Towards AI-assisted Correctness-by-Construction Software Development

Maximilian Kodetzki[0009−0008−9022−859X], Tabea Bordis[0009−0003−2886−0862], Michael Kirsten[0000−0001−9816−1504], and Ina Schaefer[0000−0002−7153−761X]

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{maximilian.kodetzki, tabea.bordis, kirsten, ina.schaefer}@kit.edu

Abstract. In recent years, research on artificial intelligence (AI) has made great progress. AI-tools are getting better in simulating human reasoning and behavior every day. In this paper, we discuss the extent to which AI-tools can support Correctness-by-Construction (CbC) engineering. This is an approach of formal methods for developing functionally correct programs incrementally on the basis of a formal specification. Using sound refinement rules, the correctness of the constructed program can already be guaranteed in the development process. We analyze the CbC process regarding steps for potential AI-tool support in the tool CorC, which implements CbC. We classify the findings in five areas of interest. Based on existing work, expert knowledge, and prototypical experiments, we discuss for each of the areas whether and to what extent AI-tools can support CbC software development. We address the risk of AI-tools in formal methods and present our vision of AI-integration in the tool CorC to support developers in constructing programs using CbC engineering.

Keywords: Correctness-by-Construction · Artificial intelligence · Formal methods · Deductive verification

1 Introduction

In the last years, *artificial intelligence* (AI) has made rapid progress in nearly every domain. Since *ChatGTP* was released in November 2022 [23], new AI-tools are deployed on a daily basis. Among them, generative chatbots support software engineering, i.e., by generating source code given a description in natural language or more formal specifications as *JavaDoc* [16]. Whereas ChatGPT still works as a standalone browser chatbot, various tools such as *GitHub Copilot* [1] or *Tabby* [2] come with integration into common IDEs. The basis of every AI is an AI-model that should mimic human decisions and behavior. For understanding requests written by humans, so-called *prompts*, and for generating appropriate

[1] https://github.com/features/copilot, [accessed 2024/05/13]
[2] https://tabbychat.com, [accessed 2024/05/13]

responses, nowadays, a commonly used basis for AI-tooling are *large language models* (LLMs). These models can be trained *self-supervised* using many unlabeled data sets. A current field of research focuses on explaining the reasoning of AI-models, as they are oftentimes black-boxes, even for developers. Covered by the term *eXplainable AI* (XAI), research is being conducted in a wide variety of directions in order to make the decisions of an AI-model comprehensible to humans. One approach is *Chain-of-Thought* prompting (CoT). In CoT, models are trained step-wise and with detailed information about the way a solution was deduced [7]. As a result, these models are able to explain the given solution in a more detailed way than traditional models [34].

A step-wise process as in CoT can also be found in the field of formal methods, often used for software development of safety-critical systems. The approach of *Correctness-by-Construction* (CbC) describes a process where code is developed incrementally using refinement rules. Based on a formal specification, these refinement rules are applied step-by-step [15]. The application of each of the refinement rules can be verified individually using specific side conditions. With this fine-grained implementation and verification process, errors can be detected earlier in development than it is possible with post-hoc verification, where the verification takes place after the implementation is completed. The Eclipse-plugin `CorC` offers tool support for CbC engineering and brings the incremental approach into practice [28]. In 2020, user studies have shown a decreased verification effort using `CorC` compared to post-hoc verification [29]. This is because `CorC` requires developers to think of the implementation steps before actually writing code. However, the formal process of CbC requires more effort than traditional software engineering does. Developers need a lot of knowledge and experience in the fields of formal methods. Especially, the definition of specifications, intermediate conditions, and further formal annotations is an advanced task, as their correctness and completeness is of utmost importance. In summary, CbC offers advantages in terms of verification efficiency and understanding of an implementation, but requires experience and knowledge to successfully develop functionally correct software.

In this paper, we want to discuss in which ways AI and `CorC` can work together. The goal is to evaluate to what extent AI is able to support developers in CbC engineering for addressing the problems of CbC mentioned above. Based on the common workflow, we analyze in which steps of the CbC-process AI-tools can be used to support the actions of a developer or even take over a developer's manual work completely. Analyzing the development process using `CorC`, we determine the challenges that developers have to face in the CbC engineering process and which steps of the process could be optimized. We classify the obtained challenges in five areas of interest. These are, next to the aforementioned definition of specifications, the refinement rule application process, the automated generation of source code, the verification of the side conditions of the applied refinement rules, and the usability of the `CorC` tool for developers. For each of these areas, we present general existing work on AI-support. Based on expert knowledge and prototypical experiments, we discuss whether and in

what way AI is applicable in `CorC`. Finally, we evaluate the findings regarding their benefits, risks, and challenges for the CbC-process.

In summary, we make the following contributions:

- We analyze the CbC engineering process using the tool `CorC` and determine areas of interest where AI-tools could support developers.
- We introduce existing work on AI in software engineering and formal methods in these areas of interest. Together with expert knowledge and prototypical experiments, we evaluate whether and in what way AI is applicable to the CbC-process and in `CorC`.
- We discuss the presented approaches regarding their benefit and the effort that is needed for the integration in the development process. Further, we discuss disadvantages and risks of AI applied in a formal verification process.

2 Background

In this section, we present background about CbC using the example of the *LinearSearch* algorithm. Further, we introduce the tool `CorC` that enables the development of Java-code using CbC.

Correctness-by-Construction Engineering

Correctness-by-Construction (CbC) is an incremental approach to implement functionally correct software. The implementation of a method starts with an abstract *Hoare*-triple $\{P\}\ S\ \{Q\}$ [15]. Precondition P and postcondition Q are defined in first-order logic. The abstract statement S is to be substantiated through the application of *refinement rules*. The triple can be interpreted in such a way that if precondition P is fulfilled, program S will terminate and postcondition Q will be satisfied. The final result of the CbC implementation process is a program in *Guarded Command Language* (GCL) [10]. GCL-programs can contain different programming constructs, which are represented as refinement rules in CbC. In Figure 1, we present the available rules and their logical definition. The application of these refinement rules to the abstract statement S of the starting Hoare-triple leads to an implementation that complies with the starting specification [15].

Using *deductive verification*, it is possible to verify the side conditions, i.e., individual pre- and postconditions, of each refinement step and, thus, guarantee their correctness. Deductive verification is a branch of formal methods for reasoning based on logical inference rules in order to prove a logical formula [11]. The available logical inference rules form the logical calculus which, depending on the calculus' complexity, determines whether constructing a complete proof is possible for any given valid formula, maybe even automatically, or whether the calculus is incomplete.

In Figure 2, we show the implementation of the *LinearSearch* algorithm as an example of using CbC. In the following explanation, we refer to the circled

{P} S {Q}	*can be refined to*
1. Skip:	{P} *skip* {Q} *iff* P *implies* Q
2. Assignment:	{P} x:=E {Q} *iff* P *implies* Q[x:=E]
3. Composition:	{P} S_1;S_2 {Q} *iff there is an intermediate condition* M *such that* {P} S_1 {M} *and* {M} S_2 {Q}
4. Selection:	{P} **if** $G_1 \rightarrow S_1$ **elif** … $G_n \rightarrow S_n$ **fi** {Q} *iff* (P *implies* $G_1 \vee G_2 \vee \ldots G_n$) *and* {P ∧ G_i} S_i {Q} *holds for all* i
5. Repetition:	{P} **do** [I, V] G → S **od** {Q} *iff* (P *implies* I) *and* (I ∧ ¬G *implies* Q) *and* {I ∧ G} S {I} *and* {I ∧ G ∧ V=V_0} S {I ∧ 0≤V ∧ V<V_0}
6. Method Call:	{P} b:=m(a_1, …, a_n) {Q} *with method* {P'} return r m(param p_1, …, p_n){Q'} *iff* P *implies* P'[p_i:=a_i] *and* Q'[p_i^{old}:=a_i^{old}, r:=b] *implies* Q

Fig. 1. CbC refinement rules [15].

numbers that are used as annotations in the figure. In the first step of every CbC implementation, a starting specification, supplementary global conditions, method parameters, and variables have to be defined. For the *LinearSearch* algorithm, we define an integer array `A` and an element `x` that we are looking for as parameters ①. As we already plan to use a loop to iterate over the elements of array `A`, we also define a local variable `k` as the index variable. For the starting specification ②, we define a predicate `app`, which evaluates to `true` or `false`, depending on whether value `x` is present in the provided range of array `A`. Thus, our precondition ensures that there is at least one element with value `x` in the range [0, A.len] of array `A`. As postcondition, we state that we found an element in array `A` that equals the value `x` at position `k`. For covering cases of empty or uninitialized arrays and for ensuring the boundaries of variable `k`, we define a set of global conditions ③ that has to be satisfied at all times. With this information set up, we can start the actual implementation process. We already thought of a loop to iterate over the elements of array `A`. For this, we need at least two programming constructs, one initializing the index variable `k` and another one for the loop. Thus, we first apply the composition rule ④ (see Figure 1) which comes with an intermediate condition. We define the intermediate condition to ensure index `k` to be initialized with the last position of array `A`[3] next to the predicate `app(A, x, 0, A.len)` known from the starting specification. The abstract statement S_1 that was introduced with the composition rule is now refined with an assignment `k := A.len-1` ⑤. The conditions for that assignment are propagated from the composition rule. The abstract statement S_2 is refined with the repetition rule ⑥. For this rule, we introduce an invariant `!app(A, x, k+1,`

[3] We iterate over the elements of the array from back to front to simplify the definition of the loop variant.

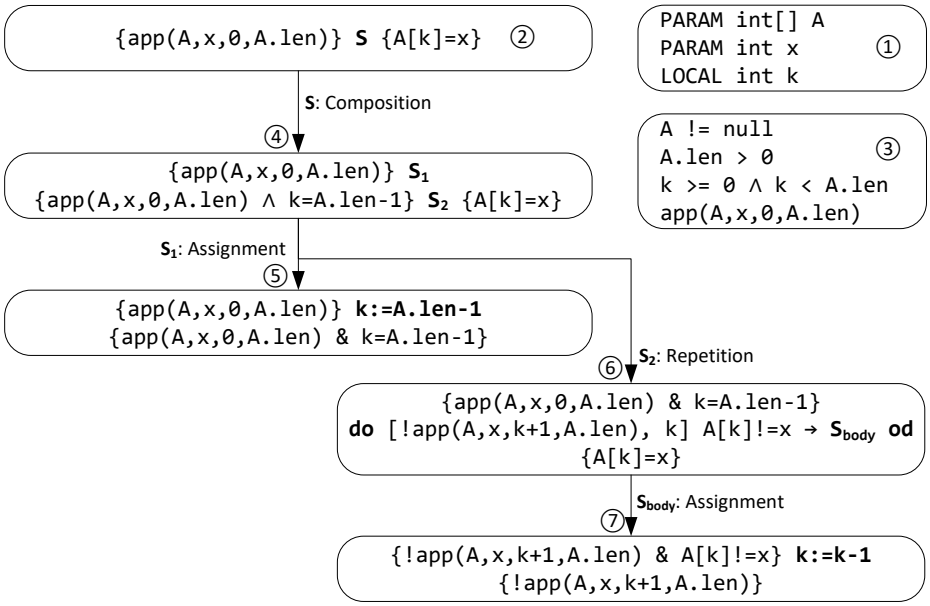

Fig. 2. Implementation of the *LinearSearch* algorithm using CbC [28].

`A.len)`, that has to hold after every loop iteration and ensures that there is no value `x` in the range of array `A` that we already checked. Next to the invariant, we define `k` as the variant for the loop. A variant is a variable or expression that decreases monotonically to show that the loop terminates. Further, we define the loop guard as `A[k] != x`, e.g., the currently considered element does not equal the value we are looking for. Finally, we refine the abstract loop body S_{body} with the assignment rule that decreases the value of the index `k` by one in each iteration ⑦. With that, the implementation is complete. To show the correctness of the implementation, we are able to verify each of the refinement steps directly after their application. Due to their individual side conditions, we do not need to wait until the implementation is completed.

Tool Support: CorC

To bring CbC into practice, Runge et al. [28] developed the open source Eclipse-plugin `CorC` [4] which makes it possible to implement programs using CbC. Using `CorC`, developers are able to implement programs by applying the refinement rules of CbC as shown in the example of the *LinearSearch* algorithm above, whereby specifications are first-order logical formulas. The plugin is based on an *Eclipse Modeling Framework*-meta-model [31] and comes with a textual and a graphical editor. In `CorC`, the side conditions of each refinement step can be

[4] https://github.com/KIT-TVA/CorC, [accessed 2024/05/03]

verified individually using the tool KeY [1] which offers possibilities of deductive verification for Java programs. In the past years, `CorC` was extended to support object-oriented projects [3] as well as software product lines [4].

3 Correctness-by-Construction Engineering in CorC

In this section, we analyze the development workflow in `CorC` in detail and identify steps where AI-tools could potentially support the process. For a structured discussion, we define the following areas of interest (AoI): *General Process* (refinement rule application process), *Specification Generation* (generation of all sorts of formal specification), *Code Generation* (generation of all Java-like code fragments), *Deductive Verification* (application of verification steps), and *Interpretation* (interpretation and explanation of a given subject, such as specifications or code). For each of the steps of the CbC-process in `CorC`, we determine the areas that could potentially support the respective step. The process is depicted in Figure 3.

Preparation. The basis of every implementation using CbC is a formal specification. This specification is usually derived from the expected behavior and not easy to define, as it must be precise (AoI: *Specification Generation, Interpretation*). Based on the specification, the developer prepares the actual implementation by defining the method signature that includes parameters and the return value. Additional local variables may also be defined. At this point, the starting specification is usually extended by global conditions (see Section 2) (AoI: *Specification Generation*).

Repetitive Process. After the preparation phase, the actual program construction in `CorC` starts. It consists of the repetitive application of refinement rules to the initial abstract statement as well as to other abstract statements that are introduced when applying selected refinement rules (AoI: *General Process*). This process is continued until all abstract statements are refined and all side conditions of the refinement rules are verified.

Refinement Rules. For each applied refinement rule, the following steps are performed: Depending on the chosen refinement rule, it may be necessary to define further annotations. These can be loop invariants or intermediate conditions for the composition rule (AoI: *Specification Generation, Interpretation*). For basic refinements, such as assignments or method calls, certain Java-code may be added to the refinement (AoI: *Code Generation*).

Verification. In `CorC`, each refinement rule is verified individually by the tool KeY (AoI: *Deductive Verification*). In the case of a successful verification, the process is continued with the next refinement rule. A failed verification indicates an error in specification or code. In `CorC`, there are several options to find the reason for a failed proof. The most intuitive option is to *look at the current refinement*. Oftentimes, errors can be found by checking the provided annotations and program statements. A common example are typing errors.

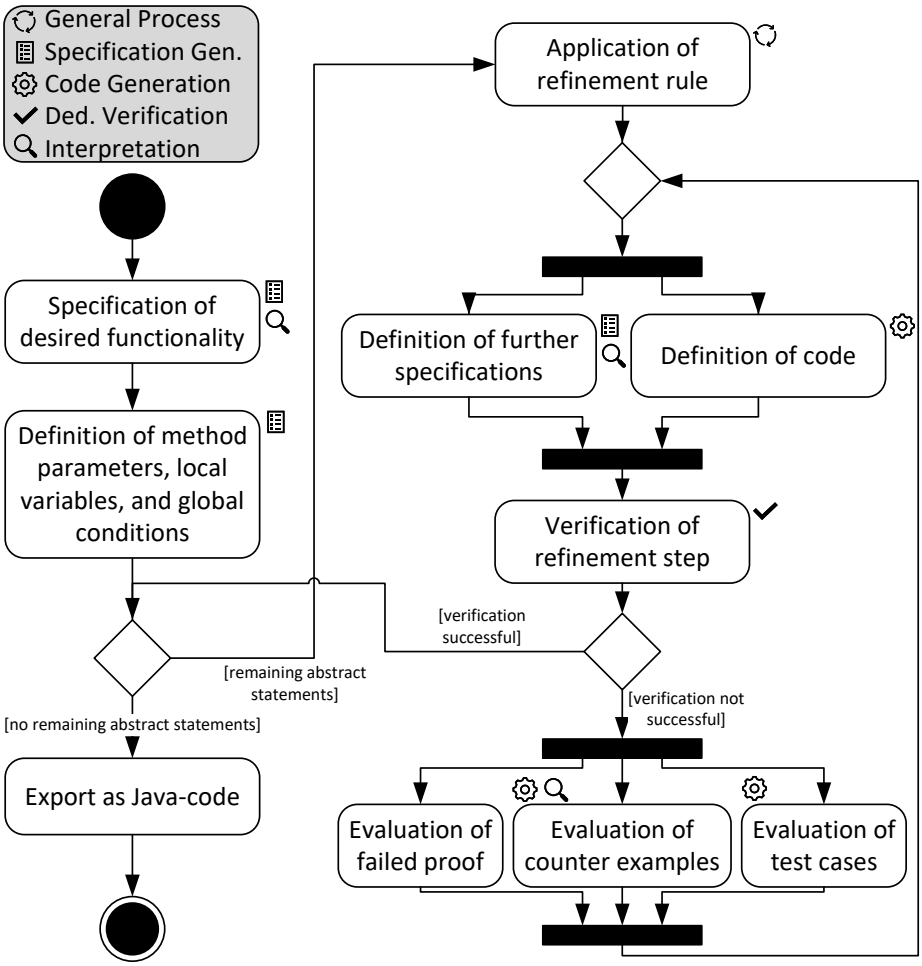

Fig. 3. CbC development process in `CorC`.

Debugging. Other errors may not be visible at first sight. For those cases, developers can track the proof that was executed by KeY. However, this technique requires knowledge of dynamic logic, symbolic execution, and the heuristics that are used in KeY. A more intuitive way to find an error are test cases and counter examples that can be generated automatically in `CorC` using the testing framework *TestNG*[5] and the theorem prover *Z3*,[6] respectively (AoI: *Code Generation*). These tools offer wide functionality in `CorC`. However, their results are not comprehensible at first glance and require time to analyze them (AoI: *Interpretation*). Based on the results of the debugging step, code or specification can

[5] https://testng.org, [accessed 2024/05/23]
[6] https://github.com/Z3Prover/z3, [accessed 2024/05/23]

be adjusted. In some cases, it may also be necessary to choose a different refinement rule, if the original strategy does not work out. For the case of adjusted annotations or code, the initial verification is rerun.

Finalization. The debugging process is repeated until the verification is successful. As soon as all abstract statements are replaced by refinement rules and all proofs could be closed, the resulting CbC program can be exported as Java-code and is functionally correct-by-construction.

4 AI-assistance for Correctness-by-Construction

In the previous section, we identified five areas of interest in the CbC-process in `CorC` where we see a potential for AI-support. In this section, we present existing work on possible applications of AI-tools and our own visions for each of these areas. For selected areas, we also conduct prototypical experiments.

4.1 General Process

The main characteristic of CbC engineering is the incremental refinement rule process. So far, refinement rules are applied by hand, based on the underlying formal specification and mainly by intuition. There is no one-fits-all solution, which is why there is no general algorithm or procedure. In the following, we discuss whether AI-tools could automatically select and apply refinement rules based on the starting specification and further side conditions.

Existing Work. At this point in time, there is no specific AI-tool to automatically apply refinement rules based on a given specification. However, modern software development offers the option of *program synthesis* [20], where programs are derived from a formal specification. This technique is usually used on the level of source code and can close the gap between human intent and executable code. Originally, program synthesis works without AI. However, recent research combines program synthesis and AI-tools to receive more precise results [9]. In contrast to traditional program synthesis, AI-assisted code generation is not limited to formal specifications. Using LLMs, it is usually enough to provide a description of the desired functionality in natural language, as this is more intuitive and does not require a specific syntax.

Discussion. So far, there is no AI-tool that can automatically construct programs using CbC engineering. However, there are multiple approaches that might be useful in future for the automatic application of refinement rules. The technique of program synthesis is combined with AI-tools in current research and could possibly be used on a more coarse-grained level than it is commonly to receive programming constructs that match the constructs that are used in CbC engineering. Further, AI-models can be trained to learn from existing data to be able to mimic human behavior. At this point in time, training an AI-model to a

satisfiable level to automate the development process using CbC does not seem to be feasible. This is due to the low number of methods implemented using CbC that could serve as training data. There are about 80 methods [28] implemented in `CorC`, which are way too few to receive a sufficiently trained model. An option to get more training data is to implement existing Java-code using CbC, to specify it, and to verify it. However, this requires a high level of human effort. Another option is *semi-supervised learning* [32] where a part of the training data is labeled, and the rest is not.

Generally, sufficiently trained models are needed to receive precise and correct results. However, even sufficiently trained models are not a guarantee for correct results which is a problem that is omnipresent in AI, as it cannot be guaranteed that the results produced by an AI-model are correct. This is of great disadvantage, especially in the area of formal verification, to which CbC belongs to, as all results generated by AI-tools have to be checked again by a human to avoid errors.

Depending on future research in the fields of AI and CbC, however, it is conceivable that an AI-tool could take over the application of refinement rules in `CorC`.

Our Vision. For the general refinement rule application process in `CorC`, we envision an AI-model that supports developers by suggesting suitable and applicable refinement rules in the CbC development phase. We imagine an integration into `CorC` that comes into play on request and is limited to the support of developers, i.e., does not automate the refinement rules application process completely. Ideally, a model would be able to analyze its suggestions to be able to check the AI-generated results for correctness before actually applying them in the CbC development process.

4.2 Specification Generation

CbC has a specification-centered process where behavioral specifications are defined before writing code and in a more fine-grained way compared to many post-hoc approaches. We therefore want to assess whether AI-tools can support the developer in defining correct and complete specifications.

Existing Work. Recently, Lathouwers and Huisman [17] published a survey of specification generators (AI- and non-AI based) for deductive verifiers for Java. They looked at tools that generate pre- and postconditions, and loop invariants in *Java Modeling Language* (JML). They categorized them into static, dynamic, and natural language processing generator tools and performed small experiments to report their experiences with the tools. In total, they discuss 14 tools, from which two are based on AI. The first tool is called *ALearner* [13], a dynamic specification generator that executes test cases and classifies the resulting program states. One of the implemented classification algorithms uses *Support Vector Machines* (SVMs) and the specification candidate is later improved with active learning. The second tool is ChatGPT [23] (GPT-4), which is based on

a large language model (LLM). They categorized ChatGPT into the natural language processing category and performed experiments where they generated pre- and postconditions from 1) code and 2) natural language specifications. Both code and natural language performed with similar results and outperformed the other specification generation tools in the survey. One drawback that the authors noticed is that ChatGPT does not fully stick to the given input, i.e., might generate additional properties that had not been provided in the prompt.

Even though Lathouwers and Huisman [17] did not explicitly include loop invariants in their experiment with ChatGPT, it is possible to generate loop invariants with ChatGPT in the same way as for pre- and postconditions. ALearner does not generate loop invariants and to the best of our knowledge, there is no other AI-based loop invariant generator for Java. For other programming languages, there are some tools that can learn loop invariants [5,30,6]. All of these tools require the program as input and involve a set of training data or human effort.

Discussion. The underlying idea of CbC is that the specification guides the development process and is therefore always defined first. This enhances thorough thinking about the algorithm itself instead of a trial-and-error programming and verification process. With that in mind, we prefer natural language processing based mechanisms for CbC in general. Another advantage of natural language as input for specification generation is that we still describe the intent of the program rather than giving the actual behavior. This is a point that has been discussed for specification generation based on code in the past, and can be mitigated by using a natural language description as input instead.

Based on the results of Lathouwers and Huisman [17], ChatGPT currently is the best candidate for generating specifications, also supporting natural language descriptions as input. However, generated specifications must be checked for inconsistencies between the generated specification and the input. In the future, we can imagine that an LLM-based tool that has specifically been trained to generate formal specifications might even improve on the results we can already achieve with ChatGPT.

Our Vision. For `CorC`, we envision a prompting interface, where the developer can interact with an LLM-based tool, such as ChatGPT, to generate specifications using natural language. We believe that natural language complements the process of CbC best. Based on the results of the experiments of Lathouwers and Huisman [17] and our own impression, we think that AI-based specification generation can be a powerful tool to support the developers in finding good specifications. Even though the generated specifications can be wrong or incomplete, revising a generated specification might be easier for the developer than defining one from scratch.

4.3 Code Generation

In CbC engineering, programs are constructed by an incremental application of refinement rules. Some of these rules, e.g., assignments, require the developer

to add explicit Java-code by hand to address the side conditions of the regarding refinement rule. Besides refinement rules, source code is also used at test case and counter example generation. In the following, we discuss the automatic generation of source code in `CorC` for these use cases using AI-tools.

Existing Work. The automated generation of source code based on a specification is one of the main research fields in AI-assisted software engineering. We briefly introduced *program synthesis* and the increasing connection with AI-tools in Section 4.1. Besides program synthesis, the intensive research of the last years in the field of AI led to tools that cannot only be used to generate source code from a description or specification, but also test cases, counter examples, documentations, and more. For that, state-of-the-art tools [24], such as *GitHub Copilot*, [7] *Tabby*, [8] or *Amazon Q Developer*,[9] are trained with incredibly large data sets, e.g., *CodeNet* [25] or *BigQuery* [22]. The result are LLMs that are used to create syntactically and functionally correct code.

The aforementioned tools offer an integration into common IDEs. This is of great advantage as prompting is easier than using standalone chatbots, such as ChatGPT. For those, a developer has to provide all necessary information in the prompting window. IDE-integrated AI-tools can retrieve this information automatically from the considered project. This offers opportunities for using existing fields and methods, automated code completion, and the adaption of naming conventions as well as the general coding style. As a result, the generated code can be used directly in the existing project and is usually more precise than code generated by chatbots [35].

In 2024, Corso et al. [8] evaluated four existing AI-tools on generating Java-code. Among them, GitHub Copilot, that proved to be the most effective assistant, whereby none of the tools was able to generate solutions for all provided problems. Finally, the authors suggest a combination of multiple AI-tools to receive satisfying results.

Discussion. Among the areas of interest discussed in this paper, AI-assisted code generation is the area for which existing AI-models could be used in the context of CbC right away. IDE-integrated tools like GitHub Copilot offer a range of functionality that could be used in `CorC` directly. The ability of code generation based on a formal specification and the project surrounding would enable automatic completion of certain refinement rules, such as assignments. The AI-assisted test case generation could support the test cases that are generated algorithmically so far. The same applies for counter examples. This could be particularly advantageous, as the current generation of counter examples in `CorC` by the theorem prover Z3 is not always successful. Despite the option of an IDE integration, automatic retrieval of project information in `CorC`, e.g., fields and methods, might not be as easy as for standard software projects.

[7] https://github.com/features/copilot, [accessed 2024/05/13]
[8] https://tabbychat.com, [accessed 2024/05/13]
[9] https://aws.amazon.com/q/developer, [accessed 2024/05/17]

As for every AI-assistance, there is a risk of incorrect results. This is relevant for the correctness of the whole program in `CorC`, as the verification of the side conditions of the applied refinement rules depends on the provided specification and the corresponding Java-code. Usually, faulty AI-generated code should be detected at verification time. However, if the specifications are also generated by AI, both errors may not be found. In the end, the developer has to ensure that specifications and code are correct to be able to guarantee the correctness of the constructed program with respect to the intended behavior.

Our Vision. We envision an AI-tool integration in `CorC` to support the CbC engineering process in the refinement process and the debugging step. Especially for the Java-code used in refinement rules, the automated generation based on the corresponding side conditions could work out well, as state-of-the-art AI-tools already provide high-quality results. To avoid erroneous source code generated by an AI-tool, it is conceivable that several suggestions for possible source code are proposed to a developer when a corresponding refinement rule is applied. For the debugging of a failed verification, an extension of the currently generated test cases and counter examples can be useful. In the future, it might also be possible for an AI-tool to interpret the results of test cases to support the developer in finding bugs.

4.4 Deductive Verification

In the CbC-process, the individual refinement steps are verified using logical reasoning, commonly by deductive verification. In `CorC`, the theorem prover KeY is used for the verification of the side conditions of each applied refinement rule. For simplicity, the verification in KeY is performed using an automatic strategy. A manual verification in KeY would impede the intended workflow significantly. As an alternative to both, automatic and manual verification, we discuss deductive verification using AI-tools in this section.

Existing Work. In a recent survey, the research of employing AI-tools to guide formal reasoning and deductive verification was categorized into five tasks: autoformalization, premise selection, proofstep generation, proof search or guidance, and others [18]. *Autoformalization* concerns the automatic conversion of informal theorems or proofs, e.g., given in natural language, into formal, i.e., machine-verifiable, formats. *Premise selection* is the task of retrieving lemmas that are helpful contributions for constructing a successful proof. The core problem of theorem proving is *proofstep generation*, which consists in predicting one or more (next) steps within building a proof. Within theorem proving, it often makes sense to do this at a higher level and predict a whole proof tactic, i.e., essentially a group of proofsteps. A closely related task is *proof search*, which is the systematic traversal of potential proof paths for constructing a valid proof tree for completing the proof. Finally, there are various other tasks that are relevant or beneficial in deductive verification such as *automated conjecturing* to generate potentially useful theorem candidates, *predicting intermediate propositions*

for filling up holes in a proof, *proof matching* to see whether already-existing proofs can be used or transformed for a problem at hand, or *theorem extraction* for decomposing existing proofs into several smaller proofs that may be (more) useful. All of these tasks are very active research areas with the main challenges of (1) suitably characterizing mathematical formulas and proofs, (2) selecting a suitable learning method, (3) choosing the right abstraction level for applying guidance by AI, and (4) constructing larger feedback loops and meta-systems that combine learning and reasoning [2]. Lastly, it should be noted that the related tasks of collecting and generating useful proof corpora for all the above tasks are very active as well, and we refer the interested reader to the concise taxonomy in Figure 3 of the survey by Li et al. [18].

Discussion. Since the verification of individual pre- and postconditions for a refinement step in CbC usually only concerns relatively small and not-so-complex formulas, the tasks of autoformalization, premise selection and proofstep or tactic generation are likely the most relevant ones among the tasks described above. AI-tools such as *Draft, Sketch, and Prove (DSP)* take an informal statement to be proven, draft it into an informal proof, then generate a formal sketch, and finally fill the gaps by producing a full formal proof [14]. Zhao et al. [36] further extend DSP by sub-goal proofs and an efficiently directed prompt selection for ChatGPT, so that besides writing the informal statement, potentially no further user interaction is needed. For premise selection, e.g., the recently developed *Magnushammer* effectively learns dependencies in order to efficiently select the premises that are needed for the proof at hand while being agnostic to the logic or type system used [21]. Finally, for proofstep or tactic generation, the recently presented LEGO-prover [33] prompts GPT-4 for generating sub-goal lemmas in a proof and finalizes the proof by proving or retrieving these lemmas, while also actively maintaining and updating these lemmas in a reusable evolving lemma library. On a higher level, deductive reasoning can also use AI for carrying out self-verification in a step-by-step manner for a (potentially larger) *Chain-of-Thought* to enhance confidence in subsequent reasoning steps [19]. All the described approaches very much depend on the quality of the underlying LLMs, but since all the generated lemmas and proofs are in the end formally verified, low quality LLMs might entail further feedback loops, but overall correctness is guaranteed. The major caveat is likely the autoformalization, since this cannot be fully verified purely by the theorem prover, but needs to be checked by the user as well. However, since the refinement steps in CbC are rather small and of low complexity, this check likely does not require much knowledge in formal logics.

Our Vision. We envision an AI-supported process that allows rapid prototyping for verifying the refinement steps, where the CbC engineer can interact in natural language without much formal knowledge. Moreover, the envisioned AI-support makes verification agnostic to the logic so that multiple theorem provers can be run at once and the best one is used. This flexibility also allows learning from the most suitable proof data collection and, at the same time, producing learning

data for each of them. A potential AI-tool in `CorC` might also be able to directly produce candidates for subsequent refinement steps within the *Chain-of-Thought* reasoning.

4.5 Interpretation

CbC engineering is an advanced approach of constructing software. Due to the formal approach, expert knowledge is required to successfully develop software in `CorC`. In the construction process, certain information is not comprehensible at first glance, which is why we discuss whether AI-tools might be able to support developers in understanding CbC engineering. We focus on the interpretation and explainability of formal specifications, counter examples, and test cases as these are of the highest importance in the CbC-process.

Existing Work. The interpretation of given objects or data is a common use case for AI-tools. Especially in the medical sector, AI is used to interpret collected data, such as MRI, CT, and X-Ray images, to identify irregularities [26,27]. In contrast, the field of AI-tools that are trained explicitly to interpret the CbC use cases mentioned above, is not as pronounced. In practice, there are tools to interpret source code using LLMs, e.g., the tool *AI Code Mentor* [10] or the ChatGPT plugin *ChatGPT Code Interpreter*. [11] However, these tools focus on the translation of source code into natural language. They are usually not able to do the same for formal specifications. For formal specifications, there is more tool support for generating specifications from natural language than the other way around (see Section 4.2). As we could see in multiple experiments that are presented below, still, LLM-based chatbots, such as ChatGPT, are able to interpret formal specifications to a limited extent. The quality of the results depends on the provided information about the program as well as the used AI-model.

Experiments. We conducted multiple prototypical experiments on the interpretation of data taken from `CorC`. First, we want formal specifications to be interpreted regarding their content. Second, we interpret the output of Z3 from the generation of counter examples to extract explicit values for the variables of the corresponding CbC program. For the experiments, we prompted basic requests to the GPT-4o model of ChatGPT. As results of AI-tools are not generally reproducible [12], we requested three responses for each prompt to show the consistency of the explanations. Due to space limitations, we only show excerpts of the first response from ChatGPT. The complete chat history can be found online. Links are provided in the respective experiments.

For the first experiment, we examined formal specifications. For that, we used the predicate `app` from the specification of the *LinearSearch* algorithm that was introduced in Section 2. We asked ChatGPT to explain the definition

[10] https://code-mentor.ai, [accessed 2024/05/27]
[11] https://openai.com/index/chatgpt-plugins, [accessed 2024/05/27]

of the predicate `app`. [12] Within seconds, we received detailed information about the provided specification. The results contain explanations of the used syntax, a description of the specification as whole in natural language, as well as examples. Most of the answers conclude with a summary in natural language. The summary of the first results is shown below. We checked all results manually and can confirm their correctness.

Experiment 1

Prompt Explain the following formal specification: 'app(A, x, begin, end) = (\exists q; (q >= begin & q < end & A[q] = x))'.

ChatGPT [...] The predicate *app(A, x, begin, end)* is true if and only if there exists an index q within the range from *begin* to *end* (where *begin* is inclusive and *end* is exclusive) such that the element at position q in the array A is equal to x. [...]

As second experiment, we focused on a complete specification consisting of the pre- and postcondition of the *LinearSearch* algorithm. The prompting style does not differ from the first experiment. However, we explicitly mention in the prompt that we provide a specification consisting of a pre- and a postcondition. This experiment is performed in the same chat as the first experiment. Thus, ChatGPT knows the definition of predicate `app`. As in the first experiment, we receive detailed explanations and examples for the given specifications in all of the three responses. The summary of ChatGPT of the first response is shown below. Again, all information was checked and is correct.

Experiment 2

Prompt Explain the formal specification consisting of precondition 'app(A, x, 0, A.len)' and postcondition 'A[k] = x'.

ChatGPT [...] This specification could describe a function or algorithm that searches for the value x in the array A and identifies an index k where x is located. Essentially, the specification ensures that if x is present in the array (as required by the precondition), then the function will correctly identify and confirm an index k where x is found (as stated by the postcondition). [...]

We mentioned above, that the second experiment was conducted in the same chat as the first one. We also performed the second experiment in a separate chat where ChatGPT does not know of meaning or definition of the predicate

[12] Experim. 1 & 2: https://chatgpt.com/share/c0ed775a-a9e9-4742-bd03-f72a376e8ddf, [accessed 2024/05/31]

`app`. [13] We used the same prompt as above, but received incorrect results. The name of the predicate `app` was interpreted as abbreviation for *append* instead of *appears* in all results. In the context of this interpretation, the explanations are correct, but, in fact, they do not match the intention of the specification for this use case.

Experiment 3 (new chat)

Prompt Explain the formal specification consisting of precondition 'app(A, x, 0, A.len)' and postcondition 'A[k] = x'.

ChatGPT [...] In summary, the specification describes a function that appends an element 'x' to the end of an array 'A' and ensures that after the operation, 'x' is located at the last index of the updated array.

In addition, we conducted the interpretation of counter examples generated by Z3 in the debugging phase of the CbC-process. The results of Z3 in `CorC` are usually very confusing and the assignments of the required variables are not immediately apparent. First experiments have shown that just prompting the output of Z3 to ChatGPT and asking for explanations does not help in extracting the necessary variables because ChatGPT explains all parts of the output, but does not extract the required values in a readable manner. Below, we show an example for a prompt to interpret the results of Z3 for a method calculating the factorial f for a given value n. [14] Using prompts like this, that explicitly ask ChatGPT to return the values of given variables, lead to more precise results. However, for this, the user needs to provide the names and types of all variables in addition to the output of Z3.

Experiment 4

Prompt Extract the values of variables int n and int f from the output of Z3 that calculated a counter example for given code and specification: [Z3 output]

ChatGPT [...]
To summarize:
- 'int n = 0'
- 'int f = 1'

Discussion. Existing work and our experiments show that AI-tools can be used to interpret development artifacts, such as formal specifications and counter ex-

[13] Experiment 3: https://chatgpt.com/share/1b07adc1-6480-486c-a069-72b9bd5bfe0b, [accessed 2024/05/31]
[14] Experiment 4: https://chatgpt.com/share/cbc6a8d3-9611-4c1f-b37e-06f0db93e574, [accessed 2024/05/31]

amples. Although we did not use an AI-tool that is specialized on formal specifications or on the output of Z3, we received detailed and oftentimes correct information. In general, the quality and correctness of the explanations highly depends on the provided information about the context of the development artifacts to be explained, e.g., available variables, methods, or predicates. We could see this especially when interpreting the specifications of the *LinearSearch* algorithm in Experiment 3. In this experiment, the specification was not interpreted correctly due to missing context of the predicate `app`. As discussed previously, incorrect results of AI-tooling are a risk for CbC engineering and its highly formal process. As it is unclear, how much context have to be given to an AI-tool to receive correct results, such tooling has to be used with care.

Our Vision. To enable a higher comprehension for less-experienced developers in `CorC`, we envision a prompting opportunity within `CorC` that uses general AI-models, such as GPT-models. So far, there is no AI-model available that focuses explicitly on the development artifacts of `CorC`. However, LLM-based chatbots, such as ChatGPT, can be a prototypical solution. We envision a prompting interface that is available on request to support developers that need help in understanding certain aspects of CbC engineering.

5 Conclusion

In this paper, we discussed how AI can support CbC engineering. For that, we analyzed the CbC-process in the tool `CorC` and determined steps in which AI-tools can assist developers. We categorized the findings in five areas of interest and presented existing work on AI-tools for each of them. Further, we discussed potential applications in CbC engineering and the tool `CorC`. The main findings are the following:

- So far, there is no targeted AI-tool support for CbC engineering.
- LLM-based chatbots, such as ChatGPT, are able to provide high-quality results for prompts regarding specification generation and code generation, as long as certain information about the context is provided. Interpretations of given programming artifacts can be explained in detail as well.
- The tasks of applying refinement rules and performing deductive verification are not feasible at this point in time as there are no AI-models that are trained up to a satisfying level to receive precise and correct results.
- The results of AI-tools are not always correct. They need to be checked by humans as the correctness of code and specifications is of high importance to be able to guarantee the functional correctness of programs constructed using CbC.
- Following that, it is not possible to replace the whole CbC-process with AI-tools. The functional correctness could not be guaranteed any more, as, for example, an incorrectly generated specification would lead to code that is correct regarding the generated specification, but not the intended functionality.

We conclude that AI-tools can support developers in CbC engineering in many steps. At this point in time, specification- and code generation as well as the interpretation of artifacts of the CbC-process are feasible and can be implemented in `CorC` in the near future. Other AI-support requires more preparation, e.g., in training AI-models. However, we are confident that, with the increasing research in the field of AI, there will be AI-tools for these tasks in the future. Finally, we underline that AI-tools can assist developers, but cannot replace them in CbC engineering, as they do not return correct results every time.

Acknowledgements. This work was partially supported by funding from the pilot program Core-Informatics of the Helmholtz Association (HGF).

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016)
2. Blaauwbroek, L., Cerna, D., Gauthier, T., Jakubův, J., Kaliszyk, C., Suda, M., Urban, J.: Learning Guided Automated Reasoning: A Brief Survey. Computing Research Repository (CoRR) (2024). https://doi.org/10.48550/arXiv.2403.04017
3. Bordis, T., Cleophas, L., Kittelmann, A., Runge, T., Schaefer, I., Watson, B.W.: Re-CorC-Ing KeY: Correct-by-Construction Software Development Based on KeY. In: The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday. pp. 80–104. Lecture Notes in Computer Science, Springer (2022). https://doi.org/10.1007/978-3-031-08166-8_5
4. Bordis, T., Runge, T., Knüppel, A., Thüm, T., Schaefer, I.: Variational Correctness-by-Construction. In: VaMoS '20: 14th International Working Conference on Variability Modelling of Software-Intensive Systems. pp. 1–9. ACM (2020). https://doi.org/10.1145/3377024.3377038
5. Bounov, D., DeRossi, A., Menarini, M., Griswold, W.G., Lerner, S.: Inferring Loop Invariants through Gamification. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems. pp. 1—13. CHI '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3173574.3173805
6. Brockschmidt, M., Chen, Y., Kohli, P., Krishna, S., Tarlow, D.: Learning Shape Analysis. In: Static Analysis. pp. 66–87. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-66706-5_4
7. Chu, Z., Chen, J., Chen, Q., Yu, W., He, T., Wang, H., Peng, W., Liu, M., Qin, B., Liu, T.: A Survey of Chain of Thought Reasoning: Advances, Frontiers and Future. Computing Research Repository (CoRR) (2023). https://doi.org/10.48550/arXiv.2309.15402
8. Corso, V., Mariani, L., Micucci, D., Riganelli, O.: Generating Java Methods: An Empirical Assessment of Four AI-Based Code Assistants (2024). https://doi.org/10.1145/3643916.3644402
9. Dehaerne, E., Dey, B., Halder, S., De Gendt, S., Meert, W.: Code Generation Using Machine Learning: A Systematic Review. IEEE Access **10**, 82434–82455 (2022). https://doi.org/10.1109/ACCESS.2022.3196347

10. Dijkstra, E.W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Communications of the ACM **18**(8), 453—457 (1975). https://doi.org/10.1145/360933.360975
11. Gödel, K.: Die Vollständigkeit Der Axiome Des Logischen Funktionenkalküls. Monatshefte für Mathematik und Physik **37**(1), 349–360 (1930). https://doi.org/10.1007/BF01696781
12. Gundersen, O.E., Kjensmo, S.: State of the Art: Reproducibility in Artificial Intelligence. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence. vol. 32, pp. 1644–1651. AAAI Press (2018). https://doi.org/10.1609/aaai.v32i1.11503
13. H. Pham, L., Tran Thi, L.L., Sun, J.: Assertion Generation Through Active Learning. In: Formal Methods and Software Engineering. Lecture Notes in Computer Science, vol. 10610, pp. 174–191. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-68690-5_11
14. Jiang, A.Q., Welleck, S., Zhou, J.P., Li, W., Liu, J., Jamnik, M., Lacroix, T., Wu, Y., Lample, G.: Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs (2023). https://doi.org/10.48550/arXiv.2210.12283
15. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer (2012). https://doi.org/10.1007/978-3-642-27919-5
16. Kramer, D.: API Documentation from Source Code Comments: A Case Study of Javadoc. In: SIGDOC99: 17th Annual International Conference on Documentation. pp. 147–153. ACM (1999). https://doi.org/10.1145/318372.318577
17. Lathouwers, S., Huisman, M.: Survey of Annotation Generators for Deductive Verifiers. Journal of Systems and Software **211**, 111972 (2024). https://doi.org/10.1016/j.jss.2024.111972
18. Li, Z., Sun, J., Murphy, L., Su, Q., Li, Z., Zhang, X., Yang, K., Si, X.: A Survey on Deep Learning for Theorem Proving (2024). https://doi.org/10.48550/arXiv.2404.09939
19. Ling, Z., Fang, Y., Li, X., Huang, Z., Lee, M., Memisevic, R., Su, H.: Deductive Verification of Chain-of-Thought Reasoning. In: Advances in Neural Information Processing Systems. vol. 36, pp. 36407—36433. Curran Associates, Inc. (2023)
20. Manna, Z., Waldinger, R.: Synthesis: Dreams → Programs. IEEE Transactions on Software Engineering **SE-5**(4), 294–328 (1979). https://doi.org/10.1109/TSE.1979.234198
21. Mikula, M., Antoniak, S., Tworkowski, S., Jiang, A.Q., Zhou, J.P., Szegedy, C., Kucinski, L., Milos, P., Wu, Y.: Magnushammer: A Transformer-Based Approach to Premise Selection. CoRR **abs/2303.04488** (2023). https://doi.org/10.48550/arXiv.2303.04488
22. Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C.: CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis (2023). https://doi.org/10.48550/arXiv.2203.13474
23. OpenAI: Introducing ChatGPT. https://openai.com/index/chatgpt/#OpenAI (2022)
24. Poser, N.: Evaluating AI-Assisted Software Engineering Tools. Bachelor's thesis, Karlsruhe Institute of Technology, Karlsruhe (2024)
25. Puri, R., Kung, D.S., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J., Chen, J., Choudhury, M., Decker, L., Thost, V., Buratti, L., Pujar, S., Ramji, S., Finkler, U., Malaika, S., Reiss, F.: CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks (2021). https://doi.org/10.48550/arXiv.2105.12655

26. Rajpurkar, P., Chen, E., Banerjee, O., Topol, E.J.: AI in Health and Medicine. Nature Medicine **28**(1), 31–38 (2022). https://doi.org/10.1038/s41591-021-01614-0
27. Rajpurkar Pranav, Lungren Matthew P.: The Current and Future State of AI Interpretation of Medical Images. New England Journal of Medicine **388**(21), 1981–1990 (2023). https://doi.org/10.1056/NEJMra2301725
28. Runge, T., Schaefer, I., Cleophas, L., Thüm, T., Kourie, D., Watson, B.W.: Tool Support for Correctness-by-Construction. In: Fundamental Approaches to Software Engineering, FASE 2019. Lecture Notes in Computer Science, vol. 11424, pp. 25–42. Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_2
29. Runge, T., Thüm, T., Cleophas, L., Schaefer, I., Watson, B.W.: Comparing Correctness-by-Construction with Post-Hoc Verification - A Qualitative User Study. In: Formal Methods. FM 2019 International Workshops. Lecture Notes in Computer Science, vol. 12233, pp. 388–405. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-54997-8_25
30. Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning Loop Invariants for Program Verification. In: Advances in Neural Information Processing Systems. vol. 31. Curran Associates, Inc. (2018)
31. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education (2008)
32. van Engelen, J.E., Hoos, H.H.: A Survey on Semi-Supervised Learning. Machine Learning **109**(2), 373–440 (2020). https://doi.org/10.1007/s10994-019-05855-6
33. Wang, H., Xin, H., Zheng, C., Li, L., Liu, Z., Cao, Q., Huang, Y., Xiong, J., Shi, H., Xie, E., Yin, J., Li, Z., Liao, H., Liang, X.: LEGO-Prover: Neural Theorem Proving with Growing Libraries. CoRR **abs/2310.00656** (2023). https://doi.org/10.48550/arXiv.2310.00656
34. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E.H., Le, Q.V., Zhou, D.: Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In: Advances in Neural Information Processing Systems (2022)
35. Zhang, B., Liang, P., Zhou, X., Ahmad, A., Waseem, M.: Practices and Challenges of Using GitHub Copilot: An Empirical Study. Computing Research Repository (CoRR) pp. 124–129 (2023). https://doi.org/10.18293/SEKE2023-077
36. Zhao, X., Li, W., Kong, L.: Decomposing the Enigma: Subgoal-Based Demonstration Learning for Formal Theorem Proving. CoRR **abs/2305.16366** (2023). https://doi.org/10.48550/arXiv.2305.16366