LLM-based Program Synthesis in Correctness-by-Construction Engineering

Maximilian Kodetzki[0009−0008−9022−859X], Hannah Grimm[0009−0002−9315−2769], Fynn Demmler[0009−0002−4399−6242], and Ina Schaefer[0000−0002−7153−761X]

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{maximilian.kodetzki, fynn.demmler, ina.schaefer}@kit.edu, hannah.grimm@student.kit.edu

Abstract. The growing reliance on software in safety-critical systems demands strong guarantees of functional correctness. Formal methods address this need by providing proofs as evidence of correctness. Among these, Correctness-by-Construction (CbC) ensures functional correctness through the incremental application of refinement rules while providing early feedback on the program correctness. However, its practical adoption is limited by the required manual effort. Among others, writing concrete program code is left to the developer. In recent years, large language models (LLMs) have demonstrated strong capabilities in code generation from both natural language and formal inputs. Therefore, we explore the use of LLMs to automatically generate code in the context of CbC to make the approach more applicable. We evaluate LLM-based program synthesis using four LLMs and a prompt tailored to CbC on a dataset of 57 specifications drawn from existing CbC case studies, each consisting of a precondition and a postcondition describing the desired behavior. We assess the generated code with respect to correctness and reproducibility across multiple runs. In addition, we compare the results to classically synthesized code, as it can currently be generated in CbC. Our evaluation shows that all considered LLMs outperform the current synthesis approach. Among them, Claude achieves the best performance, reaching 80.70% correct results.

Keywords: Correctness-by-Construction · Formal methods · Deductive verification · Program synthesis · Large language models

1 Introduction

Ensuring the correctness of a software system is a central step in any software development process. In practice, testing is a common approach for assessing the functional correctness of a program. However, it can be insufficient in domains where correctness is crucial, and failures may lead to financial loss or harm to human life, making it infeasible for safety-critical systems [42]. An alternative approach to software testing are formal methods, which rely on mathematical and logical reasoning. Formal methods can provide guarantees for functional correctness, where formal proofs serve as evidence that for every possible input, a

program behaves as specified. However, formal methods are usually more heavyweight than pure testing and require expert knowledge, limiting their use in practice [20].

An approach within formal methods that aims to lower the entry barrier for using formal techniques while ensuring functional correctness upfront is *Correctness-by-Construction* (CbC) [29,10]. CbC is a development process in which software is constructed using refinement rules and aims to guide developers through program construction while maintaining functional correctness. Thereby, each refinement rule corresponds to a programming construct, such as an assignment or a loop. The CbC development process is based on a formal specification that describes the intended functionality and is defined prior to the program construction. Side conditions associated with each refinement rule ensure that each application of a refinement rule preserves the initial specification. In contrast to post-hoc verification [2] or model checking [16,24], CbC provides early feedback on the correctness of both the overall program and its individual parts, since each refinement is verified independently.

Although CbC requires less experience with formal methods than other approaches [40], it remains a largely manual process. Tasks such as defining formal specifications, selecting appropriate refinement rules, and writing program code are still performed by the developer. To reduce the manual effort associated with these tasks, Neumann proposed an approach for program synthesis in CbC in 2024 [34]. However, the approach is limited in its scope. For the target programming language Java, it supports only basic assignments and mathematical expressions. To address the limitations of classic program synthesis, we explore the use of large language models (LLMs) for generating program code as envisioned in an earlier ISoLA track [27]. LLMs have shown strong capabilities in code generation and have been integrated into many modern development environments [44]. We build on this progress to investigate whether LLM-based synthesis can reduce the manual effort required for writing code in the CbC development process. Unlike classic synthesis approaches, LLMs are not restricted to a predefined search space. However, their non-deterministic nature can lead to hallucinations, i.e., incorrect results, whereas classic synthesis guarantees that generated solutions satisfy the original specification.

In this paper, we develop a framework for evaluating LLM-based program synthesis in the context of CbC. We define a prompt tailored to the context and requirements of CbC. We synthesize code for 57 formal specifications derived from existing CbC case studies [9,39,8] using four LLMs and evaluate the correctness of the generated code. To account for the non-determinism of LLMs, we also assess the reproducibility of results across multiple runs. Furthermore, we compare LLM-based synthesis with the existing approach proposed by Neumann [34]. Finally, we identify the most reliable LLM and integrate it into the tool WebCorC [26] complementing current CbC tool support.

The remainder of this paper is structured as follows. In Section 2, we provide background on CbC. Following this, we provide an overview of existing work on program synthesis. In Section 3, we develop a custom prompt to fit the use case

of CbC. In Section 4, we formulate three research questions, describe the set of specifications used in the study, and present the evaluation setup, including how synthesis results are generated and assessed. Afterwards, we present and discuss the results. In Section 5, we describe the integration of the most reliable LLM in the tool WebCorC. Finally, in Section 6, we summarize the main findings and give a brief overview of future work.

2 Background and Related Work

In this section, we introduce CbC, its refinement rules, and available tool support. Further, we present the *maxElement* algorithm as example. Afterwards, we give an overview of related work in program synthesis.

2.1 Correctness-by-Construction

Modern CbC as proposed by Kourie and Watson [29] builds on earlier work by Dijkstra on structured program development [19], by Morgan on *programming from specifications* [33], and by Back on refinement calculi [6]. CbC aims to guide developers through the construction of programs based on a formal specification, using a finite set of sound refinement rules to implement the desired functionality. Defining the expected behavior in advance encourages developers to reason about the structure of a program rather than implementing an arbitrary solution and correcting it afterwards.

The first step in each CbC process is the definition of the *initial* specification, which captures the desired functionality of the program. In CbC, specifications are expressed as *Hoare triples* $\{P\}\ S\ \{Q\}$, where P and Q are first-order logic formulas describing the program state before and after execution of the abstract statement S. The goal is to refine S into a concrete program in *Guarded Command Language* (GCL) [18]. This is achieved by incrementally applying refinement rules, each introducing a specific programming construct. Each refinement rule is associated with side conditions that must be satisfied to preserve the correctness of the program with respect to the initial specification.

Classical CbC includes the following refinement rules [29]: the *assignment* refinement rule introduces a variable assignment. The *skip* refinement rule implements the empty statement. The *composition* refinement rule introduces a sequence of two abstract statements. The *selection* refinement rule introduces a conditional branching with a finite set of guarded branches. The *repetition* refinement rule introduces a *while*-loop. It requires an invariant that holds before and after each loop iteration. Finally, the *method call* refinement rule introduces a method call.

Example In the following, we illustrate the CbC development process using a *maxElement* algorithm as an example. The goal of the algorithm is to determine an index `int i` in an array `int[] A` such that `A[i]` is the maximum element of the array. The corresponding CbC development is shown as a tree structure

in Figure 1. For clarity, we refer to individual steps in the figure using circled numbers. As mentioned above, each CbC process begins with the definition of a formal specification in the form of a Hoare triple, consisting of a precondition and a postcondition. In this example, we assume that the array `A` is initialized and non-empty, i.e., the length of array `A` is greater than zero. Since these properties must hold in all program states, we model them as program invariants rather than as initial preconditions. Concretely, we define the program invariants `A != null` and `A.len > 0` ①. As there is no other requirement for the program state before execution, the initial precondition is defined as `true`. The postcondition of the algorithm specifies that the index `i` refers to the maximum element of the array within its valid bounds. To express this formally, we introduce a predicate `max` which is given an array, a range within the array, and the index of the maximum element:

$$\texttt{max}(\text{int[ ] array, int start, int end, int elem}) \coloneqq$$
$$\forall\, int\, q : (start \leq q < end \rightarrow (array[q] \leq array[elem]))$$

For the initial postcondition of the algorithm, we instantiate the predicate with the array `A` and its full range from `0` to `A.len`. Together with the initial precondition, this yields the Hoare triple shown in ② of Figure 1. After defining the initial specification, we determine the structure of the algorithm to decide which refinement rules to apply. The algorithm iterates over the array `A` using a loop, comparing each element with the current maximum at position `i`. If a greater element is found, `i` is updated. Otherwise, the loop continues. For this, we introduce an iterator variable `int j`.

After sketching the planned structure, we start applying refinement rules to implement the abstract statement S of the starting Hoare triple. First, we initialize both `i` and `j`. Therefore, we apply the *composition* refinement rule ③, separating the initializations and the loop. The side condition of the *composition* refinement rule requires an intermediate condition, which we define as `i=0` $\wedge$ `j=0`. The condition holds after executing the statement S_1 and before executing S_2. The initializations are implemented by applying the *assignment* refinement rule to S_1 ④, with pre- and postconditions propagated from the parent refinement. Next, we refine the abstract statement S_2 by introducing the loop. The loop guard `j != A.len` ensures that all elements of the array are processed. We omit details on the loop variant and loop invariant for simplicity.

The loop body S_{body} consists of comparing the current element with the maximum element and incrementing the iterator variable `j`. For this, we introduce another *composition* and a *selection* statement where one branch updates `i` using an *assignment* ⑤ and the other implements the empty statement `skip` if no update is needed. According specifications are propagated from the parent refinements with respect to the definitions of the refinement rules [29]. Finally, the remaining statement S_4 is refined using an *assignment* to increment `j` ⑥. With all refinement steps applied, the construction of the algorithm is complete. If all side conditions can be proven, the resulting program is *correct by construction*.

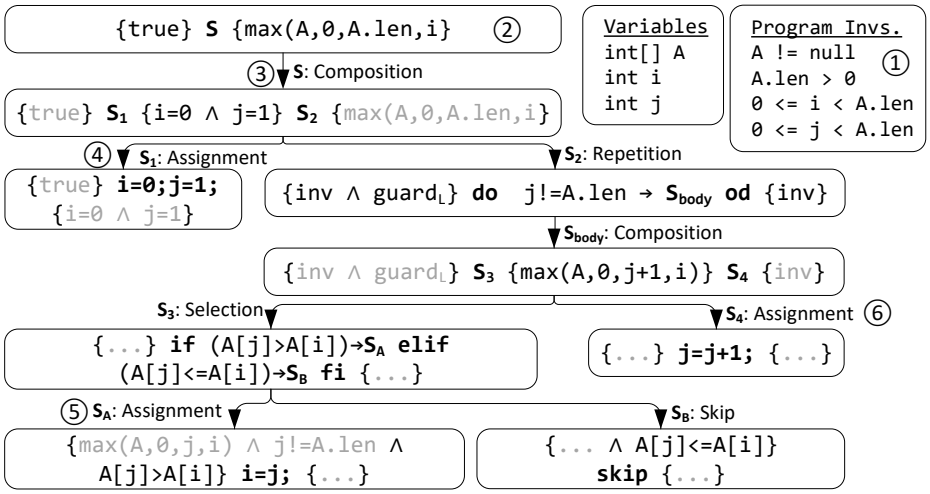

Fig. 1. CbC implementation of algorithm *maxElement*, adapted from [39]. Grey specifications are propagated from the parent refinement.

Tool Support To support the practical application of CbC, Runge et al. introduced the tool CorC [39]. CorC is an Eclipse plugin that allows constructing programs following the CbC process in both graphical and textual editors. Constructed programs can be exported as Java code. More recently, Kodetzki et al. presented the web-based tool WebCorC [1] [26], which builds on the functionality of the Eclipse plugin while omitting the need for local installation. In addition, it provides features such as collaborative work and an AI chat for the explanation of formal specifications in natural language, improving its accessibility and usability. In both tools, the verification of the side conditions of the refinement rules is performed automatically using the verifier KeY [2].

2.2 Program Synthesis

In the following, we provide an overview of the field of program synthesis. We focus on approaches that rely on formal specifications as required in the CbC development process and split existing work into two categories: classic program synthesis, i.e., deterministic techniques, and LLM-based program synthesis.

Classic Program Synthesis Program synthesis is an established research field concerned with the automatic generation of programs [23]. Within *classic* program synthesis, several paradigms can be distinguished based on how programs are derived and how the search space is defined. *Deductive synthesis* [30,31] derives programs directly from formal specifications using theorem proving. In

[1] https://corc.informatik.kit.edu

contrast, *inductive synthesis* [25,38] infers programs from input-output examples, treating synthesis as a learning problem. *Counterexample-guided inductive synthesis* [1,15,17] combines both approaches by iteratively refining candidate programs based on counterexamples obtained through formal verification. A more constrained approach is *syntax-guided synthesis* (SyGuS) [3], which derives programs with syntactic constraints, such as grammars, by restricting the search space. Finally, *constraint-based synthesis* [37] encodes synthesis problems as constraint satisfaction problems, typically solved using SAT or SMT solving.

In the context of CbC, SyGuS is currently used for program synthesis and was introduced by Neumann [34]. In this approach, synthesis is formulated as the problem of finding a program that satisfies a given semantic specification while adhering to a user-provided syntactic constraint, commonly a grammar defining the space of allowed programs. In Neumann's work, the SMT solver cvc5 [7] explores this constrained search space by generating candidate Java instructions and checking them against the specification derived from the CbC process. However, cvc5's support is currently limited to a restricted subset of Java expressions, and no other classic synthesis approach provides comprehensive support for generating arbitrary Java statements in the CbC context.

LLM-based Program Synthesis Most LLM-based program synthesis approaches use natural language descriptions as input, reflecting common usage in software development. Tools like *ChatGPT* [36] and *GitHub Copilot* [21] are integrated into development environments. However, in the CbC setting, the format of the input specification is formal, rather than in natural language. Thus, in the following overview of related LLM-based program synthesis approaches, we focus on approaches that incorporate formal specifications.

LLM4PR [14,13] is a tool that combines informal LLM-based methods to synthesize program code with formal program refinement techniques. It uses *GPT-4* [35] to iteratively refine generated code until it satisfies a formal specification. While the approach primarily targets natural language inputs, the authors report improved performance when formal specifications are included. A study from 2024 by Misu et al. [32] evaluates different prompting strategies for synthesis in *Dafny*. The results indicate that *chain-of-thought* prompting significantly improves the success rate compared to traditional prompts. The study adopts a workflow where LLM-generated candidates are checked by the Dafny verifier to ensure that the specified contract is satisfied. Bursuc et al. [12] present a study that focuses on the comparison of different LLMs used for program synthesis from formal specifications through a large-scale benchmark evaluation. The authors compare nine different LLMs across multiple benchmarks, thereby providing a broad empirical comparison of current model capabilities. The results indicate that *GPT-5* [36] performs best on two benchmark categories, while *Claude opus-4.1* [4] achieves the strongest results on the remaining one.

While existing LLM-based synthesis approaches operate at the method level, our work targets the fine grained level of individual refinements in the CbC

process. This distinction is significant, as statement-level synthesis must satisfy highly localized pre- and postconditions that emerge incrementally during the CbC process, rather than a single top-level contract. Further, no existing classic synthesis approach offers comprehensive support for generating Java statements in CbC software development, leaving LLM-based synthesis as an unexplored approach for filling this gap.

3 LLM-based Synthesis in CbC

To reduce the manual effort for developers, we propose an approach for LLM-based synthesis in CbC-based program development. With the goal of integrating the LLM which is most reliable into the tool WebCorC, we tailor the prompt to the existing structure of CbC programs.

3.1 Program Synthesis in the CbC Process

In the context of CbC, program synthesis aims to automatically generate code for refinement rules where necessary. Most refinement rules implement a structural construct of a program. For example, the *repetition* refinement rule introduces the loop construct, such that only the loop body needs to be refined further. Thereby, no loop statement has to be written manually, but is derived from the refinement rule and additionally specified information, such as the loop guard. The refinement rules that require developers to write concrete code are the *assignment* refinement rule and the *method call* refinement rule. In the following, we refer to applications of these rules as *leaf statements* in the CbC refinement tree (see Figure 1), as they cannot be refined further. The *skip* refinement rule also introduces a leaf statement, but no code needs to be implemented. Therefore, we do not consider the *skip* refinement rule in this paper.

Building on this characterization of leaf statements requiring manual implementation, we envision LLM-based program synthesis as targeting exactly these refinement steps. For each leaf statement, the corresponding specification is derived by propagating pre- and postconditions from its parent refinement according to the semantics of the applied refinement rule. This propagated specification forms the basis for prompting an LLM to generate code. However, the pre- and postconditions of a leaf statement are insufficient to ensure correct synthesis into the surrounding program context. Additional information, such as the set of available variables and program invariants, must be provided to the LLM. Further, we allow developers to specify *modifiable* variables, i.e., those that may be altered within a leaf statement, to constrain the synthesis results. We additionally incorporate context of the surrounding project in the form of existing source code files, enabling LLMs to generate calls to available methods.

3.2 Prompt Engineering

In the following, we present the prompt for synthesizing leaf statements aiming to satisfy the provided specification. The prompt is based on a *rule-based structure*,

encoding constraints for program synthesis using LLMs [5,45]. The full prompt provided to the LLMs is shown in Listing 1.

The prompt consists of four main parts. First, we establish the context of the interaction by defining the overall task, i.e., the synthesis of Java code, as it is required for the CbC tooling, from formal specifications (cf. Line 1). Next, we specify the input format, including the representation of the specification, the available (modifiable) variables, and whether the statement to be synthesized is part of a loop body (cf. Line 3). Program invariants, if applicable, are part of the precondition and postcondition rather than provided separately for design reasons. We provide the input as JSON-object for technical reasons.

The main part of the prompt defines constraints on the generated code. We defined the rules based on basic knowledge of the CbC process and its requirements to Java code in leaf statements. In addition, we iteratively refined the rules in an explorative manner, i.e., prompted different LLMs with different synthesis task on a random basis and assessed the results. Therefore, the prompt is not fully optimized, but can be improved. We discuss possible adjustments in Section 4.2.

The defined rules restrict write access to modifiable variables only (cf. Lines 6, 7), hints to access fields of objects correctly (cf. Line 8), and limit the allowed constructs to assignments and method calls, thereby excluding control structures such as loops or selections, as well as *return* statements, which would require additional refinement steps in CbC (cf. Lines 9–11). Here, we do not limit the synthesis to single assignments or method calls. Multiple Java statements may be generated for satisfying the given specification. However, assignments are restricted to previously declared variables to prevent the introduction of new variables, which would entail additional effort when integrating the generated code into the corresponding CbC program (cf. Lines 13, 14). Further constraints include the use of existing temporary variables for array element swaps (cf. Line 15), adherence to array bounds (cf. Line 16), and guidance on maintaining loop variants (cf. Line 17). We also explicitly require that the synthesized code satisfies the postcondition (cf. Line 18) and define the expected output format (cf. Lines 19–21).

Following this prompt, the JSON-object containing basic information for the synthesis and eventually available additional information, such as existing Java code, is passed to the respective LLM.

```
1 You are a small-step synthesis assistant for Java updates in CbC/KeY workflows.
      Your task is to generate the minimal but logically sufficient Java
      statement block that transforms any state satisfying the PRE-condition into
       one satisfying the POST-condition.
2
3 Input format (JSON): { "variables": [{"name": str, "modifiable": bool, "type":
      str}, ...], "pre_text": str, "post_text": str, "is_loop_update": bool }
4
5 Rules:
6 - Modify only variables flagged "modifiable".
7 - Never modify or write to non-modifiable variables.
8 - Access members of the current object syntactically correct.
9 - Prefer straight-line (loop-free) code unless "is_loop_update" is true.
```

```
- Only emit assignment statements and method-call statements on declared
    variables.
- Do not emit return statements. If available, use variable 'ret' for assigning
    return values.
- Do not emit control flow or branching constructs (`if`, `else`, `switch`, `for
    `, `while`, `do`, `try`, ternary `?:`, or extra block braces).
- Use only declared variables and their types. No new variables.
- You may read from non-modifiable arrays or objects to use their values in
    assignments.
- If a swap or movement between array partitions is required by the pre/post
    difference (e.g., element classification boundaries shift or an element
    crosses between partitions), you must perform the appropriate value swap
    using a temporary modifiable variable if available.
- Ensure array accesses remain within bounds implied by PRE.
- Always ensure that the variant variable strictly decreases when present.
- Produce all statements necessary to make POST true - not just the minimal
    syntactic change, but the minimal *semantic* change that preserves all
    invariants and updates all affected variables.
- Use Java boolean literals `true` and `false` (lowercase), never `TRUE`/`FALSE`.
- You may call helper methods from provided context files (e.g., Helper.
    someMethod(...)), but you must not modify those files; treat them as
    already correctly implemented.
- Always output EXACTLY this JSON format: {"java": "<Java statements separated by
     semicolons>"}

The user message will contain:
1. A single JSON object with the synthesis task (variables, PRE, POST,
    is_loop_update).
2. After that JSON, one or more context source files in comment-style blocks.
```

Listing 1. Prompt for LLM-based synthesis in the context of CbC.

4 Evaluation

In this section, we present a comprehensive evaluation of LLM-based program synthesis for CbC, focusing on the correctness and reproducibility of the generated implementations. In addition to LLM-based synthesis, we perform classic program synthesis using the approach of Neumann [34] (see Section 2.2) to enable a comparison between LLM-generated code and a classic technique.

To guide the evaluation, we formulate three research questions. The primary objective of the evaluation is to assess the correctness of the generated code with respect to the provided context, including the formal specification, program invariants, and the available variables and methods. Thereby, we must avoid hallucinations, i.e., plausible but incorrect outputs, as even small deviations can invalidate correctness. Thus, we aim to construct functionally correct programs, i.e., code satisfying its formal specification. Otherwise, automatic code generation provides limited value. We define the first research question as follows:

RQ1: How accurately do classic synthesis and LLM-based code generation approaches produce correct Java programs from formal specifications in the context of CbC?

While the first research question focuses on the correctness of the generated programs, it does not capture the causes of unsuccessful synthesis attempts. Analyzing failed synthesis runs is important to identify common limitations and error sources in both classic synthesis and LLM-based approaches. Therefore, we formulate the second research question as follows:

> **RQ2:** What are the reasons for failed classic synthesis and LLM-based code generation of Java programs from formal specifications in the context of CbC?

The third research question aims to assess the reproducibility of synthesized code. Due to the non-determinism of LLMs, different runs may yield different results, which can affect the reliability of the approach. We therefore evaluate the reproducibility of the generated code across multiple runs. We do not assess the reproducibility of classic synthesis, as its results are deterministic and do not vary between executions. Accordingly, we define the third research question as follows:

> **RQ3:** How reproducible are LLM-based synthesis approaches for CbC engineering in generating correct Java programs from formal specifications across multiple runs?

4.1 Methodology

In the following, we outline our evaluation methodology, including the experimental setup, the considered LLMs, and the subject systems.

LLMs The rapidly increasing number of available LLMs results in a wide range of potential candidates for evaluating program synthesis of Java code. To enable a meaningful comparison, we consider a set of four representative LLMs in this evaluation. Specifically, we include *Claude opus-4.7* by Anthropic [4], *Gemini 3.1-Pro* by Google [22], *GPT-5.5* by OpenAI [36], and *Grok-4.3* by xAI [43]. These models have demonstrated strong performance in prior work (see Section 2.2) and are increasingly integrated into modern development environments [44].

To ensure a fair comparison, each LLM is prompted under identical conditions. In particular, all models are provided with the same prompt, the same dataset, and an equal number of synthesis runs. As certain models do not allow custom sampling parameters, such as the temperature, all LLMs are prompted with their default values. This setup allows us to systematically compare the results and identify the most reliable LLM for integration into the WebCorC tool.

Case Study	Method	#Assignment Tasks	#Method Call Tasks	#Total Tasks
Algorithms	maxElement	4	-	4
	linearSearch	2	-	2
	exponentiation	3	-	3
	logarithm	2	-	2
	factorial	2	1	3
IntList	push (B)	5	-	5
	push (L)	-	1	1
	push (S)	-	2	2
	sortInc	4	-	4
	sortDec	4	-	4
BankAccount	update (BA)	3	-	3
	update (L)	6	1	7
	undoUpdate (BA)	3	-	3
	undoUpdate (L)	6	1	7
	transfer	4	3	7
	$\sum$	**48**	**9**	**57**

Table 1. Overview of case studies and the synthesis tasks.

Subject Systems For this evaluation, we use a set of 57 specifications for synthesizing Java code. These specifications are derived from existing CbC case studies [39,9], summarized in Table 1. We distinguish between the expected synthesis task types, namely assignments and method calls.

Besides basic algorithmic problems, the dataset also includes the product line case studies *IntList* and *BankAccount* [41,11]. Unlike the basic algorithmic examples, these case studies represent more realistic software systems and involve complex method interactions as well as richer specifications. Therefore, they broaden the evaluation beyond isolated algorithmic tasks. To make the *IntList* and *BankAccount* case studies suitable for our evaluation, we simplify them by removing variability inherent to product lines. After these modifications, the dataset comprises 15 methods with a total of 57 synthesis tasks.

Evaluation Setup The evaluation of LLM-based and classic program synthesis is conducted using two separate workflows.

LLM-based Synthesis As first step of the LLM-based synthesis workflow, all relevant information for a synthesis task is extracted from the corresponding method. As described in Section 3, this includes the specification of the leaf statement to be synthesized, the available variables, and the program invariants. This information is transformed into the JSON format required by the prompt and passed to the respective LLMs using their APIs. According to the prompt definition, the LLMs are expected to return Java code in JSON format.

The generated code is verified automatically using the verifier KeY [2] together with the corresponding specifications and other required Java classes. This verification process forms the basis for answering RQ1 by determining whether synthesized code satisfies its formal specification. If the verification succeeds, the synthesis is considered correct. Otherwise, it is considered incorrect. Failed verification attempts are analyzed manually to identify common causes of synthesis failures, thereby contributing to the investigation of RQ2. To account for the non-deterministic behavior of LLMs and to evaluate the reproducibility of synthesized results (RQ3), each synthesis task is executed ten times for every LLM. Thereby, each run is executed in a fresh interaction without prior context to avoid biases from previous messages and to ensure equal conditions for each task.

The workflow is implemented as a *Python* pipeline that automates both the interaction with the LLMs and the verification process [28].

Classic Synthesis We compare the results of LLM-based synthesis against a classic synthesis approach. For this purpose, we adopt the method proposed by Neumann [34], which provides a comparable evaluation pipeline. In contrast to the LLM-based approach, classic synthesis is executed only once per task, as it is deterministic and produces the same results across repeated runs. The approach either yields a valid solution that is correct by construction (RQ1), since correctness is guaranteed by the soundness of the underlying solver cvc5 [7], or it fails to produce any solution if the specification is unsatisfiable or cannot be resolved within the given constraints (RQ2).

4.2 Results and Discussion

We divide this section according to the research questions. The detailed results, including the logs of the LLM conversations, are available online [28].

RQ1: How accurately do classic synthesis and LLM-based code generation approaches produce correct Java programs from formal specifications in the context of CbC?

In Table 2, we present the results of both the LLM-based and classic program synthesis. For each LLM, we performed a total of 570 runs for 57 synthesis tasks, with ten repetitions each. We show the number of successfully generated and verified runs per method. For classic synthesis, we show the number of successful runs. Here, each synthesis task is only executed once (see Section 4.1).

Claude Among the considered LLMs, *Claude* achieved the highest number of correct synthesis results. Overall, 78.95% of the synthesized code fragments could be verified successfully. The distribution of successful runs across the case studies indicates that most algorithmic problems without method calls were synthesized correctly. More complex methods, such as *push (L)/(S)* and *transfer*, which involve richer specifications and multiple method calls, resulted in a higher number of incorrect syntheses, although correct runs were still observed for most tasks.

	Method	Claude	Gemini	GPT	Grok	Classic S.
Algorithms	maxElement	40/40	40/40	39/40	40/40	4/4
	linearSearch	10/20	20/20	20/20	20/20	2/2
	exponentiation	21/30	30/30	21/30	29/30	0/3
	logarithm	20/20	15/20	19/20	19/20	0/2
	factorial	*20/30*	*13/30*	*0/30*	*0/30*	0/3
IntList	push (B)	50/50	50/50	49/50	50/50	0/5
	push (L)	0/10	10/10	7/10	5/10	0/1
	push (S)	0/20	20/20	4/20	16/20	0/2
	sortInc	40/40	40/40	40/40	40/40	0/4
	sortDec	40/40	32/40	38/40	39/40	0/4
BankAccount	update (BA)	30/30	18/30	29/30	20/30	0/3
	update (L)	52/70	38/70	36/70	28/70	0/7
	undoUpdate (BA)	30/30	16/30	29/30	17/30	0/3
	undoUpdate (L)	50/70	38/70	32/70	36/70	0/7
	transfer	47/70	30/70	21/70	34/70	0/7
$\sum$		**450/570** **78.95%**	**410/570** **71.93%**	**384/570** **67.37%**	**393/570** **68.95%**	**6/57** **10.53%**
semantically correct		80.70%	74.91%	72.63%	74.21%	10.53%

Table 2. Successful synthesis runs of LLM-based and classic synthesis per method. The sum of semantically correct results includes unverified, but correct code of method *factorial*.

Gemini The LLM *Gemini* achieved particularly strong results for the algorithmic problems and the *IntList* case study. In contrast, correctness decreased for the more complex *BankAccount* case study, where only 51.85% of the runs were synthesized correctly. Overall, *Gemini* achieved a correctness rate of 71.93% verifiable code fragments.

GPT The LLM *GPT* achieved a correctness rate of 67.37% across all evaluated synthesis tasks. The model performed comparatively well on specifications expected to be satisfied by assignments, but showed difficulties when synthesizing method calls. This behavior was particularly visible in methods such as *push (L)/(S)*, *update (L)*, *undoUpdate (L)*, and *transfer*.

Grok The results of *Grok* are comparable to those of *Gemini*. Although the model achieved a lower overall correctness rate of 68.95%, it demonstrated strong performance on the algorithmic problems and the *IntList* case study. Similar to the other LLMs, the more complex specifications and extensive method interactions of the *BankAccount* case study resulted in lower success rates.

Classic Synthesis Classic synthesis as proposed by Neumann [34] succeeded for only 10.53% of all tasks. The successful applications were limited to the algo-

LLM	#Empty Result	Verification failed #Referencing Issue	#Limitation of KeY	#Incorrect Result	$\sum$
Claude	30 (25%)	30 (25%)	10 (8.33%)	50 (41.67%)	120
Gemini	12 (7.5%)	68 (42.5%)	17 (10.63%)	63 (39.38%)	160
GPT	124 (66.67%)	6 (3.23%)	30 (16.13%)	26 (13.98%)	186
Grok	95 (53.67%)	29 (16.38%)	30 (16.95%)	23 (12.99%)	177
$\sum$	**261 40.59%**	**133 20.68%**	**87 13.53%**	**162 25.19%**	**643 100%**

Table 3. Assessment of failed runs of LLM-based program synthesis.

rithms *maxElement* and *linearSearch*, both of which could be synthesized completely. No additional tasks from the remaining case studies were successful.

Discussion All considered LLMs outperformed the classic synthesis approach for CbC in terms of correctness of the synthesized code with respect to the provided specification. Due to the limitations of classic synthesis, only a small fraction of statements could be synthesized successfully. Among the considered LLMs, *Claude* achieved the highest correctness rate, with 78.95% of the generated programs being verified correct. Nevertheless, all LLMs were able to synthesize correct implementations for the majority of statements.

RQ2: What are the reasons for failed classic synthesis and LLM-based code generation of Java programs from formal specifications in the context of CbC?

To gain insight into the causes of unsuccessful synthesis attempts, we manually analyzed all failed runs. We present an overview of reasons for failed runs of LLM-based synthesis in Table 3 summarized per LLM. Thereby, we distinguish between empty results, where the LLMs did not generate any implementation, and synthesized statements that could not be verified by KeY.

Claude For *Claude*, 30 failed runs (25% of failures) were caused by empty responses. Since the prompt requested plain Java code without additional reasoning (see Section 3.2), the underlying causes for these empty outputs cannot be assessed further in this paper. For non-empty, but unverifiable results, we identified three main categories of failures. First, 30 runs (25% of failures) failed due to incorrect referencing of existing objects. Although object access constraints were included in the prompt (see Listing 1, Line 8), *Claude* encountered challenges with generating valid references. Second, for the *factorial* method, *Claude* frequently generated calls to the provided recursive method `Helper.factorial`. These generated method calls are semantically correct and satisfy the given specification. However, we were unable to formally verify them due to a limitation of

the verifier KeY. The verifier currently does not support automatic modular reasoning about unspecified recursive method calls in the context of CbC. This limitation affected the verification of the ten failed runs of method *factorial* (8.33% of failures). Consequently, although verification failed formally, we still consider these synthesized code fragments semantically correct. The remaining 50 failed runs (41.67% of failures) originate in various incorrectly synthesized code fragments without clearly identifiable patterns. Rating the recursive method calls for the tasks of the *factorial* method as correct leads to a total correctness of 80.70% for *Claude*.

Gemini The *Gemini* LLM stands out for the lowest number of empty results. Only twelve runs (7.50% of failures) did not contain any synthesized code. However, the LLM exhibits the highest number of referencing issues, accounting for 68 failed runs (42.50% of failures). In these cases, incorrectly generated object references prevented successful verification using KeY. *Gemini* was able to synthesize code for 13 of the tasks of the *factorial* method, which could be verified by KeY. The remaining failed runs for this method were semantically correct, but were affected by the previously discussed limitations of KeY. 63 failed runs (39.38% of failures) were caused by incorrectly synthesized code without a recurring error pattern.

GPT The majority of failed runs of *GPT* were caused by empty results. Specifically, 124 runs (66.67% of failures) did not contain any synthesized code. As with the other LLMs, the prompts requested plain Java code without additional explanation, preventing further analysis of the underlying causes for these empty outputs. *GPT* exhibits comparatively few issues related to incorrect object referencing, distinguishing it from the other considered LLMs. For the *factorial* method, *GPT* showed the same behavior observed for the other LLMs. All 30 synthesized code fragments were semantically correct, but could not be formally verified. In contrast to the high number of empty results, only 26 failed runs (13.98% of failures) were caused by incorrectly synthesized code without a clearly identifiable error pattern. Considering the semantically correct, but unverifiable recursive method calls for the *factorial* method as correct increases the overall correctness of *GPT* to 72.63%.

Grok The LLM *Grok* did not synthesize any code for 95 runs (53.67% of failures). *Grok* produced code with referencing issues for 29 runs (16.38% of failures). All synthesized code fragments for the *factorial* method were semantically correct, but could not be formally verified because of the limitation of KeY. Furthermore, *Grok* exhibited the lowest number of remaining incorrectly synthesized code fragments without identifiable error pattern among all considered LLMs (23/177, 12.99% of failures). Considering the unverified, but semantically correct code fragments of the method *factorial*, *Grok* synthesized 74.21% of tasks correctly.

Classic Synthesis As discussed in Section 2.2, the classic program synthesis approach is restricted to basic assignments and does not support array modifications or method calls. Consequently, methods from the *IntList* and *BankAccount*

case studies could not be synthesized, as they inherently require such constructs. The same limitations also prevented the synthesis of the remaining algorithmic problems.

Discussion A recurring issue across all LLMs concerns incorrect object field accesses caused by referencing errors. This observation suggests that synthesis quality may be improved by defining additional prompting rules targeting object references explicitly. Moreover, negative verification results obtained with KeY should be interpreted with care, as unsuccessful proofs do not necessarily imply semantically incorrect code. As illustrated by the *factorial* method, limitations of the verifier itself may prevent the successful proof of correctly synthesized code.

The remaining failed runs consist of either incorrect code or empty responses. We could not identify a clear pattern among the incorrect results, as both assignments and method calls occasionally failed to be synthesized correctly. However, differences between the LLMs can be observed with respect to failure behavior: *Claude* and *Gemini* more frequently hallucinate by producing incorrect implementations, whereas *GPT* and *Grok* more often fail by returning no implementation at all.

RQ3: How reproducible are LLM-based synthesis approaches for CbC engineering in generating correct Java programs from formal specifications across multiple runs?

To evaluate the reproducibility of results across multiple runs, in Figure 2, we show the distribution of tasks by the number of successfully verified runs.

Among the considered LLMs, many tasks were reproduced consistently across all runs, meaning that either all ten runs were correct (x=10) or all ten runs were incorrect (x=0). In contrast, tasks with mixed outcomes across runs occurred comparatively rarely, as shown in Figure 2. This indicates that the synthesis behavior of the considered LLMs is relatively stable for the defined prompt and the given specifications. In particular, the high number of tasks with ten correct runs suggests that many synthesis tasks are solved stably for LLMs that are capable of generating correct code for a given specification. In difference, tasks with zero correct runs indicate systematic limitations of the LLMs for certain specification patterns or shortcomings in the defined prompt rather than random generation failures. This behavior is especially visible for the *GPT* LLM.

Among the considered LLMs, *Claude* achieved the highest reproducibility, with 64.90% of all tasks synthesized correctly in every run. *Gemini*, *GPT*, and *Grok* show lower numbers of consistently correct runs. In contrast, *GPT* exhibits the largest proportion of consistently incorrect code fragments, reflecting the lower overall synthesis capability observed in RQ1.

Discussion The results on the reproducibility of the synthesis results are relevant for integrating LLM-based synthesis into CbC tooling (see Section 2.1).

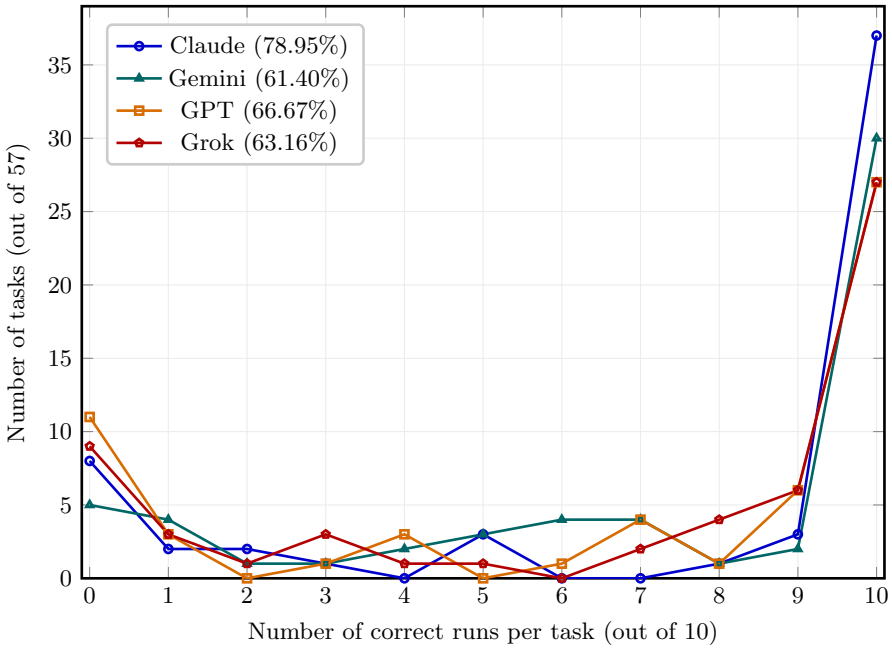

Fig. 2. Distribution of correct runs per task across the LLMs. The legend indicates the percentage of tasks for which the results were reproduced consistently across all runs, including both correct (x=10) and incorrect (x=0) outputs.

The observed reproducibility indicates that repeated prompting alone may not be sufficient to overcome synthesis limitations. Instead, improvements regarding the defined prompt, specification simplification or refactoring, or LLM reasoning capabilities may be necessary to increase correctness for complex synthesis tasks.

4.3 Threats to Validity

Although we conducted the evaluation with utmost care, certain threats to validity remain.

Internal Validity In Section 3.2, we defined a custom prompt for CbC engineering, including rules derived from development constraints and prior experience with CbC. Changes to the prompt structure, rules, or input/output format may therefore affect the results of the LLM-based synthesis. We address this threat by using the same prompt and evaluation workflow for all considered LLMs to ensure comparability of the reported results within the scope of this work.

Each synthesis task is based on a given specification. Although these originate from previous work, their correctness and completeness cannot be guaranteed.

In particular, leaf-level specifications and program invariants may be incorrect or too weak, potentially resulting in code that satisfies the specification but not the intended behavior. Nevertheless, the use of established CbC artifacts and formal verification with KeY provides an objective correctness criterion and increases confidence that verified implementations satisfy the provided formal specifications.

The proposed workflow of LLM-based synthesis relies on manual extraction of case study information into the required prompt. Additionally, the case studies *IntList* and *BankAccount* were adapted for evaluation. Despite careful handling, this process may have introduced errors that affect the synthesis results. To reduce this risk, the extracted artifacts were reviewed by multiple authors.

External Validity Due to the fine-grained nature of CbC specifications, results may not generalize to other formal methods, specification languages, or programming languages with broader contextual scopes. Consequently, the conclusions of this work are intentionally restricted to CbC and verification in Java.

In our evaluation, we use state-of-the-art LLMs in their most recent versions at the time of evaluation. Results, therefore, depend on the selected models and their optimization objectives. Different models or versions may yield different outcomes. To mitigate this, all considered models were prompted under the same conditions, enabling a comparative analysis within the scope of this paper.

As discussed in Section 4.2, the non-determinism of LLMs affects reproducibility. We addressed this issue by repeating each task ten times, but additional runs may still produce unseen outcomes. Thus, generated code should be interpreted cautiously, especially in safety-critical contexts where correctness must be ensured via formal verification, e.g., using KeY as in this work.

Finally, limitations of the verifier KeY may influence verification results. As illustrated for the *factorial* method, KeY may fail to verify correct code due to limited support for unspecified recursive method calls in the CbC setting, which limits the generalizability of negative verification outcomes. Therefore, we carefully analyzed negative verification results and did not treat them as definitive evidence of incorrect synthesis.

5 Tool Support

The goal of integrating automated program synthesis into the CbC development process is to reduce manual effort in implementing *correct by construction* programs. However, as discussed in the previous section, classic program synthesis is only applicable in a limited scope in the CbC context.

To overcome these limitations, we integrate the LLM-based synthesis approach presented in this paper into the tool WebCorC [26], which provides a web-based interface for CbC development (see Section 2.1). This integration allows developers to generate Java implementations for leaf statements directly from formal specifications within the refinement workflow, thereby increasing the

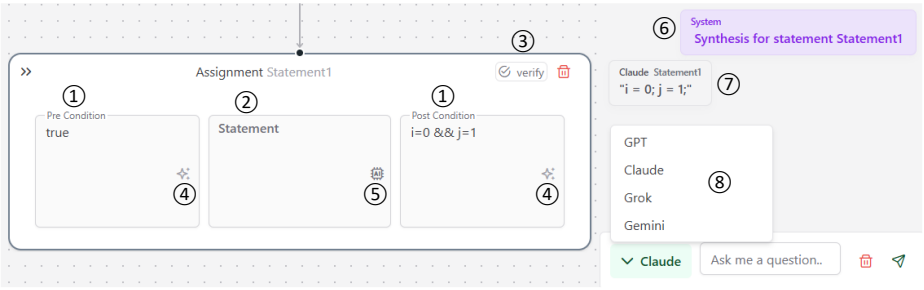

Fig. 3. Exemplary refinement and AI chat in WebCorC.

level of automation in the development process. The tool is accessible online [2] and also available for local execution [26].

LLMs in WebCorC WebCorC already provides LLM-based assistance to support developers during the CbC development process. In particular, WebCorC features an AI chat that uses *GPT-5.5* [36] to explain formal specifications in natural language and to answer questions related to the CbC methodology. This functionality is intended primarily to support unexperienced developers and to improve the usability of the approach. To preserve context across interactions, the chat maintains a conversation history. However, users may reset the history at any time to start a new session.

We extended this existing AI support by integrating the LLM-based synthesis approach presented in this paper. Figure 3 shows the refinement for an assignment and the AI chat in WebCorC. The refinement consists of the surrounding specification ①, an input field for Java code ②, the current verification status ③, controls for generating natural-language explanations of specifications ④, and a new option for automatically synthesizing Java code ⑤. When triggered, the synthesis feature prompts the *Claude opus-4.7* LLM [3] ⑥ using the prompt presented in Section 3.2, together with necessary contextual information. The prompt differs slightly from the evaluation setup, since the evaluation required synthesized code to be returned as JSON objects for technical processing purposes, but WebCorC presents generated results directly as plain Java code directly in the chat window ⑦. Synthesized code is not inserted automatically into the corresponding refinement. Instead, developers remain responsible for deciding how to incorporate synthesized code into the refinement process. Thus, formal verification using KeY only takes place after inserting synthesized code into the corresponding refinement.

[2] https://corc.informatik.kit.edu

[3] We selected *Claude opus-4.7* as the default model as it achieved the best results in our evaluation with respect to both correctness and reproducibility. Nevertheless, users may choose alternative LLMs within the AI chat interface ⑧, enabling experimentation with different models for specific synthesis tasks.

6 Conclusion

In this paper, we evaluated LLM-based synthesis for the CbC development process. We analyzed the CbC development process with respect to its potential for automated program synthesis using LLMs. Based on the identified synthesis use cases, we designed a prompt for generating Java code from formal specifications in the context of CbC. We evaluated the correctness and reproducibility of synthesized code for a set of 57 specifications for four LLMs. The evaluation showed that *Claude* achieved the highest correctness and reproducibility among the considered models. Failed verification origins in empty results from the LLMs, referencing problems, and limitations of the verifier KeY. We integrated LLM-based synthesis using *Claude* into the WebCorC tool to provide automated synthesis support within the CbC development process.

Future work will address the failed synthesis attempts and empty results observed in our evaluation by refining the prompting strategy and improving the subsequent verification process in the evaluation setup. In particular, verifier-guided feedback loops could improve synthesis quality by providing information about failed proof attempts for generated code and corresponding feedback to the LLMs. Beyond statement-level synthesis, a promising research direction is the automated generation of complete CbC programs from initial specifications. This may include the automatic selection and application of refinement rules to derive correct by construction programs with minimal user interaction.

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

References

1. Abate, A., David, C., Kesseli, P., Kroening, D., Polgreen, E.: Counterexample Guided Inductive Synthesis Modulo Theories. In: Computer Aided Verification. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_15
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification – The KeY Book, Lecture Notes in Computer Science, vol. 10001. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6
3. Alur, R., Bodik, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: 2013 Formal Methods in Computer-Aided Design (2013). https://doi.org/10.1109/FMCAD.2013.6679385
4. Anthropic: Claude (2026), https://www.anthropic.com
5. Azaria, A., Mitchell, T.: The Internal State of an LLM Knows When It's Lying. In: Findings of the Association for Computational Linguistics: EMNLP 2023. Association for Computational Linguistics, Singapore (2023). https://doi.org/10.18653/v1/2023.findings-emnlp.68
6. Back, R.J., Wright, J.: Refinement Calculus. Springer, New York, NY (1998). https://doi.org/10.1007/978-1-4612-1674-2

7. Barbosa, H., et al.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
8. 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. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-08166-8_5
9. Bordis, T., Kodetzki, M., Runge, T., Schaefer, I.: VarCorC: Developing Object-Oriented Software Product Lines Using Correctness-by-Construction. In: Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops. Springer International Publishing, Cham (2023). https://doi.org/10.1007/978-3-031-26236-4_13
10. Bordis, T., Kodetzki, M., Schaefer, I.: From Concept to Reality: Leveraging Correctness-by-Construction for Better Algorithm Design. Computer **57**(7) (2024). https://doi.org/10.1109/MC.2024.3390948
11. Bordis, T., Runge, T., Schultz, D., Schaefer, I.: Family-Based and Product-Based Development of Correct-by-Construction Software Product Lines. Journal of Computer Languages **70** (2022). https://doi.org/10.1016/j.cola.2022.101119
12. Bursuc, S., et al.: A benchmark for vericoding: formally verified program synthesis (2025). https://doi.org/10.48550/arXiv.2509.22908
13. Cai, Y., Hou, Z., Sanan, D., Luan, X., Lin, Y., Sun, J., Dong, J.S.: Automated Program Refinement: Guide and Verify Code Large Language Model with Refinement Calculus. Proceedings of the ACM on Programming Languages **9** (2025). https://doi.org/10.1145/3704905
14. Cai, Y., et al.: Towards Large Language Model Aided Program Refinement (2024). https://doi.org/10.48550/arXiv.2406.18616
15. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Computer Aided Verification. Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/10722167_15
16. Clarke, E.M.: Model Checking. In: Foundations of Software Technology and Theoretical Computer Science, vol. 1346. Springer Berlin Heidelberg, Berlin, Heidelberg (1997). https://doi.org/10.1007/BFb0058022
17. Clarke, E., Gupta, A., Strichman, O.: SAT-based counterexample-guided abstraction refinement. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **23**(7) (2004). https://doi.org/10.1109/TCAD.2004.829807
18. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18**(8) (1975). https://doi.org/10.1145/360933.360975
19. Dijkstra, E.W.: A discipline of programming. Englewood Cliffs, N.J. : Prentice-Hall (1976)
20. Garavel, H., Ter Beek, M.H., Van De Pol, J.: The 2020 Expert Survey on Formal Methods. In: Formal Methods for Industrial Critical Systems, vol. 12327. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-58298-2_1
21. GitHub, I.: GitHub Copilot (2026), https://github.com/features/copilot
22. Google: Gemini (2026), https://gemini.google.com/
23. Gulwani, S., Polozov, O., Singh, R.: Program Synthesis. Foundations and Trends in Programming Languages **4**(1-2) (2017). https://doi.org/10.1561/2500000010
24. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). https://doi.org/10.1145/1592434.1592438

25. Kitzelmann, E.: Inductive Programming: A Survey of Program Synthesis Techniques. In: Approaches and Applications of Inductive Programming. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11931-6_3
26. Kodetzki, M., Bastron, M., Flaschenträger, M., Grimm, H., Payne, C., Sharfeddin, J.: KIT-TVA: WebCorC 1.2. Zenodo (2026). https://doi.org/10.5281/ZENODO.20181249
27. Kodetzki, M., Bordis, T., Kirsten, M., Schaefer, I.: Towards AI-Assisted Correctness-by-Construction Software Development. In: Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering Methodologies. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-75387-9_14
28. Kodetzki, M., Grimm, H., Demmler, F., Schaefer, I.: Evaluation Data and Pipeline. Zenodo (2026). https://doi.org/10.5281/ZENODO.20181120
29. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27919-5
30. Manna, Z., Waldinger, R.: A Deductive Approach to Program Synthesis. ACM Transactions on Programming Languages and Systems **2**(1) (1980). https://doi.org/10.1145/357084.357090
31. Manna, Z., Waldinger, R.: Fundamentals of Deductive Program Synthesis (1992)
32. Misu, M.R.H., Lopes, C.V., Ma, I., Noble, J.: Towards AI-Assisted Synthesis of Verified Dafny Methods. Proceedings of the ACM on Software Engineering **1**(FSE) (2024). https://doi.org/10.1145/3643763
33. Morgan, C.: Programming from specifications. Prentice-Hall, Inc., USA (Apr 1990)
34. Neumann, D.: Program Synthesis from Formal Specifications for Correctness-by-Construction in CorC (2024)
35. OpenAI: GPT-4 Technical Report (2024). https://doi.org/10.48550/arXiv.2303.08774
36. OpenAI: GPT-5 (2026), https://openai.com
37. Osera, P.M.: Constraint-based type-directed program synthesis. In: Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development. ACM, Berlin Germany (2019). https://doi.org/10.1145/3331554.3342608
38. Polozov, O., Gulwani, S.: FlashMeta: a framework for inductive program synthesis. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM, Pittsburgh PA USA (2015). https://doi.org/10.1145/2814270.2814310
39. 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, vol. 11424. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-16722-6_2
40. 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, vol. 12233. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-54997-8_25
41. Scholz, W., Thüm, T., Apel, S., Lengauer, C.: Automatic detection of feature interactions using the Java modeling language: an experience report. In: Proceedings of the 15th International Software Product Line Conference, Volume 2. ACM, Munich Germany (2011). https://doi.org/10.1145/2019136.2019144

42. Valmari, A.: The state explosion problem. In: Lectures on Petri Nets I: Basic Models: Advances in Petri Nets. Springer, Berlin, Heidelberg (1998). https://doi.org/10.1007/3-540-65306-6_21
43. xAI: Grok (2026), https://www.grok.com
44. Zhang, B., Liang, P., Zhou, X., Ahmad, A., Waseem, M.: Practices and Challenges of Using GitHub Copilot: An Empirical Study. Computing Research Repository (CoRR) (2023). https://doi.org/10.18293/SEKE2023-077
45. Zhou, Y., Muresanu, A.I., Han, Z., Paster, K., Pitis, S., Chan, H., Ba, J.: Large Language Models Are Human-Level Prompt Engineers (2023). https://doi.org/10.48550/arXiv.2211.01910