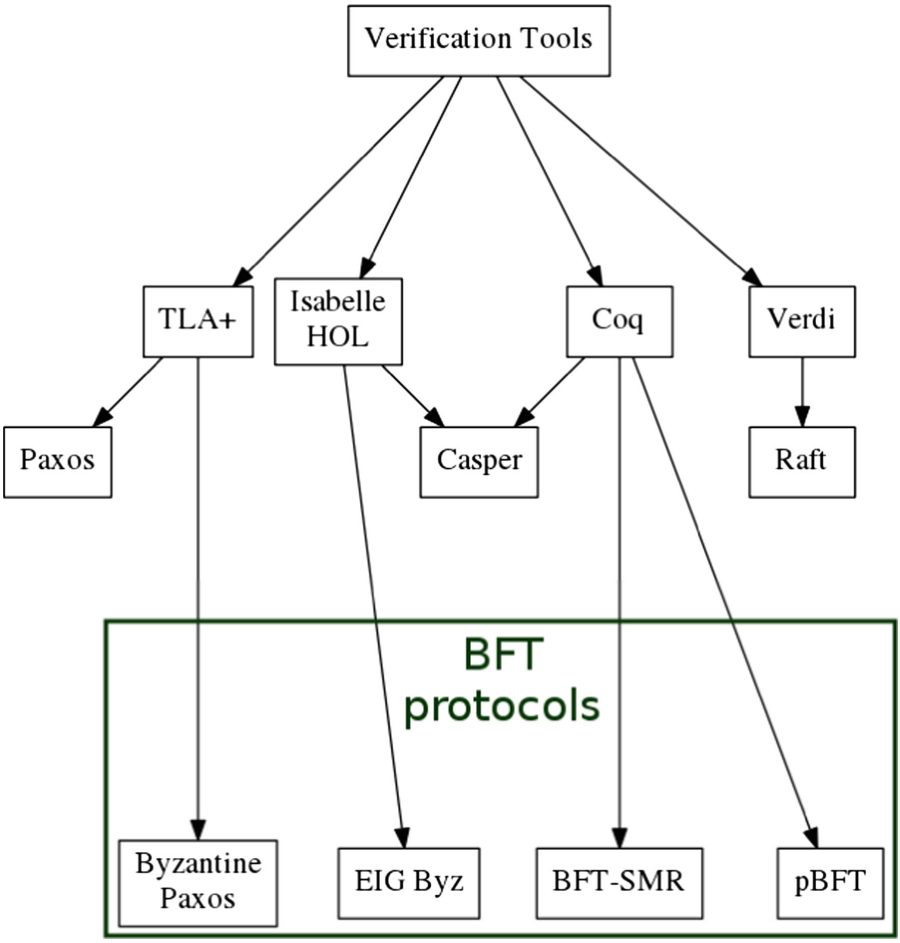

Fig. 1. Verification tools usage for verification of consensus algorithms.

GUIs and user guides. Verdi is a library for modeling distributed systems [3]. It has a command line interface.

Some cases of usage of considered tools for consensus algorithm verification are presented in Fig. 1.

For the Casper protocol there is implementation in both Isabelle/HOL and Coq. Verdi was used for Raft protocol only. To get more information about verification Byzantine Fault-Tolerant protocols, see [22,23].

2.1 Isabelle/HOL

Isabelle/HOL is a proof assistant, which itself is part of the generic system for logical formalism implementation Isabelle [8,9]. HOL stands for Higher-Order Logic. It is a language for theorem descriptions, their according proofs as well as formulas and other mathematical constructions. It also includes various automated proof tools. Basic data types such as `natural`, `real` and `complex` numbers, `bool` and `integer` are supported out of box. The user can also extend existing data types or create new individual types.

There are also more complex data types. For example, `pair`, which is an ordered pair of two elements. Access to the single elements is provided by the `fst()` and `snd()` methods, which return the first and second part of the pair, respectively. `Tuple` is a generalization of pair and contains one to seven ordered elements. A tuple can also be interpreted as a set of nested pairs, from right to left. The degenerated version of the container of ordered numbers called `unit` contains a single element. New data types can also be defined by the user. Listing 1.1 provides an example of this for data using the `record` keyword. Here, a `point` type with two integer fields, `Xcoord` and `Ycoord`, is defined.

```
record point =
    Xcoord :: int
    Ycoord :: int
```

Listing 1.1. Isabelle/HOL definition of `point`

The Isabelle/HOL notation includes a predefined data structure called `list`. It is a sequence of values of the same type or might be empty. For a `list`, concatenation operations are defined as well as append, reverse and many other instructions.

Isabelle/HOL operates on the concept of `theory`. It is described in a separate file and may include other theories previously described. Mathematical statements are defined inside the `theory`, e.g. `terms` and `formula`. Here, `formula` is a specific `term` with `bool` return value. Isabelle/HOL also supports lambda expressions. If `theory B` is included in `theory A`, then one could use statements in `theory A` which were proven in `theory B`.

Theorems and lemmas are described by the keywords `theorem` and `lemma` and are interchangeable. They indicate just the importance of approval. The content of the theorem is described in quotes, and its proof ends with the keyword `done`. Listing 1.2 presents an example of a simple lemma.

```
lemma app_Nil2 [simp]: ``xs @ [] = xs"
____apply(induct_tac xs)
____apply(auto)
done
```

Listing 1.2. Isabelle/HOL lemma example

It says that if adding a blank `list` to another one, it does not change the content of the first. This lemma is proved by the use of built-in automatic methods.

Isabelle/HOL has a number of automatic proof methods. The `simp` method tries to solve a problem by simplification, `arithm` tries to solve a problem as an arithmetic formula and the `auto` method looks for suitable resolve methods automatically. More complex theorems need step-by-step proofs of intermediate expressions. The general scheme of any proof consists of two steps: breaking a global goal into simple subgoals and then proving them according to predefined expressions or previously proved ones. For proofs of intermediate subgoals, one can use different introduction, elimination, destruction, and forward rules. Conjunction, disjunction, implication, modus ponens can act as rules. Isabelle/HOL's most effective and fast proof method is `blast`. It can even resolve complex expressions by consistently applying the rules automatically. For HOL there is a set of freely available theories. Anyone can submit their own theories to a public theory base [10].

2.2 Coq

The formal verification tool Coq [4,5] is similar to Isabelle/HOL. It also contains methods for describing mathematical statements but differs in syntax as well as in a wider range of predefined types and containers. The Coq proof assistant supports basic data types, such as `natural` and `bool`. There is also an extended type which is described by using the `option` keyword. However, unlike Isabelle/HOL, Coq also has types

- `sum(A, B)` and `prod(A, B)`, disjoint sum and product respectively,
- comparison types, (less, equal, greater),
- `ascii` is an eight-bit character
- `string` is a set of eight-bit characters.

More complex data types, e.g. real and rational numbers, are described in separate libraries, `Reals` and `QArith`, respectively. For those, there are also separate libraries of functions that act on these numbers like integration, trigonometry, the theory of series and sequences for real numbers.

As in Isabelle/HOL, in Coq it is possible to define your own data types using the `Record` keyword. In Coq, also classes are supported for which methods are defined for the according instances.

The `list` and `stream` containers defined in Coq are similar. Their description is in the `Coq.Lists` library. List is the limited sequence of items. Concatenation methods are defined as well as methods that return the length of the list and pointers to the first and last elements. `Stream` contains an infinite sequence of elements.

The keyword `Function` specifies the function. An example is shown in Listing 1.3. Here a function called `sum` is created, which adds two numbers of the `nat`(natural) type.

```
Function sum (m n : nat) {struct n} : nat := match n with
    | 0 => m
    | S p => S (plus m p)
end.
```

Listing 1.3. Coq example: addition of two natural numbers.

`Theorem` is used to describe the theorem. An example of a description with according proof is shown in Listing 1.4. The body of it is limited to the keywords `Proof` and `Qeq`.

```
Theorem my_first_proof : (forall A : Prop, A -> A).
Proof.
    intros A.
    intros proof_of_A.
    exact proof_of_A.
Qed.
```

Listing 1.4. Coq theorem example.

To perform the proof, it is necessary to include in the consideration of the hypothesis and the variables that are used. This is done using the `intros` keyword. For automatic proof, a number of different tactics can be used. This example presents the `exact` tactic. If T is our goal, let p be an expression for U. `Exact p` applies if T and U are converted into each other.

The choice of proof tactics depends on the complexity of the expression which is to be proved. For example, `reflexivity`, `trivial` and `auto` tactics can be used for trivial proofs. A detailed description of tactics and cases of their use are described in the Coq documentation [6]. A broad set of examples is also provided to demonstrate use [7].

2.3 Verdi

Verdi is a Coq-based framework for verification of distributed systems. It allows to search for errors and check the fault tolerance and correctness of building distributed systems. The user describes the specification and behavior. Verdi then simulates various system failures. The library includes a set of templates, i.e. lock server with proof of safety or liveness. In addition, Verdi provides free access to the implementation of the Raft distributed consensus protocol [14,15].

Nevertheless, the development of Verdi has very little progress now. Such a conclusion can be made on the basis of GitHub statistics [20,21] and the official site Verdi. The latest release was submitted in January 2017. The statistics from Verdi and Verdi Runtime repositories are presented in Tables 1 and 2.

Table 1. Verdi GitHub statistics.

Last commit	January 2017
Active pull requests	0
Merged pull requests	0
Open issues	0
Closed issues	0
Contributors (last 6 months)	1
Commits (last 6 months)	2

Table 2. Verdi Runtime library GitHub statistics.

Last commit	January 2017
Active pull requests	0
Merged pull requests	0
Open issues	0
Closed issues	0
Contributors (last 6 months)	2
Commits (last 6 months)	7

2.4 TLA+

TLA+ operates with modules. Modules are described in files using the `MODULE` keyword [16]. These can be nested, that is, another one can be described inside the body of a module. One can also include modules in each other using the `EXTENDS` keyword.

In TLA+ logical, string type and basic numeric data types are available. However, to access operations on numeric types, it is necessary to connect the corresponding modules.

In addition to single elements, TLA+ also provides more complex types - `tuple` and `structure`. A `tuple` is an ordered set of values and is indicated by double triangular brackets. Several functions are defined in order to work with these: combining tuples, retrieving the first and last elements, determining the length of the tuple, adding one element. The `structure` contains key-value pairs of elements. An example of the structure is shown in Listing 1.5. Here, a structure with two fields a and b defined. Accessing an element is done by key.

```
x == [a |-> 1, b |-> {2, 3}];
x.a = 1;
x["b"] = {2, 3};
```

Listing 1.5. TLA+ structure type

`Set` is one more type. It is denoted by curly brackets and can be of any degree of nesting. Basic set operations are available, such as filtering, fetching a sample, and applying a function to all its elements. Mathematical operators such as union, intersection, the set difference are also available. Functions are defined over sets. An example of the definition of a function that adds elements to a set is given in Listing 1.6.

```
Sum == [x, y \in S |-> x + y]
```

Listing 1.6. TLA+ function that adds elements of set S

Here, one first indicates the name by which the function is accessed, and then the inputs and function body in square brackets.

If necessary, a detailed and consistent description of any algorithm for the model in TLA+ is provided using PlusCal. It is a programming language that is translated into TLA+.

TLAPS (TLA Proof System) is a system that allows users to prove mathematical statements. To describe the theorem, the keyword `THEOREM` is used. The schema for describing it is shown in Listing 1.7.

```
THEOREM Name == <theorem text>
    BY e1, ..., em
    DEF d1, ..., dn
```

Listing 1.7. TLA+ description of theorem

Here $e1, ..., em$ are assumptions and $d1, ..., dn$ are definitions. Assumptions are described using the `ASSUME` keyword.

For the proof, the keyword `BY` is needed. If the proof is obvious and does not require consideration of additional assumptions, then the `BY OBVIOUS` construct can be used.

Also in TLA+ it is possible to use external packages for proving theorems. The corresponding tools that can be used are described in Table 3.

For each of the tactics presented in the Table 3, it is possible to set a time limit by adding to the name tactics `T(time in sec)`. For proof verification using Isabelle with a time limit of 30 s, `BY IsaT(30)` must be specified. The time limit here is the time after which TLA+ either allows the statement to be proved or gives the message that this cannot be done within the framework of the assumptions under consideration.

Table 3. TLA+ tactics.

Command	What does it do
BY Zenon	Calling Zenon with default timeout of 10 s
BY Isa	Calling directly Isabelle with default tactic and timeout
BY IsaM("blast")	Calling directly Isabelle with modified tactic
BY SMT	Call baseline SMT solver with default timeout of 5 s
BY CVC3	Call CVC3 backend with default timeout of 5 s

3 Summary

This paper discusses several systems of automatic verification and proofs of theorems. All of them are similar in functionality, but there are some differences in terms of interface and application.

TLA+ has the simplest interface for proving theorems and its syntax is close to the mathematical description method. Here, the user operates mainly with the terms of sequences, sets, and other mathematical concepts. TLA+ also provides a set of external tools that verify proofs. Among them is Isabelle which discussed in this article as well as other tools. In addition to the high-level interface, TLA+ also includes PlusCal as a detailed description of the algorithms. Thus, TLA+ is the most versatile tool with a simple interface.

Isabelle/HOL is a programming language that allows the user to describe mathematical statements and how they are proved step-by-step. Compared to TLA+, it is a lower level tool. Isabelle/HOL has a wider functionality for describing mathematical problems. Moreover, it is used in TLA+ as one way to prove out of the box. Coq is very similar to Isabelle/HOL. It has the same basic differences in functionality from TLA+ as Isabelle. Nevertheless, there exist also a number of differences in logic and predefined axioms. For a more detailed comparison of Coq and Isabelle/HOL, see [11].

Verdi is a tool for describing and verifying distributed systems and is based on Coq. The tool provides a simple interface for building systems but has a narrow scope. Although this framework is suitable for verification of consensus algorithms, it is not actively developed at the moment.

Table 4. Verification tools comparison.

	High-level interface	Low-level interface	Easy-to-use	Active community
Isabelle/HOL	No	Yes	No	Yes
Coq	No	Yes	No	Yes
Verdi	Yes	Inherit from Coq	Yes	No
TLA+	Yes	Yes	Yes	Yes

A short brief of compared tools is presented in Table 4. Note that all these instruments are suitable for distributed systems description.

4 Conclusion

Verdi is suitable for a generalized description of distributed ledger systems. There is an implementation of verification of the Raft consensus algorithm using Verdi. However, at the moment, the development of the framework is significantly slowing down.

Both Coq and Isabelle/HOL are used to verify the algorithms of the largest distributed ledger platforms. They are powerful tools for working with both mathematical statements and the formal approach to distributed systems. TLA+ is the most versatile tool presented. It combines both an easy-to-use high-level TLA+ language and the strict algorithmic language PlusCal.

Acknowledgements. We are grateful to Dr. Martin Reinhardt for the help to prepare the text of this paper.

References

1. Proof General. A generic Emacs interface for proof assistants. https://proofgeneral.github.io/. Accessed 20 Mar 2019
2. Leslie Lamport: The TLA+ Toolbox. https://lamport.azurewebsites.net/tla/toolbox.html. Accessed 20 Mar 2019
3. Verdi. Formally Verifying Distributed Systems. http://verdi.uwplse.org/. Accessed 20 Mar 2019
4. Chlipala, A.: Certified Programming With Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. MIT Press, Cambridge (2013)
5. The Coq Proof Assistant. https://coq.inria.fr/. Accessed 20 Mar 2019
6. Coq Tactics Cheatsheet. https://www.cs.cornell.edu/courses/cs3110/2017fa/a5/coq-tactics-cheatsheet.html. Accessed 20 Mar 2019
7. Coq Tactic Index. https://coq.inria.fr/distrib/current/refman/coq-tacindex.html. Accessed 20 Mar 2019
8. The Isabelle development team. Isabelle. https://isabelle.in.tum.de/. Accessed 20 Mar 2019
9. Paulson, T.N.L.C., Wenzel, M.: A Proof Assistant for Higher-Order Logic (2013). http://isabelle.in.tum.de
10. Isabelle Archive of Formal Proofs. https://www.isa-afp.org/. Accessed 20 Mar 2019
11. Yushkovskiy, A., Tripakis, S.: Comparison of Two Theorem Provers: Isabelle/HOL and Coq (2018)
12. Palmskog, K., Gligoric, M., Peña, L., Moore, B., Rosu, G.: Verification of Casper in the Coq Proof Assistant (2018)
13. PoS related formal methods. https://github.com/palmskog/pos. Accessed 20 Mar 2019
14. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.: Planning for change in a formal verification of the raft consensus protocol. In: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, pp. 154–165. ACM (2016)

15. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference (USENIXATC 2014), pp. 305–319 (2014)
16. TLA+ User Guide. https://learntla.com/tla/. Accessed 20 Mar 2019
17. Pîrlea, G., Sergey, I.: Mechanising blockchain consensus. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 78–90. ACM (2018)
18. Charapko, A., Ailijiang, A., Demirbas, M.: Bridging Paxos and Blockchain Consensus (2018)
19. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018), pp. 66–77. ACM, New York (2018). https://doi.org/10.1145/3167084
20. Verdi GitHub repository. https://github.com/uwplse/verdi. Accessed 20 Mar 2019
21. Verdi Runtime library repository. https://github.com/DistributedComponents/verdi-runtime. Accessed 20 Mar 2019
22. Rahli, V., Vukotic, I., Völp, M., Esteves-Verissimo, P.: Velisarios: Byzantine fault-tolerant protocols powered by Coq. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 619–650. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_22
23. Vukotic, I.: An Ecosystem for Verifying Implementations of BFT Protocols (2018)