TriCo — Triple Co-Piloting of Implementation, Specification and Tests

Wolfgang Ahrendt[1] [0000−0002−5671−2555]
Dilian Gurov[2] [0000−0002−0074−8786]
Moa Johansson[1] [0000−0002−1097−8278]
Philipp Rümmer[3] [0000−0002−2733−7098]

[1] Chalmers University of Technology,
{ahrendt, moa.johansson}@chalmers.se
[2] KTH Royal Institute of Technology, dilian@kth.se
[3] University of Regensburg and Uppsala University, philipp.ruemmer@ur.de

Abstract. This white paper presents the vision of a novel methodology for developing safety-critical software, which is inspired by late developments in learning based co-piloting of implementations. The methodology, called TriCo, integrates formal methods with learning based approaches to co-pilot the agile, simultaneous development of three artefacts: implementation, specification, and tests. Whenever the user changes any of these, a TriCo empowered IDE would suggest changes to the other two artefacts in such a way that the three are kept consistent. The user has the final word on whether the changes are accepted, rejected, or modified. In the latter case, consistency will be checked again and re-established. We discuss the emerging trends which put the community in a good position to realise this vision, describe the methodology and workflow, as well as challenges and possible solutions for the realisation of TriCo.

1 Introduction

In this white paper, we present the vision of a software methodology, called *TriCo* (Triple Co-Piloting), which targets in particular, but not exclusively, the development of safety-critical software, where defects can incur high financial or reputational costs, or can compromise human safety.

The latest safety standards in the avionics (DO-178C) and automotive (ISO 26262) areas now require or recommend the use of formal methods for ensuring the robustness of safety-critical embedded software, as conventional methods such as testing alone do not provide the required safety guarantees. However, traditional formal methods are not a good match for the current state-of-practice. The threshold to apply formal methods is high, as their application requires significant expertise, and formal methods are not well integrated into development processes. Tackling this challenge, TriCo aims to formulate a new approach to develop robust software cost-efficiently. TriCo builds on the enormous progress that has recently been made in a number of relevant fields: in automated reasoning

and constraint solving, which have in the last years produced tools that are significantly more scalable than the techniques available before; in verification and model checking, where new algorithms have been found to fully automatically analyse software programs of substantial complexity; and in machine learning, which is today able to automatically solve problems that were long thought to be beyond the reach of computers. Leveraging those advances, TriCo represents a new co-development paradigm that treats specifications as first-class objects, to be developed *simultaneously* with implementations and tests, and proposes the use of co-piloting to intelligently assist software developers in this process.

In this paper, we outline our vision of how efficient development of robust software will be conducted in the near future, with the help of novel tools, acting as the developer's co-pilot, which integrate recent advances in automated reasoning, machine learning, and synthesis.

1.1 Triple Co-piloting at a Glance

There are two main reasons why the adoption of traditional formal methods in industrial practice has been slow. Firstly, formal specifications are in general difficult to create and maintain, and require a considerable level of expertise and training. Secondly, formal techniques do not always scale well with the size of software, and their application often lacks the necessary efficiency. Both of these issues are particularly problematic when formal methods are applied a posteriori, i.e., after the software code has been produced. On the other hand, developing good specifications first, to only then start coding, has not either been a workable approach in software development practice.

The TriCo methodology aims to address these problems, using a novel method for co-development of code, tests and specifications, by means of a software development *co-pilot* integrated seamlessly into development environments. The purpose of the co-pilot is to enable the developer to create *code*, *tests* and *specifications simultaneously*, from the very beginning, and to guide the developer in this process by suggesting human-understandable modifications in the respective other artefacts whenever the developer changed one of them, such that *the three are kept consistent*. Figure 1 illustrates our approach, which is based on exact, logic-based techniques, boosted by machine learning. It can be seen as a generalisation of the well-known *correctness-by-construction* paradigm, in that all three types of artefacts are treated as citizens with equal rights when it comes to them being correct.

Recent advances in AI, and in particular in machine learning (ML), have made the latter a powerful tool to efficiently solve problems in many areas, such as computer vision, pattern recognition, and natural language processing, which have hitherto been solved only inefficiently with algorithmic approaches. The TriCo methodology aims to use ML to enhance exact techniques in a number of ways, for example, to enable the co-pilot to learn suggesting human-understandable specifications based on test-cases and implementations. ML will be used to train the co-pilot, before and during its use, learning when and what suggestions to give to the developer in different situations.

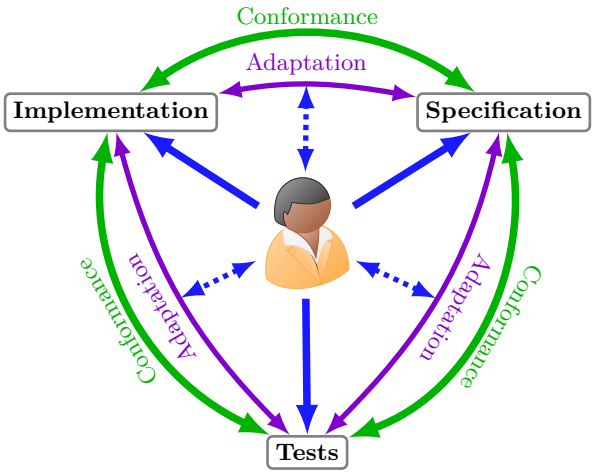

Fig. 1. Co-development and co-piloting of implementations, tests, and specifications.

The method should be supported by IDEs, providing automated co-piloting to the developer, facilitating the agile development and maintenance of the implementation-tests-specification triangle. The system would constantly analyse formal consistency of the three artefacts when one of them has changed, and provide suggestions for adaptation of the other two, to be *accepted or rejected by the developer*. The system should feed the user decisions into a common training set, thereby achieving a *federated learning* of appropriate artefact adaptions. By combining the complementary strengths of *machine learning* on the one hand and *logic based automated reasoning* on the other, the TriCo methodology forms a novel, seamlessly integrated method for developing *robust software efficiently*.

2 Emerging Trends in Software Technology

Our approach of co-development of code, tests and specifications in the spirit of co-piloting is well suited for encompassing many emerging technologies, and thus support human developers in different ways. We note that our methodology does not aim at automated *decisions* about any of the artefacts. Rather, it aims at automated *suggestions* for adapting the artefacts to each other. The suggestions are *triggered* by the user, who changed one of the artefacts, and they have to be *accepted* (and possibly modified) by the user, to bring all three artefacts in sync again in a way which matches the intentions. In the following, we review emerging trends which this endeavour can build on.

2.1 Code synthesis through large language models

With the current development of large language models with billions of parameters, trained on open source code from, e.g., GitHub, code synthesis co-pilots

seem set to soon becoming an everyday tool for programmer's integrated in common IDEs [10,6,11]. The user may type in some code or a description in natural language, and the system automatically provides a suggestion. Other tasks include translating between different programming languages, or suggesting fixes for broken code, given a compiler error message. In many cases the code is perfectly sensible, but it should be noted that no guarantees whatsoever can be given about actual correctness, and sometimes nothing useful will be generated at all. Furthermore, the large language models make no claims about understanding how the generated programs function, but they are very good at picking up common patterns seen in the vast training data, making them suitable for generation of boiler-plate code for frequently used programming languages and applications, such as web programming. The authors of Google's recent PaLM system summarise some of the limitations and risks [11], p.26:

> *"When deploying LM-based systems within software development, a key risk is that the generated code could be incorrect, or introduce subtle bugs.[...] Developers should review suggested code before adding it to a program, but they may not always find subtle bugs in suggested code.[...] Functional correctness is only one aspect of source code quality; LM-produced suggestions must also be readable, robust, fast, and secure"*

The authors further point out that there is little work on software testing and verification methods for systems including code synthesis from large language models.

Our proposed methodology can cover also software development including this kind of "untrusted" synthesis components. Suggestions from, e.g., a large language model are harnessed in a larger environment which complements it with test-cases and specifications. The latter may be written by a human, but an alternative is to allow automated generation of (suggested) test cases and specifications, which we survey next.

2.2 Automated Test Case Generation

For TriCo, a particularly relevant automated test generation technique is *property-based random test-case generation*, which manifests itself in the QuickCheck family of tools [16]. It features strong requirements coverage and very effective test-case minimisation. Here, the programmer writes down specific properties that are desired by the system, commonly as quite compact statements. One can envision a user typing in such properties for testing, which perhaps may also be used by the kind of code synthesis system we describe above as cues. The synthesised code can then be tested automatically. *Symbolic execution*-based techniques [24,3], on the other hand, feature instead high code coverage (e.g., MC/DC, full feasible branch coverage). Further, there are advances in enhancing coverage-directed test generation by machine-learning techniques [17]. Our proposed co-piloting methodology is meant to exploit automated test generation of a variety of styles, to exploit the complementary strengths of the techniques.

However, as far as the test oracle is concerned, the user shall have the final word on whether or not an oracle matches the intended behaviour.

2.3 Specification Synthesis

Formal specifications serve several purposes. They provide a compact and precise description of the functions and properties of software, and can be used as input to formal verification systems for proving correctness, or to test-case generation systems. Our co-pilot will need to perform specification synthesis to produce suggestions of specification updates when code has been changed, or to provide a compact description of code that has been synthesised. Even without passing it to a verification system, it can be helpful for the user to spot an error in generated code if complemented by a suggested specification of its functionality. Does the code do what the specification says? If not, which of them matches the intention?

Due to the effort required to write specifications by hand, specification synthesis has been recognised as a problem of increasing importance, and received attention in several communities. Specifications in the form of contracts can, for instance, be computed using the classical weakest-precondition calculus, symbolic execution [15], counterexample-guided abstraction refinement [23], or through model checking [5]. The concept of Maximal Specification Inference [4] generalises the inference of weakest pre-conditions and considers the specifications of multiple functions simultaneously. Specifications can also be derived using dynamic analysis, as for instance in the Daikon tool [12].

Theory exploration [27] is another emerging method. Given a (functional) program, it invents a concise formal description of the program behaviour as a set of equational statements. Generation is interleaved with automated testing or symbolic evaluation [26]. These properties may then be inspected by the programmer, and passed to a theorem prover to verify that the code satisfies the statements. Naturally, if given a buggy program, a specification generation tool will generate a corresponding "buggy" specification. In our co-piloting setting, the human user will still have the task of checking that the generated specification is in accordance with the users intentions, but we argue that this task can be easier than spotting a bug in the source code itself.

2.4 Formal Software Verification

Reasoning about conformance is one of the principal tasks underlying the TriCo methodology. Any change in the three considered artefacts—implementation, tests, specification—implies that the three conformance relations have to be reevaluated, so that possible mismatches are identified and can be corrected. The conformance *Implementation* $\leftrightarrow$ *Specification* is known as the formal software verification problem, and has been studied for a long time and in a variety of fields. For us, in particular two styles of verification are relevant, namely *deductive verification* (e.g., [19,2,7]) in which automated tools are applied to check fully-annotated programs; and *software model checking* [18], which attempts to

handle also programs that are only partially annotated in an automated way. A recent direction of research that combines concepts from deductive verification with model checking algorithms are approaches based on *Constraint Horn Clauses* [9,13], which form an intermediate verification language in software model checking, but more generally can provide automation for a wide range of program logics.

Machine Learning for Verification ML for automatic verification can largely be split into three different areas: (i) methods that see each verification instance as a separate learning task, e.g., the task of learning a loop invariant from observed program behaviour; (ii) methods that see each verification instance as one data point, which are then used to train a verification system; and (iii) methods for parameter learning. An overview of (i) is given in the recent tutorial [21]; the approach has seen increasing adoption over the last years, and uses algorithms like decision-tree learning, syntax-guided synthesis, or reinforcement learning [25] to solve verification problems. Approach (ii) has turned out much less successful, up to this point, given the big gap between the approximate reasoning of statistical inference and the exact nature of program verification. Approach (iii) is significantly easier to implement, and is today used, for example, to choose the right verification back-end for each verification task (e.g., [22]).

3 The TriCo Methodology

The ambition of the TriCo methodology is to take test-driven agile development [20] to a higher robustness level. In addition to the developer eagerly writing unit tests which describe the behaviour for a set of selected input cases, the methodology advocates that formal specifications covering all inputs and prestates are co-developed along with tests and implementation. The workflow is an extension of test-driven development, but gives much stronger robustness guarantees. The methodology aims in the first place at sequential correctness of programs, i.e., the absence of errors which already manifest in sequential execution.

Consider the productive use of *redundancy* in existing development processes. Typically, the latter make extensive use of *testing* to validate software behaviour: two artefacts, implementations and tests, are written that both describe intended program behaviour. Unit/regression tests are written early during the process, and are re-executed and augmented whenever new functionality is introduced, or defects are corrected. Development maintains the invariant that, at any commit point, implementations and test cases are consistent, i.e., the implementation will pass all tests. Inconsistencies between the implementation and the tests constitute a bug (in the implementation, in the tests, or in both), and today's development infrastructure will enforce the immediate correction of such bugs. It is the process of keeping implementations and tests consistent that leads to higher quality of *both* artefacts.

Test cases, however, only describe the behaviour of the implementation on selected inputs and prestates. Assurance of code behaviour beyond specific test

cases can only be provided by *formal, mathematical methods*. Formal specifications describe the intended behaviour of a program (unit) for *all* inputs and prestates in a precise, unambiguous, and machine-readable format. Specifications can, for instance, include data-consistency invariants of classes and data structures, loop invariants, procedure contracts, or assumptions and guarantees between caller and callee units. Most formal verification methods require formal specifications. Unfortunately, formal specifications are hard to produce. Furthermore, in the way formal methods are used today, their benefits during the software lifetime are long-term and hard to quantify [14]. As a consequence, most software projects do not work with formal specifications.

3.1 Envisaged Workflow

The TriCo methodology provides a software development method and the necessary tool support for efficient, agile software development, while achieving a high level of robustness. Our approach is based on three main principles, as illustrated on Figure 1: (a) *co-development* of three artefacts: implementation, test cases, and specification, (b) *co-piloting* of adaptions made necessary by changes in any of the artefacts, and (c) *combining learning- and logic*-based methods to achieve (b). To achieve scalability, the method is *compositional* w.r.t. code units, in the sense that each unit gives rise to a separate triple of artefacts, to be co-developed by the user with help of the co-pilot.

With the three artefacts come three conformance relations:

1. Conformance of an *implementation* unit and the associated unit *test cases* means foremost that the implementation makes the oracle of each test case pass successfully. But other aspects can also be included into this conformance relation, such as code coverage criteria.
2. Conformance of an *implementation* unit and its *specification*, on the other hand, means that the behaviour of *every possible run* of the unit's code satisfies the specification. This can be determined by numerous formal verification methods. In TriCo, the analysis of this conformance shall be provided in an integrated manner, in the background, without the user having to choose, install, or learn about the various methods and backend-tools which collaboratively solve this task under the hood.
3. Finally, conformance of a unit's *test cases* and the corresponding *specification* means that the specification is strictly a generalisation of all the unit's test cases (each consisting of a concrete input/prestate plus an oracle on the output/poststate). In turn, this means that each of the test cases can be understood as a very concrete 'lemma' implied by the specification. Also here, the according analysis methods execute in the background.

A basic principle of the TriCo methodology is that *mutual conformance* of implementation, tests, and specification is an *invariant that is intact in between different work passes* of code development and maintenance. When the user changes any one of the three artefacts (blue solid arrow in Fig. 1), this will

typically break the conformance invariant. For instance, a new test case may not be satisfied by the implementation, and/or require a generalisation of the specification. Or a change in the implementation may invalidate a property from the specification, and/or make some test cases fail. To address these arising inconsistencies, a TriCo empowered IDE shall analyse in the background whether conformance to any of the other two artefacts is broken. If that is indeed the case, the system computes an *adaption* of the affected artefact(s) that re-establishes mutual conformance. Computing such adaptions is based on a combination of *machine learning* with *exact, formal methods.* The adaption is presented as a suggestion to the user, who has to accept or reject it (blue dotted arrow in Fig. 1). *Only after the user has accepted a suggested adaption, it will take effect.* It is this very point which makes the method very *user-centered*. We argue that this will lead to higher-quality software than automated adaption decisions, in particular because a mismatch between two artefacts can indicate an error on either of them, and the user should decide which side meets the intention of the software unit. Finally, the user's decisions to accept or reject suggested adaptions are used to *train* the learning facilities, which will generate ever better adaption suggestions over time.

3.2 Use Cases

To be more concrete, let us consider the use case where, starting from mutually conforming implementation, test cases, and specification, the user changes the implementation. This may lead to a violation of the specification (among others), to be detected by formal methods operating in the background. Once a violation is diagnosed, the co-pilot will suggest an adaption of the specification. The adaption is produced by the trained machine learning facility, but validated with exact methods, to be finally accepted or rejected by the user, thereby training the co-pilot in turn. In case the user decides to reject an adaption of the specification, numerous proceedings can come into play, such as reverting the latest change of implementation, or hand-editing the rejected suggestion (to be re-checked), or letting the co-pilot suggest a finer modification on the latest implementation.

As a second use case, let us consider a code unit which was developed with (pure) *test driven development*, so far without any formal specifications. The user can then ask the co-pilot to incrementally suggest *specifications, obtained by generalisation from the test cases*, against which (if accepted by the user) the implementation will be verified. As a variation of this use case, a code unit may be developed from the start with the co-pilot, but by developers initially unfamiliar with formal specifications. Continuous generalisation of the incrementally developed test cases, and the corresponding verification, will make the development of formally verified software a side-product of agile test driven development, thereby offering a smooth learning curve into using the TriCo methodology.

As a third use case, let us consider the integration of legacy code into the TriCo process. When connecting actively developed code with legacy code, what is needed most for further analysis are interface specifications of the legacy units.

TriCo shall support intelligent editing of such interface specifications, empowered by machine learning. Moreover, the caller-side use of legacy functionality can be employed to infer contract specifications, and corresponding test cases to validate the contract specifications. Another approach to the same problem is to provide data generators, run the legacy code on the generated data, and generalise the observed output to synthesise specifications, following a theory exploration approach.

3.3 Envisaged Technology

Achieving the goal of integrated co-piloting of implementation, test cases, and specification requires a hybrid, highly integrated set of functionalities. For that, the TriCo methodology envisages to employ, expand on, and combine numerous techniques, such as machine learning (neural networks, reinforcement learning), formal software verification, symbolic execution, symbolic debugging, specification/contract inference, specification mining, theory exploration, test automation, automated test case generation, runtime verification, code refactoring, and combinations thereof.

We envisage a tool chain which provides to users an environment where they can co-develop, in an agile way, the three aforementioned artefacts with continuous co-piloting support. The architecture of the tool chain would be such that the core functionalities are provided in a web-service, paired with a user side web-interface. The server architecture also enables *federated training* of the learning facilities.

4 Research Efforts Required for TriCo

While much of what is presented in this paper is a vision still waiting to be turned into reality, we believe that the time is right to start a research programme that will provide the required components for TriCo. This research can build on the enormous progress that has recently been made in a number of relevant fields: in automated reasoning and constraint solving, which have in the last years produced tools that are significantly more scalable than the techniques available before; in verification and model checking, where new algorithms have been found to fully automatically analyse software programs of substantial complexity; and in machine learning, which is today able to automatically solve problems that were long thought to be beyond the reach of computers. This section discusses some of the remaining research challenges in the different fields.

Challenge 1: Representation and Transformation of Artefacts

The challenge: Formal methods have to handle the complexity of real-world programming and specification languages, e.g., of languages like C, Java, ACSL, or JML. In TriCo, this complexity is amplified by the multitude of modifications and transformations taking place between the three different artefacts (Fig. 1):

all three artefacts can be edited by the developer, at any point; all three artefacts can also be synthesised or adapted by the co-pilot to (re-)establish the conformance relations. Changes computed by the co-pilot must not destroy the cues the developer depends on (e.g., formatting, naming, or comments), and editing by the developer must not compromise meta-data maintained by the co-pilot.

A possible solution: This challenge could be addressed by defining a uniform intermediate language to represent all three artefacts, implementations, specifications, and test cases. This intermediate language is not intended for direct human editing, but is connected through meta-data to multiple real-world presentation languages (e.g., C and ACSL). The developer works with the presentation languages, and any changes made are automatically and incrementally mapped to the intermediate representation. The co-pilot primarily operates on the intermediate language, and changes on this layer are reflected by carefully updating the presentation layer.

As one suitable intermediate language, we consider the use of extended versions of *Constraint Horn Clauses (CHC)* [9,13]. CHCs have been adopted in software model checking as a common interface between programming language front-ends and verification back-ends, since CHCs are general enough to capture many programming language features (including control structure, procedure calls, various concurrency models, heap models), provide simple and unambiguous semantics in terms of the SMT-LIB theories [8], and can easily be connected to various automatic verification approaches.

Challenge 2: Efficient Automatic Conformance Analysis

The challenge: In TriCo, any change in the three artefacts (implementation, tests, specification, Fig. 1) requires a co-pilot to reevaluate the three conformance relations, so that possible mismatches are identified and can be corrected. The existing methods for conformance checking, which are formal software verification, software testing, and checking specification refinement, are all algorithmically hard, and often limited in terms of scalability.

A possible solution: Given the existing huge body of research on the different kind of conformance checking, no step-changes are to be expected in the near future, but we believe that the right application and combination of existing methods can carry TriCo a long way. A co-pilot can be based on (i) a bespoke combination of static and dynamic methods, utilising their complementary strengths, so that rapid feedback can be provided to the developer after each change; (ii) incrementality in checking, which is achieved, e.g., by the caching of conformance certificates in the form of rich program annotations (e.g., computed loop invariants); and (iii) the use of machine learning to boost checking, for instance to predict likely cases of conformance violations, or along the lines described in Section 2.4.

Challenge 3: Defect Diagnosis and Suggestion of Adaptations

The challenge: At the core of TriCo is the ability to identify and suggest adaptations of any of the artefacts—implementations, specifications, tests—when any of the other artefacts change. This is largely a new direction of research, since not all of the six possible combinations of artefacts have been considered in prior research. Closely related areas include *program repair,* which can be used to compute updates of an implementation when the specification is modified, and *model-based diagnosis* to explore the space of possible explanations for observed inconsistencies.

A possible solution: Diagnosis is triggered by conformance violations detected by the methods from Challenge 2, and aims at inferring plausible explanations for which part or feature of an artefact is responsible for the non-conformance. For this, as well as for computing adaptations, a combination of techniques from different fields is needed: *formal methods,* to exactly model the relationship between the artefacts, based on results of conformance checking; *constraint solving and optimisation,* which ensure minimality of the provided explanation or adaptation, and can be provided through modern techniques from the Satisfiability Modulo Theories (SMT) field; and *machine learning,* which is able to rank explanations and adaptations in a way that is consistent with the expectations of the user.

An important aspect is that techniques should reflect existing software development practices: a co-pilot has the purpose of supporting the developer, not to enforce a coding style the developer is unfamiliar or uncomfortable with. One line of thought towards this goal is the concept of *transformation models,* which are abstract characterisations of typical artefact modifications; covering, for instance, notions of refactoring, but also other typical steps in code editing. By learning how transformations of one artefact induce corresponding transformations of the other artefacts, a co-pilot will be able to compute and rank adaptations in a way that is consistent with developer expectations and habits. This ambitious goal of the methodology requires research in several directions: (i) an effective, domain-specific language to express transformation models has to be defined; (ii) algorithms to mine transformation models from recorded editing sequences of developers; and (iii) methods for pairing, on-the-fly matching, and instantiation of transformation models.

5 Turning Vision into Reality

What are the concrete steps to turn the TriCo vision into reality? As a first step towards implementing a full-fledged co-pilot, we envisage the development of a TriCo demonstrator in the form of an advanced integrated development environment (IDE) that includes the co-piloting functionality proposed in Section 3. The IDE could for instance be designed as a web application, to be gradually extended over time. The implementation could proceed as follows:

1. Initially, the IDE would mainly be an *editor* for implementations, specifications, and tests. This editor can be built on top of existing JavaScript frameworks, for instance the CodeMirror system [1]. It can already offer opt-in functionality to record the editing steps done by a developer, and automatically collect this data, which can later serve as training data for machine learning.
2. A second version of the IDE could include *conformance checking*, and thus initial support for co-editing all three artefacts.
3. Next, support for *diagnosis* and *adaptation* could be added to the co-pilot. Again, the IDE shall be able to record and collect developer interaction sequences, so that data becomes available to further improve the co-pilot.
4. Further extensions could include support for *collaborative development*, the integration with version control (GitHub), and support for standard continuous integration systems.

All this would lay the ground for plugging in, and closely integrate, a glowing variety of techniques, reasoning based and learning based, analytic and synthetic, to keep the three artefacts mutually in sync, in a co-piloting fashion, when the user evolves any of them.

Co-development and co-piloting have to be constantly evaluated in the light of the original objectives: to enable *efficient* production of *robust* software. Further, data has to be produced as input for training the verification and adaptation methods, using the data collection functionality, and guidelines to be developed for the integration of the TriCo approach into industrial development practices, including Agile development [20] in general and, for instance, Scrum [28]. For evaluation purposes, case studies can be formulated and carried out in different contexts: with industrial collaborators, covering development tasks that are close to the industrial practice; and within course projects at universities, providing a setup in which comparative studies between different development practices carrying out the same development task are possible. To evaluate efficiency and robustness, one could monitor the development effort in the projects and assess the quality of developed implementations and specifications through code review and independent testing, and by interviews with the developers.

We strongly believe that co-development of implementations, specifications, and tests is the future of software development. Triple co-piloting is our vision of how this future can materialise. We now invite the community to join us in discussing and further developing this vision, and to participate in its realisation.

References

1. Codemirror. `https://codemirror.net`.
2. W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, editors. *Deductive Software Verification—The KeY Book*, volume 10001 of *LNCS*. Springer, 2016.
3. W. Ahrendt, C. Gladisch, and M. Herda. Proof-based test case generation. In *Deductive Software Verification—The KeY Book*, volume 10001 of *LNCS*. Springer, 2016.

4. A. Albarghouthi, I. Dillig, and A. Gurfinkel. Maximal specification synthesis. In *Proceedings of POPL*, volume 51. ACM, 2016.
5. A. Alshnakat, D. Gurov, C. Lidström, and P. Rümmer. Constraint-based contract inference for deductive verification. In *Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY*, volume 12345 of *LNCS*. Springer, 2020.
6. J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program synthesis with large language models, 2021. arXiv:2108.07732.
7. M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6), 2011.
8. C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.
9. N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *LNCS*. Springer, 2015.
10. M. Chen et al. Evaluating large language models trained on code, arxiv:2107.03374, 2021. arXiv:2107.03374.
11. A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel. PaLM: Scaling language modeling with pathways. 2022. arXiv:2204.02311.
12. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.
13. G. Fedyukovich and P. Rümmer. Competition report: CHC-COMP-21. In H. Hojjat and B. Kafle, editors, *Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021*, volume 344 of *EPTCS*, pages 91–108, 2021.
14. M. Gleirscher, S. Foster, and J. Woodcock. New opportunities for integrated formal methods. *ACM Computing Surveys (CSUR)*, 52(6), 2019.
15. M. Gordon and H. Collavizza. Forward with Hoare. In *Reflections on the Work of C. A. R. Hoare.*, pages 101–121. Springer, 2010.
16. J. Hughes. Software testing with QuickCheck. In *Central European Functional Programming School: Third Summer School, CEFP 2009, Revised Selected Lectures.* Springer, 2010.
17. C. Ioannides and K. I. Eder. Coverage-directed test generation automated by machine learning – a review. *ACM Trans. Des. Autom. Electron. Syst.*, 17(1), jan 2012.
18. R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), oct 2009.

19. N. Kosmatov, V. Prevosto, and J. Signoles. A lesson on proof of programs with Frama-C. Invited tutorial paper. In M. Veanes and L. Viganò, editors, *Tests and Proofs*. Springer, 2013.
20. R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.
21. M. Parthasarathy and P. Garg. Machine-learning based methods for synthesizing invariants. Tutorial at CAV 2015.
22. C. Richter, E. Hüllermeier, M. Jakobs, and H. Wehrheim. Algorithm selection for software validation based on graph kernels. *Autom. Softw. Eng.*, 27(1):153–186, 2020.
23. M. N. Seghir and D. Kroening. Counterexample-guided precondition inference. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 451–471, 2013.
24. K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, page 263–272, New York, NY, USA, 2005. Association for Computing Machinery.
25. X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song. Learning loop invariants for program verification. In *Advances in Neural Information Processing Systems 31, NeurIPS 2018, Montréal, Canada*, 2018. `https://proceedings.neurips.cc/paper/2018`.
26. E. Singher and S. Itzhaky. Theory exploration powered by deductive synthesis. In *Computer Aided Verification*. Springer, 2021.
27. N. Smallbone, M. Johannson, K. Claessen, and M. Algehed. Quick specifications for the busy programmer. *Journal of Functional Programming*, 27, 2017.
28. H. Takeuchi and I. Nonaka. The new new product development game. *Harvard Business Review*, 1986.