X-by-Construction: Towards Ensuring Non-functional Properties in by-Construction Engineering

Maximilian Kodetzki
Karlsruhe Institute of Technology
Karlsruhe, Germany
maximilian.kodetzki@kit.edu

Tabea Bordis
Karlsruhe Institute of Technology
Karlsruhe, Germany
tabea.bordis@kit.edu

Alex Potanin
Australian National University
Canberra, Australia
alex.potanin@anu.edu.au

Ina Schaefer
Karlsruhe Institute of Technology
Karlsruhe, Germany
ina.schaefer@kit.edu

Abstract

Correctness-by-Construction engineering (CbC) is a refinement-based approach to develop functionally correct programs based on a formal specification. By correctly applying refinement rules during development, CbC enables detection of bugs during program construction, unlike post-hoc verification, which proves correctness only after implementation. Support for CbC engineering for non-functional properties is summarized under the term X-by-Construction (XbC). However, current XbC approaches are limited to information flow properties, leaving other non-functional properties of software quality, such as performance or reliability, unsupported. To address this gap, we present our vision for generalizing XbC to integrate non-functional properties into by-Construction engineering. In this way, we leverage the development of high-quality software through a refinement-based approach for future software engineering. With that, it will become possible to develop software ensuring that it not only exhibits functional correctness, but also non-functional guarantees by construction. Further, we propose ideas for ensuring energy efficiency in by-Construction engineering. We assess what it needs to integrate non-functional properties into by-Construction engineering and discuss arising challenges.

CCS Concepts: • **Software and its engineering → Formal software verification**; *Software design tradeoffs*; **Software performance**; *Software verification.*

Keywords: Software Quality, Correctness-by-Construction, Formal Methods, Resource Consumption, Energy Efficiency

This work is licensed under a Creative Commons Attribution 4.0 International License.

Onward! '25, Singapore, Singapore
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2151-9/25/10
https://doi.org/10.1145/3759429.3762623

ACM Reference Format:
Maximilian Kodetzki, Tabea Bordis, Alex Potanin, and Ina Schaefer. 2025. X-by-Construction: Towards Ensuring Non-functional Properties in by-Construction Engineering. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3759429.3762623

1 Introduction

In our digital age, the demand for high-quality software is increasing each day. Functionally correct software is more important than ever, especially in domains like healthcare, aviation, or finance, where software failures can lead to serious consequences, such as data loss, financial damage, or even endangering human lives. However, software quality is not only about functionality, but also other criteria such as performance, reliability, security, and maintainability, as defined in ISO 25010[1]. The set of properties beyond functional correctness is often referred to as *non-functional properties.*

Common to all properties, functional and non-functional, is that they are usually analyzed after the implementation of the software, i.e., *post-hoc* [2, 3]. For ensuring functional correctness, the verification of software against a formal specification that describes the desired functionality is an established method [2]. However, only proving the correctness of a program after its implementation increases the difficulty of finding bugs if the verification fails, as the problem is usually not visible right away. A similar challenge applies to non-functional properties, where a developer needs to find out, after analyzing a program regarding a certain property, how to increase the level of fulfillment.

An approach that differs from traditional *post-hoc verification* to receive functionally correct code is *Correctness-by-Construction engineering* (CbC) [48]. CbC is advantageous in comparison to post-hoc verification as the correctness

[1] https://www.iso.org/standard/78176.html, [accessed 2025/04/01]

of a program is already established in the implementation phase and not just afterwards. CbC describes an incremental process, where a set of sound refinement rules is applied to an abstract program [10]. The selection of the corresponding rules is based on a given formal specification.

So far, CbC only considers the functional correctness of software. For non-functional properties, ter Beek et al. [80] coined the term *X-by-Construction* (XbC) as an extension of CbC to ensure non-functional properties, whereby the *X* can represent any program property. However, out of the non-functional properties of ISO 25010, which we focus on in this paper, only security has been considered so far. The results are *Confidentiality-by-Construction* (C14bC) [76] and *Information Flow Control-by-Construction* (IFbC) [70, 71] which ensure that programs do not leak secret information by considering a data-flow policy. Despite the existing definition of the term X-by-Construction, no further approaches have been developed to ensure other criteria of software quality using by-Construction engineering. Thus, the paradigm is limited to functional correctness and information flow properties.

In this paper, we present our vision of generalizing by-Construction engineering to support further non-functional properties to enhance the development of high-quality software using a refinement-based process. We introduce a road map for extending XbC to a software development paradigm that encompasses all essential capabilities required for future software engineering. In addition to a general overview of our vision, we present initial ideas regarding energy efficiency as the first non-functional property beyond information flow by introducing symbolic energy annotations. We present existing work on assuring performance properties and propose how energy efficiency can be ensured side-by-side with functional specifications. Based on the experiences that we made with these ideas and the existing work on XbC, C14bC and IFbC, we discuss the suitability of different kinds of non-functional properties to a constructive development process and define requirements for future extensions of XbC. We highlight the main challenges, such as trade-off decisions, hardware-dependencies and modularity issues. In addition to the conceptual aspects of extending XbC, we present our vision of tool support to enable the development of programs for a parametric set of non-functional properties. Further, we discuss the applicability to real-world systems and introduce a customizable web-framework called XCorC, that is based on the tool CorC [72], which currently supports the development of programs using CbC and IFbC, and its web-framework WebCorC[2].

This paper is divided into five main sections. First, we provide background on CbC and introduce the running example used in this paper. In Section 3, we present the state-of-the-art of XbC and provide details on different interpretations of the term. Following that, we sketch our first ideas regarding an extension of XbC for energy efficiency using abstract, symbolic energy annotations in Section 4 and discuss related work on performance efficiency. In Section 5, we focus on the generalizability of XbC to support further non-functional properties. We define six requirements that have to be met to integrate a non-functional property into by-Construction engineering and evaluate the suitability of the non-functional properties of ISO 25010 to XbC. Finally, in Section 6, we present our vision of extending XbC to receive a comprehensive approach to programming while increasing the quality of software already in the construction phase. We discuss arising issues and challenges, as well as ongoing and future work.

2 Correctness-by-Construction Engineering

In this section, we provide background on Correctness-by-Construction. For that, we introduce the implementation of the *linear search* algorithm using CbC that serves as a running example in this paper. We show, how CbC can be used in practice, present approaches on applying CbC to large-scale systems, and introduce existing tool support for CbC.

2.1 Correctness-by-Construction

Correctness-by-Construction (CbC) is a refinement-based, incremental process for constructing functionally correct software and roots in work of Dijkstra, Gries, Back, and Morgan on weakest precondition reasoning [4, 20, 29, 58]. The CbC development process as proposed by Kourie and Watson in 2012 [48] starts with a formal contract, which can be summarized in a *Hoare*-triple *{P} S {Q}* where *P* and *Q* are pre- and postconditions of the abstract program *S*. The conditions *P* and *Q* are generally specified in first-order logic. The triple states that if precondition *P* is fulfilled before program *S* is executed, the program will terminate and postcondition *Q* will be fulfilled after the execution.

To construct a concrete program, different refinement rules can be applied incrementally to the abstract program *S*. The main rules of CbC are shown in Figure 1. They represent main procedural programming constructs such as assignments, if-queries, and loops. For some refinement rules, further annotations have to be defined, e.g., loop invariants for the *repetition refinement rule*. For each applied rule, the side conditions can be verified individually. This process results in a program in *Guarded Command Language* (GCL) [19] that fulfills the original formal specification and is, thus, correct by construction.

2.2 Running Example: Linear Search

In the following, we present the process of implementing our running example, the *linear search* algorithm, using CbC. The

[2]https://corc.informatik.kit.edu, [accessed 2025/04/03]

{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

Figure 1. CbC refinement rules [48].

complete CbC-implementation is shown in a tree-structure in Figure 2. For the algorithm, we are given an integer array A and a value x that shall be found in the bounds of array A. The position of this element is to be returned by the method. The specification of the starting triple *{P} S {Q}* is shown in the top box of Figure 2. It requires the value x to be part of array A at least once using the predicate app[3]. We define this requirement to avoid handling case distinctions for arrays not containing value x. The postcondition of the contract states that the return value of the algorithm, expressed by the keyword \result, shall point to some occurrence of value x in array A. Next to the general contract of the method, we require certain conditions to be fulfilled at any time of the execution of the program. Those are declared as *global conditions* on the right side of the figure and ensure that array A is initialized and not empty. Further, we require the iterator variable k to stay in the bounds of array A and that there is at least one occurrence of value x in array A at all times. Finally, the predicate sorted[4] requires array A to be sorted increasingly. This condition is required for an extension of this example later in the paper. We already introduce the predicate at this point to maintain consistency. For now, it does not influence the construction process of the algorithm, as *linear search* is operating independently of the order of the elements in the considered array.

Based on the defined contract, we start constructing the algorithm. As we will iterate over array A, we define a local variable k which will serve as iterator variable. The first refinement rule to apply to the abstract statement S is the *composition rule*, as we need to initialize variable k with one end of the array and implement a loop for the iteration. The *composition rule* introduces two new abstract statements S_1 and S_2. The pre- and postconditions are propagated from the starting specification w.r.t. the refinement rules shown in Figure 1 and do not need to be defined again. However, when applying the *composition rule*, we define an intermediate condition which needs to hold between the execution of S_1 and S_2. It consists of the precondition of the algorithm's contract and the expectation on what S_1 should do, i.e., to initialize variable k to the last element of array A.[5] Following the definition of the intermediate condition, the abstract statement S_1 is refined with the *assignment rule* for the initialization of variable k to point on the last element of array A. With that, the left branch of the composition is completed. We can now already prove the correctness of the application of the *assignment rule* using its side conditions.

To complete the algorithm, we are applying the *composition rule* a second time to receive the abstract statements S_3 and S_4 for the loop and returning the found position. Again, pre- and postcondition are propagated from the parent refinement step, i.e., statement S_1. The introduced intermediate condition states that variable k points to a position in array A that equals value x, as this is what we want the result of the loop to be. By applying the *repetition rule* to S_3, we need to define a loop guard, a loop invariant, which has to hold before and after each loop iteration, and a loop variant to show that the loop is terminating. These specifications build the pre- and postcondition of the repetition rule and its children. The chosen loop guard checks whether the current element variable k is pointing at equals the value x that is to be found. With that, the loop body is executed as long as this is not fulfilled. As invariant, we define that in the interval of array A that was already checked, we did not find value x. For this, we are using a negation of the predicate app. The variant for the loop is set to k, as it is decreasing to zero with every loop iteration and, thus, ensures that the loop is terminating. The abstract statement S_{body} is refined with another *assignment rule*, decreasing the iterator variable k by one. Finally, we return the current value of pointer k in S_4. With that, the implementation is complete. The verification of the side conditions of the individual rules proves that the implementation is correct w.r.t. the starting specification.

2.3 Tool Support

Formally specifying the desired behavior of a program is a first step towards correct code. However, defining specifications results in an increasing effort in comparison to common development techniques. Therefore, prior work focused on making the process of CbC more usable and efficient.

[3]Formal definition:
$app(arr, x, begin, end) = \exists\, int\ q\ (q \geq begin \land q < end \land arr[q] = x)$

[4]Formal definition: $sorted(arr) = \forall\, int\ y\ (0 \leq y \land y < len(arr) - 1 \rightarrow (arr[y] \leq arr[y+1]))$

[5]We iterate over array A from the end, as this simplifies finding a variant for the loop that is introduced later.

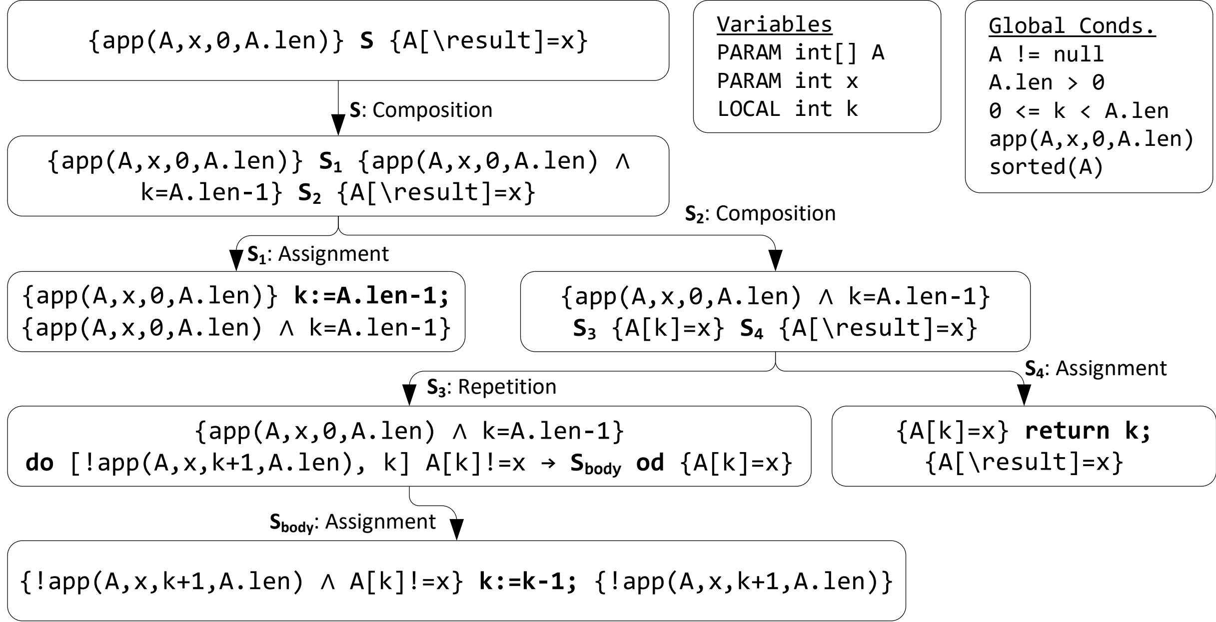

Figure 2. CbC-implementation of the *linear search* algorithm shown in a tree-structure, adapted and simplified from [72]. The nodes of the tree represent the applied refinement rules. The starting specification is propagated through the tree. Further formal specifications are defined for S, S_2, and S_3.

In 2019, Runge et al. [72] developed tool support for CbC. The open source Eclipse-plugin CorC[6] allows constructing Java-programs in a graphical editor and checks the applied refinement rules for their side conditions using the program verifier KeY [2]. In 2019, a user study compared program construction and verification using CbC against development using post-hoc verification [73]. The participants highlighted the guided construction process of CbC and the usability of the tool CorC. However, getting used to the development style of CbC was detected as hurdle. Thus, in recent work, Bordis [8] introduced a concept of three levels of correctness guarantees *specified - tested - verified* to CorC. The concept aims to lower the entry hurdle for developers by enabling automatic testing of (parts of) programs as intermediate step before verifying them. To reduce the manual effort for developers, Kodetzki et al. [47] investigated on combining CbC and the advantages of large language models (LLMs). The authors present their vision of LLMs in by-Construction engineering in terms of increasing efficiency and comprehensibility to simplify and speed up the CbC development process, as well as supporting inexperienced developers by applying suitable refinement rules automatically, generating formal specifications based on natural language descriptions, and providing an interpretation functionality for different programming artifacts, such as specifications, counter examples, or failed proofs. For this, a lightweight version of CorC was implemented as a web-framework called WebCorC[7].

[6] https://github.com/KIT-TVA/CorC, [accessed 2025/04/03]
[7] https://corc.informatik.kit.edu, [accessed 2025/04/03]

2.4 Scaling CbC

CbC as introduced by Kourie and Watson [48] aims for constructing algorithmic independent problems. However, this does not represent the size of systems developed in reality. To counter the missing support for large-scale systems, Bordis et al. [9] extended CbC and CorC by supporting object-oriented programs as well as product lines [11]. Further, a roundtrip engineering process for Java-programs was enabled [9, 10]. It allows importing existing programs, semi-automatically implementing them using CbC, verifying them with KeY, and exporting them back into the existing project as provably correct software. This also allows to implement systems consisting of parts implemented using CbC and parts not developed using CbC. To abstract CbC from code level, ArchiCorC [46] scales CbC to component level, defining *contract compatibility*, which allows flexibility in implementing interfaces of components with different functionality and, thus, scaling CbC to larger systems.

3 X-by-Construction: State-of-the-Art

In the context of the 2018 ISoLA-track *X-by-Construction*, ter Beek et al. [80] defined X-by-Construction (XbC) as "a step-wise refinement process from specification to code that automatically generates software (system) implementations that by construction satisfy specific non-functional properties". This definition leaves room for interpretation, which leads to diverse work in different directions. In the context of the 2018 ISoLA track, as well as a subsequent ISoLA track

in 2020 [79] and the European project *XANDAR* [7, 23, 55], numerous articles have been published that use the term X-by-Construction. However, only very few meet the original idea of XbC, the extension of CbC by non-functional properties. With regard to the original idea, existing work is limited to *Confidentiality-by-Construction* [76] and *Information Flow Control by-Construction* [70, 71], which can be used to ensure information flow policies with the help of typed refinement rules. In the following, we introduce these approaches in detail. Work using the term *X-by-Construction* that does not relate to the above interpretation is not explained further, as it is not relevant to this paper.

Confidentiality-by-Construction. *Confidentiality-by-Construction* (C14bC) was presented in 2018 by Schaefer et al. [76]. The authors introduce a security policy to CbC to ensure confidentiality using information flow. Schaefer et al. classify variables of CbC programs into *high-security* and *low-security* variables. The policy defines that information might flow between variables of the same security class and from variables of class *low* to variables of class *high*, but is never leaked from *high* variables to *low* variables. C14bC considers both, direct leaks, i.e., the assignment of a *high* variable to a *low* variable, and indirect leaks which can occur when evaluating guards of selections or loops, where the value of a *high* variable might determine the control flow of a program and, thus, is leaked. The authors present refinement rules similar to the rules of functional CbC (see Section 2.1) to enforce their security policy. Using these rules, classifications of variables can be propagated throughout the construction process, and the information flow policy is guaranteed to be met.

Information Flow Control-by-Construction. Based on C14bC, Runge et al. [71] introduced *Information Flow Control-by-Construction* (IFbC). It extends the aspect of security in CbC-programs with a second dimension, integrity. The authors present a lattice-based information flow policy, where different security levels represent the allowed direction of information flow, similar to the levels of C14bC [71]. Using user-defined security levels and the option to declassify variables at appropriate points in a program, it is possible to construct programs using by-Construction engineering that guarantee confidentiality and preserve integrity. The latter focuses on prohibiting information flow from untrusted to trusted variables, e.g., in a banking system. In 2022, the approach of IFbC was extended to also support object-oriented projects (IFbCOO) [70]. Both concepts, IFbC and IFbCOO, are implemented in CorC (see Section 2.3) and can be used next to CbC.

The background on functional CbC (see Section 2.1) and the above explanations show that apart from ensuring functional correctness and information flow properties, no other software quality properties can be ensured in by-Construction engineering in the current state. For this reason, we want to generalize XbC by integrating further non-functional properties to shape the future of software engineering. We envision a general approach for developing high-quality software fulfilling functional correctness guarantees as well as non-functional properties already in the implementation phase.

4 Energy Efficiency-by-Construction

In times of energy crises, climate change, and rapidly increasing numbers of mobile devices, developing energy efficient software is essential not only for reducing environmental impact, but also for enhancing device performance and providing a competitive edge in an increasingly sustainability-focused market [42]. Therefore, we consider performance parameters as in energy efficiency as the first non-functional property alongside security. In the following, we sketch how to use by-Construction engineering for constructing energy efficient programs. We introduce symbolic energy annotations for the *linear* search algorithm presented in Section 2.2 and present a second implementation of a search algorithm to show possible applications of our approach. Further, we give an overview of related work in the field of performance properties of software systems and position our approach.

4.1 Energy Annotations in by-Construction Engineering

In the following, we present our idea of energy annotations to track the energy consumption of programs and program statements. Energy annotations serve as a first step towards introducing performance efficiency as defined in ISO 25010[8] into by-Construction engineering.

Our concept of energy annotations is abstracted and refined from existing techniques of static cost analysis [3] and type systems for energy consumption [82] to fit the setting of by-Construction engineering. The energy annotations are the first step of integrating performance properties into by-Construction engineering. Thereby, we abstract from the underlying system, i.e., system- and hardware structure, to maintain generalizability. The energy annotations can be derived on program level and can be interpreted as *energy contracts*, i.e., expressions about which amount of energy is consumed between two program states. This is of advantage for comparing programs with the same functionality, but different implementations, i.e., taxonomies of algorithmic families [49, 84]. Using energy annotations, it is possible to determine which parts of an algorithm could be improved regarding their energy efficiency.

We use uninterpreted, abstract symbols to indicate the energy consumption at a certain program point. Using the abstract, hardware-independent symbols E_w, E_r, E_{binop}, and E_{ret} for writing, reading, binary operations, and returning values, we are able to make fine-grained statements about the energy consumptions of the individual statements of a

[8] https://www.iso.org/standard/78176.html, [accessed 2025/04/01]

CbC-program. At a later stage of the development process, the symbols can be mapped to a specific energy-model, i.e., concrete values for the uninterpreted E's, or concrete figures can be derived from measurements on concrete hardware. For this, a more detailed distinction of abstract symbols is planned to achieve the highest possible conformance with common energy models. However, for the presentation of our ideas in this paper, we only use the basic symbols introduced above to maintain readability.

Further, literature offers plenty of alternatives to our uninterpreted, symbolic energy annotations that need to be discussed in future work:

- Average energy consumption: calculation of the average consumption of loops and selections instead of upper bounds.
- Intervals: definition of both, upper and lower bounds of consumed energy [52].
- Energy levels: definition of levels of energy consumption, e.g., low, medium, high [17].

In our XbC-approach, each functional specification is extended by energy annotations. They represent energy program states prior, respectively after a certain refinement. The energy annotations are compositional, i.e., they can be combined while traversing the CbC-program. In general, the construction of a program is still based on functional CbC. To aim the construction only focusing on energy efficiency without considering functional correctness would end up in an empty program as this consumes no energy, but is also not implementing any functionality. We use the advantage of extending the refinement rules of functional CbC by energy annotations to be able to give guarantees about energy efficiency directly with the application of the refinement rules of CbC.

In the following examples, the application of energy annotations is split into two parts. First, when applying the refinement rules of functional CbC, energy annotations are derived for the respective program statements introduced in a top-down process. In a second step, the annotations are evaluated as soon as a (part of the) program is implemented completely. At this point, we can make a statement about the energy consumed by instantiating the symbolic energy annotations with a concrete energy model and can compare certain parts or complete programs with other implementations of the same algorithmic family. Both steps of the process of applying energy annotations follow a clearly defined procedure, as the following explanations show. This enables simple automation, for example through tool support.

Example: Linear Search. In Figure 2, we presented the CbC-implementation of the *linear search* algorithm. In the following, we introduce symbolic energy annotations to this implementation. The result is shown in Figure 3. The functional specifications are noted on the side for readability.

In general, before executing a program ①, we start with the energy annotation $\Gamma \cong 0$, meaning that no energy is consumed so far. The energy annotation Δ shown in the top box can only be derived completely as soon as the whole program is constructed, as described above. With no or an incomplete implementation, Δ is unspecified, respectively incomplete, only consisting of the energy annotations of already implemented program parts.

For the construction of the *linear search* algorithm, a set of refinement rules is applied, based on the functional specification, as described in Section 2.2. After applying the *composition rule* and the *assignment rule* to the abstract statements S and S_1, energy annotations for the initialization of variable `k` can be defined ②. The initialization takes energy for reading the length of the array `A`, performing one binary operation, and writing the result to the variable `k`. Thus, the energy annotations are the sum of these operations, $\mathrm{E}_r \oplus \mathrm{E}_{binop} \oplus \mathrm{E}_w$.[9] This is propagated back to the *composition rule* and serves as intermediate energy consumption ③. The right branch of the *composition rule* consists of the loop and the return statement of the program. The energy annotations Φ of the *repetition rule* ④ express the energy consumption of the loop body S_{body} (reading variable `k`, subtracting the constant `1`, and writing to `k`) including the evaluation of the loop guard (reading iterator `x`, reading the value of array `A` at position `k`, and comparing it with each other) added to the energy consumed so far, Σ. For every loop, the sum of the energy annotations of the loop body and guard evaluation has to be multiplied by the number of loop iterations. For now, our approach of energy annotations derives an upper bound of the consumed energy of a program. Therefore, we introduce n as the length of the array `A` as an upper bound for the number of loop iterations. The last part of the program, the return statement, is annotated with the consumption of one reading and one return operation $\mathrm{E}_r \oplus \mathrm{E}_{ret}$ ⑤. This is added to the energy annotations that are propagated to statement S_4 from the *composition rule*. With that, the implementation is finished, and the energy annotation Δ, describing the consumption of the whole program, is complete.

Example: Binary Search. As described above, the idea of energy annotations can find application in software taxonomies [49, 84], where different programs implement the same functionality, i.e., fulfill the same formal specification. To illustrate how this works, we present the implementation of the *binary search* algorithm as an alternative to *linear search* in Figure 4. In contrast to *linear search*, the *binary search* algorithm requires the array to be sorted. As we required array `A` to be sorted increasingly for the *linear search*

[9] For brevity, all arithmetic operations are handled the same and, thus, result in the same energy consumption. Future work on our approach will differentiate the operations, as this corresponds to the instruction sets of real-world systems and the energy models our symbolic energy annotations can be instantiated with.

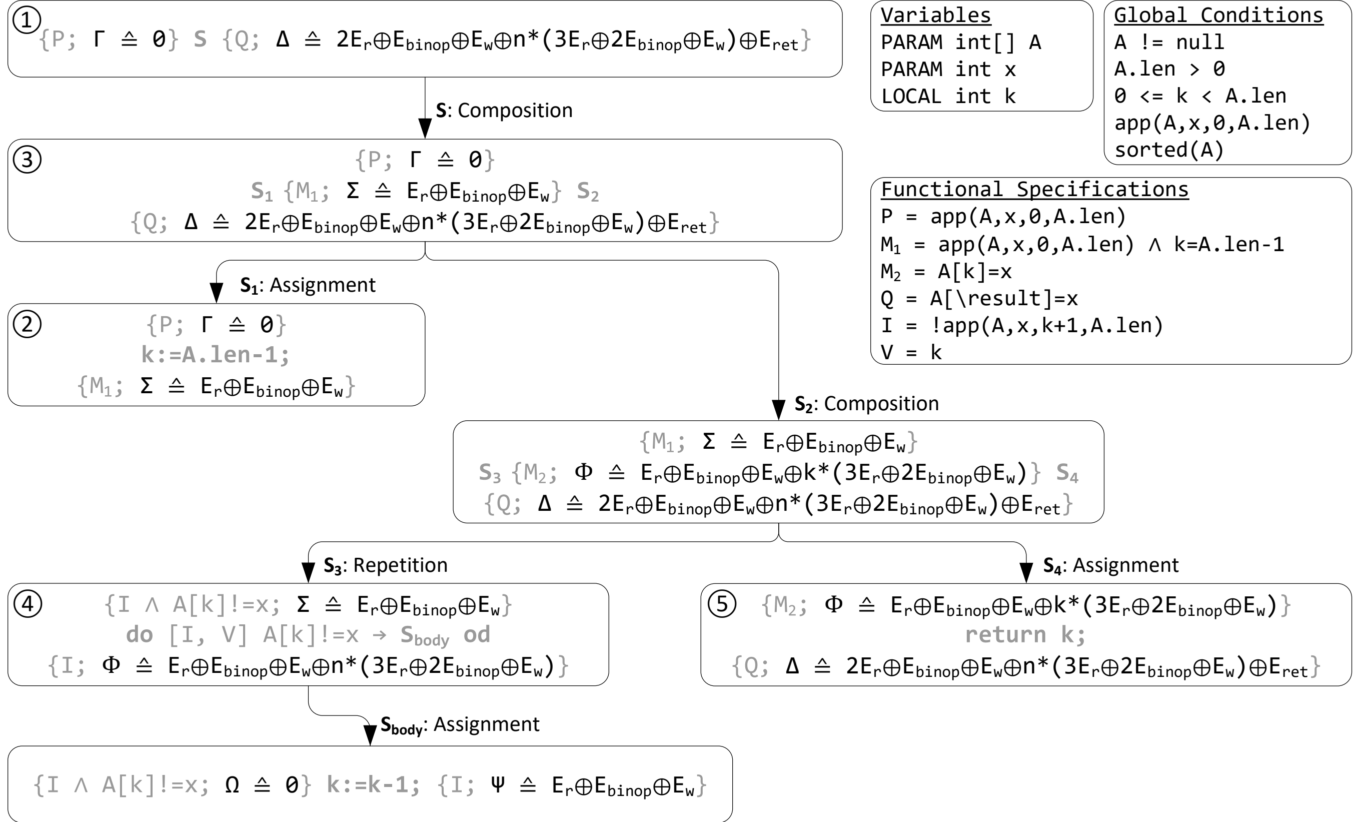

Figure 3. Implementation of the *linear search* algorithm using CbC including energy annotations. To maintain readability, functional specifications are noted on the side. The energy annotations can be derived from the applied refinement rules automatically.

algorithm using a global condition (see Figure 2), we can use the same specification for constructing both algorithms. This represents common use cases of software taxonomies in practice.

Binary search as implemented in Figure 4 repeatedly divides array `A` in half, comparing the target value `x` to the middle element, and narrowing the search to the half where the target could exist until it is found. For the sake of brevity, we do not explain the construction following the rules of functional CbC in detail. However, we provide the intermediate conditions of the composition rules, the guard, variant, and invariant of the loop, and the guards of the selection of the *binary search* implementation on the right side of Figure 4. Further, in the following, we explain the derivation of the energy annotations in detail.

The process of deriving symbolic energy annotations for the *binary search* algorithm, as shown in Figure 4, is similar to that of the *linear search* algorithm (see Figure 3). The initialization of the variables `left` and `right` describing the bounds of the partial array currently considered ① results in the interim energy annotation $E_r \oplus E_{binop} \oplus 2E_w$. The right branch of the first composition, statement S_2, implements a loop that iterates over sections of the array `A` that bisect every iteration. Depending on the read value, either the left or right half is considered for the next iteration. With this approach, the maximum number of loop iterations is $log_2(n)$ with n being the length of the array `A` ②. Inside the loop body, a selection determines whether the value the algorithm is searching for is found, or which half of the partial array currently considered is continued on. Thus, the selection has three possible outcomes. As we cannot determine statically (but only with concrete inputs) which branch will be taken, we are continuing with the energy annotation $max(S_A; S_B; S_C)$, stating that the maximum energy consumption of the three branches has to be considered ③. When deriving the energy annotations for the second and third branches, we have to keep in mind that when reaching these branches in a program flow, the previous guards have been evaluated. Thus, their energy consumptions have to be integrated in later branches. When finishing the construction of the algorithm, energy annotation Δ of the top box is completed as well.

Results. Following the above examples, we obtain the symbolic energy annotations shown in Equation 1. As described

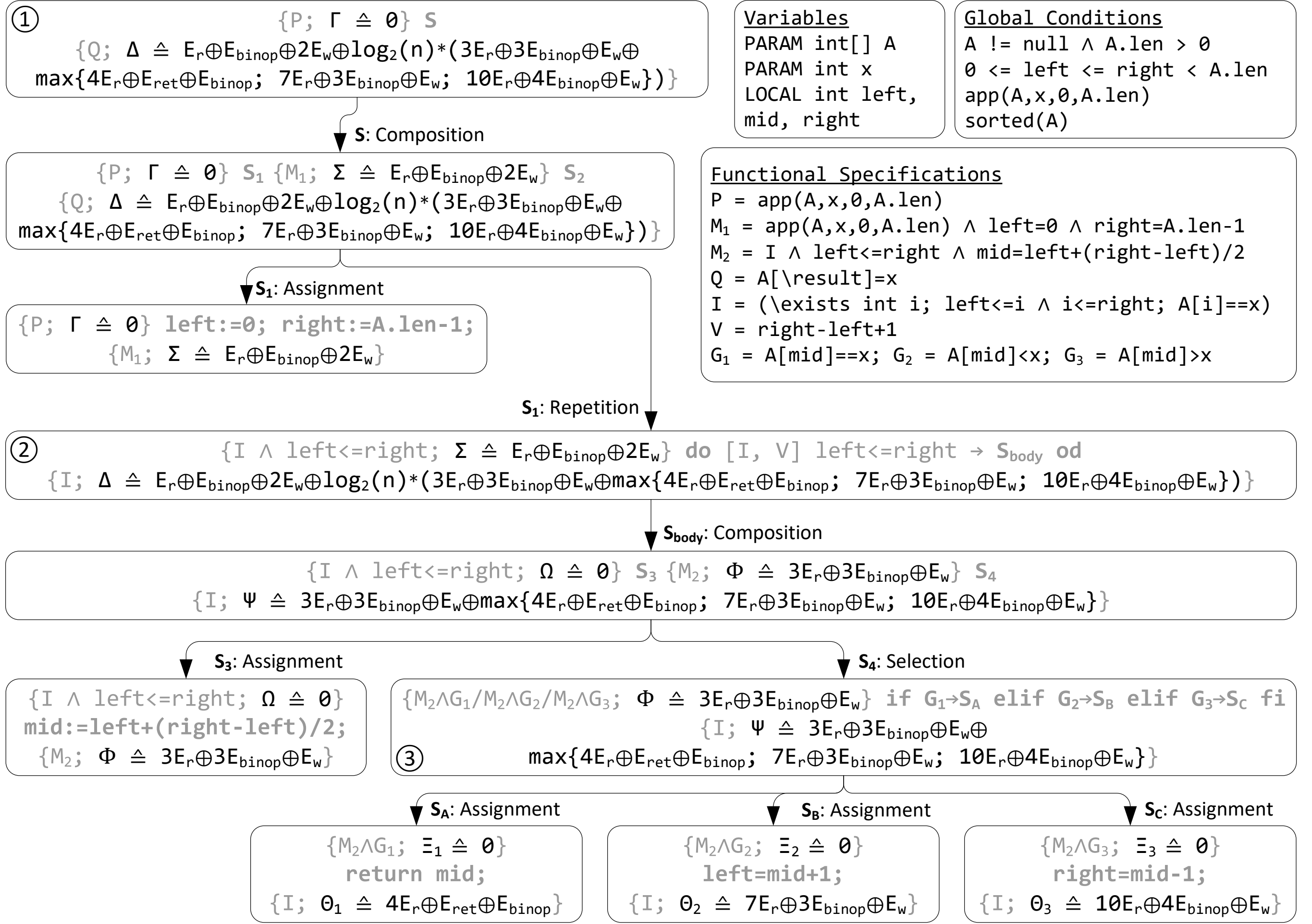

Figure 4. Implementation of the *binary search* algorithm using CbC including energy annotations. To maintain readability, functional specifications are noted on the side. The energy annotations can be derived from the applied refinement rules automatically.

in the introduction of this section, these annotations are independent of the underlying system and hardware structure. However, we are able to instantiate the symbolic energy annotations with a concrete energy model to receive concrete values of their energy consumption when executed and, thus, allow a comparison of the algorithms energy efficiency. In Equation 2, we show an exemplary instantiation of the derived energy annotations of Equation 1 using a programming-oriented PTX instruction level energy model [87]. In Table 1, we show the energy consumption of the individual instructions of the energy model. For brevity, we only show instructions that are relevant in the context of our approach. Further, as described above, we do not differentiate between the energy consumptions of arithmetic operations for now. For this exemplary instantiation, we use their mean. In future work, a detailed distinction is to be introduced.

By replacing the symbolic E's of the energy annotations shown in Equation 1 with the values of the energy model, only the approximations for the loops (n for *linear search* and $log_2(n)$ for *binary search*, with n being the length of array A) stay uninterpreted. Thus, concrete input values for array A are the only dependencies left. With exemplary input values of $A = \{8, 17, 24, 33, 56, 68, 82, 97\}$, resolving the energy annotations of Equation 2 leads to an energy consumption of 94.436 nj for *linear search*, and 122.048 nJ for *binary search*, w.r.t. the considered energy model [87]. Still, these energy consumptions serve as upper bounds, as we did not define exact but approximated variants for the loops. Further, we are yet not able to determine which branches of the selection in the *binary search* implementation are taken, as described above. To receive exact values, one needs to execute the algorithms with concrete input values to determine the number of loop iterations and distribution of the branches of the selection.

Table 1. Excerpt of PTX instruction level energy model [87] with mapping to energy annotations. For E_{binop}, we calculate the mean of the instructions *add, sub, mul, div* for brevity.

Energy Annotation	PTX Instruc.	Energy [nJ]
-	add (int)	4,556
-	sub (int)	2,150
-	mul (int)	2,880
-	div (int)	8,099
E_{binop}	-	4,421
E_r	ld	0,4748
E_w	st	0,7243
E_{ret}	mov	0,4955

Besides the calculation of the concrete energy consumption of the algorithms, we can use the symbolic representation shown in Equation 1 to compare the implementations of the programs on abstract level. Using the refinement-based approach of by-Construction engineering, this is not only possible on method level, but, for algorithmic families where multiple implementations are constructed for the same specification, it is possible to just compare parts of the algorithms. Suitable examples can be found in the literature, especially in taxonomies of algorithms [49, 84].

To simplify the comparison of our examples, we can eliminate certain parts that are common to both energy annotations of the algorithms. Further, we can optimize the term $max(S_A; S_B; S_C)$ of the energy annotations of the *binary search* algorithm, as the execution of case C consumes at least the amount of energy that case B consumes. The results are summarized energy annotations, shown in Equation 3. Due to the summarizations, these energy annotations do not represent the actual consumption any more, thus, cannot be instantiated with some energy model, but can be used as representative approximations for comparison purposes.

4.2 Related Work

Energy efficiency is a well-known research topic. Driven by climate change and the optimization of mobile and embedded system to handle limited resources effectively, research in this field increased since the early 2000s. Nowadays, the term *GreenIT* [42, 59] is often used to describe not only the *sustainable development of software*, but also the *development of sustainable software*. Consequently, many established performance analysis techniques have been adapted or extended to address energy consumption of software. In addition to the measurement of the performance of a system at runtime [14, 41, 51, 54], program analysis offers ways to determine performance exactly or as approximation statically, i.e., without executing code [3, 22, 28, 32, 81]. Though being first ideas and not fully developed, our approach of symbolic energy annotations contributes to the field of software performance by providing an abstract, symbolic method to

$$\Delta_L \equiv E_{\text{ret}} \oplus 2E_r \oplus E_{\text{binop}} \oplus E_w \oplus n * (3E_r \oplus 2E_{\text{binop}} \oplus E_w) \tag{1}$$

$$\begin{aligned}\Delta_B \equiv\ & E_r \oplus E_{\text{binop}} \oplus 2E_w \oplus log_2(n) * (3E_r \oplus 3E_{\text{binop}} \oplus E_w \oplus \\ & max\{4E_r \oplus E_{\text{ret}} \oplus E_{\text{binop}}; \\ & \quad 7E_r \oplus 3E_{\text{binop}} \oplus E_w; \\ & \quad 10E_r \oplus 4E_{\text{binop}} \oplus E_w\}) \quad (2) \\ \equiv\ & E_r \oplus E_{\text{binop}} \oplus 2E_w \oplus log_2(n) * (7E_r \oplus 4E_{\text{binop}} \oplus E_w \oplus \\ & max\{E_{\text{ret}}; \\ & \quad 3E_r \oplus 2E_{\text{binop}} \oplus E_w; \\ & \quad 6E_r \oplus 3E_{\text{binop}} \oplus E_w\}) \quad (3)\end{aligned}$$

Equation 1: Comparison of total symbolic energy annotations of the implementations of *linear* (Δ_L) and *binary* (Δ_B) search.

$$\begin{aligned}\Delta_L^{PTX} \equiv\ & 0,4955 \oplus 2 * 0,4748 \oplus 4,421 \oplus 0,7243 \oplus \\ & n * (3 * 0,4748 \oplus 2 * 4,421 \oplus 0,7143) \quad (4) \\ \equiv\ & 6,5904 \oplus n * 10,9807 \quad (5) \\ \Delta_B^{PTX} \equiv\ & 0,4748 \oplus 4,421 \oplus 2 * 0,7243 \oplus \\ & log_2(n) * (7 * 0,4748 \oplus 4 * 4,421 \oplus 0,7243 \oplus \\ & max\{0,4955; \\ & \quad 3 * 0,4748 \oplus 2 * 4,421 \oplus 0,7243; \\ & \quad 6 * 0,4748 \oplus 3 * 4,421 \oplus 0,7243\}) \quad (6) \\ \equiv\ & 6,344 \oplus log_2(n) * (21,7319 \oplus \\ & max\{0,4955; 10,9907; 16,8361\}) \quad (7) \\ \equiv\ & 6,344 \oplus log_2(n) * 38,568 \quad (8)\end{aligned}$$

Equation 2: Instantiation of symbolic energy annotations from Equation 1 of the implementations of *linear* (Δ_L^{PTX}) and *binary* (Δ_B^{PTX}) search with n being the length of array A using the energy model of Zhao et al. [87] shown in Table 1. All values are given in 10^{-9}J (nJ). For arithmetic operations, we use the mean of the instructions *add, sub, mul, div* for brevity.

$$\Delta_L^* \equiv E_{\text{ret}} \oplus E_r \oplus n * (3E_r \oplus 2E_{\text{binop}} \oplus E_w) \tag{9}$$

$$\begin{aligned}\Delta_B^* \equiv\ & E_w \oplus log_2(n) * (7E_r \oplus 4E_{\text{binop}} \oplus E_w \oplus \\ & max\{E_{\text{ret}}; \\ & \quad 6E_r \oplus 3E_{\text{binop}} \oplus E_w\}) \quad (10)\end{aligned}$$

Equation 3: Summarized symbolic energy annotations of the implementations of *linear* (Δ_L^*) and *binary* (Δ_B^*) search for comparison purposes. These energy annotations cannot be instantiated with concrete energy models.

determine and compare the energy consumption of (parts of) programs. By situating our approach alongside other performance metrics and software quality properties, it supports the broader goal of a holistic approach for developing high-quality software. In the following, we present main related work from the software performance community focused on, but not limited to, energy consumption. We split existing techniques into dynamic and static approaches.

Dynamic Approaches. *Profiling* and *tracing* are widely used techniques to analyze the performance of software during execution. Profiling provides a statistical overview of where an application spends its resources, focusing on function or code block level [5, 51]. Tracing, on the other hand, offers a detailed, chronological view of the execution flow, allowing to track the path of operations through the system [31, 38]. Both, profiling and tracing, require the considered code to be executed. Our approach of energy annotations is applied at construction time and, thus, does not require execution.

Benchmarking serves as a cornerstone of performance evaluation in computer systems research, providing standardized methodologies for comparing architectures, software, and workloads [21]. Benchmarking is useful for comparative evaluation, however, it requires runtime instrumentation, and does not generalize well to unseen programs or inputs. Still, benchmark suites such as *SPEC CPU* [36] and *TPC benchmarks* [67] have enabled rigorous evaluation of processor and database performance, respectively. In more recent work, benchmark suites capture the performance characteristics of data-intensive and AI-driven workloads [24, 68], facing the emerging challenges of computer systems. For our use-case, energy consumption, systems like *Green Miner* [37] or *perun* [30] can be used to evaluate software energy usage under standardized conditions. As for all dynamic approaches, benchmarking requires execution of code and, therefore, differs from the static concept of energy annotations.

Static Approaches. *Static estimation techniques* aim to predict performance properties without executing code. Early work on energy consumption includes instruction-level energy models derived for specific *ISAs* and symbolic techniques such as *SimplePower* [86]. More recent approaches like *energy analyzers* [83] combine control-flow analysis with hardware-specific models to perform estimations statically. These approaches reduce the execution dependency but often require fine-tuned ISA-level models and do not support fully symbolic inputs. Our work builds on this research direction by offering symbolic, hardware-independent estimations that abstract away the underlying system and concrete input values.

Asymptotic analysis is used for understanding how algorithmic and system efficiency scales with input size and resource count [28, 57]. It provides machine-independent bounds on time, space, or energy consumption. Asymptotic analysis can operate at multiple levels, but most commonly, it is applied at the algorithmic, rather than at the level of concrete code as it is done in our approach of energy annotations. In the context of energy, asymptotic models can estimate computational intensity and memory access patterns allowing developers to reason about energy efficiency alongside performance [69]. This form of analysis is especially valuable in early design phases, where it evaluates architectural decisions and algorithm selection without requiring full implementation or execution. Asymptotic analysis is similar to our concept of energy annotations, as it is performed hardware-independent and can be applied at early stages of development. Further, it is oftentimes applied to reason about algorithmic complexity and design choices, just as energy annotations. However, the results of asymptotic analysis are not as fine-grained as energy annotations. Also, the estimations are usually not as exact as the one's of our concept and only represent an algorithm's complexity without statements on its energy consumption.

Beyond measuring or analyzing the energy efficiency of a software system at runtime, more *formal approaches* consider resource consumption in general to ensure that certain limits are not exceeded. Techniques vary from logics [43] to type systems [25, 82] and proof systems [17, 62]. Among these approaches, common hurdles are dependencies to the programming language and their execution environments [65], and the impact of hardware components [34]. Depending on size and useload of a system, consumption parameters, energy and time in particular, can vary at runtime. These dependencies are not present for correctness and information flow properties currently part of by-Construction engineering. However, they need to be considered for extensions of XbC by performance properties as we discuss later.

Design-time performance optimization involves transforming software to reduce costs, often guided by profiling or static analysis. In the context of our use case, energy consumption, existing techniques range from compiler-level optimizations, e.g., loop unrolling, to energy-aware scheduling [6]. Research on *GreenIT* [59] and *green refactoring* [66] reflects a growing interest in software-level energy design. Our symbolic approach enables optimization guidance during and after program construction, and can be applied without runtime measurements as is common for the above approaches.

5 Generalizing X-by-Construction

Based on the experiences that were made with X-by-Construction (see Section 3) and our ideas for ensuring energy efficiency in by-Construction engineering (see Section 4), we define requirements for integrating non-functional properties into XbC. With respect to these requirements, we evaluate the suitability of XbC to guarantee quality properties as described in ISO 25010 by construction.

5.1 Requirements for Extending X-by-Construction

To be able to integrate non-functional properties into XbC, a quality property has to fulfill certain requirements. The requirements are derived from the basic idea of by-Construction engineering as well as the currently supported properties, functional correctness and information flow. We analyzed what aspects are common for the supported properties and what criteria the refinement process of by-Construction engineering requires. In general, the set of requirements can be considered minimal in the sense that each is necessary for ensuring that a property can be integrated into the XbC process. Each requirement supports a critical aspect of XbC, such as compositional reasoning or refinement preservation. Removing any one of them would undermine the guarantees provided by XbC or compromise its applicability to software development.

- R1 A property must be specifiable, as a specification is the basis in by-Construction engineering. It can be an advantage if a property is quantifiable, as this can simplify the specification. However, a quantifiable specification can increase the complexity of the reasoning about it. Specifications should be abstract and independent of the underlying system and hardware structure such that they can be instantiated at a later stage of development to maintain generalizability during construction.
- R2 It must be possible to establish the specification of a property without knowing of the actual structure of a program, i.e., what procedural programming constructs will be used or what components coexist, as this information is not available when defining a specification in by-Construction engineering. Further, the specification should be definable without actually executing the program, as this is not possible at program construction.
- R3 A considered non-functional property must not interfere with another functional or non-functional property. This allows to guarantee multiple properties for a single program during the same construction process. Exceptions, for example, for performance properties, must be well thought out and need strict definitions on how to handle them.
- R4 It must be possible to formally establish that the specification is fulfilled. For functional correctness, this is ensured by verifying the program against its formal specification. An equivalent static procedure must be available for non-functional properties to be able to show the validity of the specifications during program construction.
- R5 A property must be compositional, i.e., a property and its validity must be obtainable from parts of the property of the program and their validity. With that, we ensure that a program's property follows from the properties of its parts and that we do not need to complete the program construction to make statements about the validity of a property. Compositionality is to be ensured on program level. For handling crosscutting dependencies to other parts of a system such that they are not modular, assume-guarantee reasoning (AGR) [12, 15, 16, 27] can be used. Thereby, guarantees on certain properties are only given under certain assumptions regarding the system- and software structure. In a later stage of development, these assumptions are discharged.
- R6 The derivation of a non-functional property must be combinable with the refinement rules of CbC. Functional correctness has to be the basis of each implementation, as software that fulfills non-functional properties, but not the desired functionality, is generally not useful.

5.2 Assessment of Non-functional Properties in ISO 25010 for XbC-ability

Software quality as defined in ISO 25010[10] covers a wide set of quality criteria. For XbC, we focus on, but are not limited to, the non-functional properties defined in ISO 25010. In the following, we present these properties and their sub-properties, and evaluate whether they meet the requirements defined above. We do not consider *functional suitability* and *security*, as they are already covered by CbC, and XbC using information flow properties, respectively. Further, we do not discuss the suitability of the non-functional properties *flexibility* and *interaction capability*. These properties deal with criteria that are too far from the actual code that is constructed, as their assurance takes place at the organizational and management level rather than at the level of program construction. Thus, they are unlikely to be integrated into by-Construction engineering. A summary of the results is shown in Table 2. Check marks in brackets indicate that a requirement is fulfilled under certain circumstances.

Performance Efficiency. According to ISO 25010, *performance efficiency* is defined as the extent to which a product performs its designated functions while ensuring efficient utilization of system resources. This property consists of the sub-properties *time behavior*, *resource utilization*, and *capacity*.

A common advantage of resource properties is that the determination of the consumed amount of a resource can usually be done both, dynamically, i.e., by executing the program, and statically, e.g., by symbolic execution [3, 81, 82] (R2, R4), as discussed in Section 4.2. Extensive research in the field of *Quality-of-Service* on improving performance [13, 26] and other work of the performance community leads to the

[10] https://www.iso.org/standard/78176.html, [accessed 2025/04/01]

fact that, besides functional correctness, performance efficiency satisfies the most requirements among the properties defined in ISO 25010.

In general, most performance criteria, e.g., time, energy, and memory consumption, offer the advantage of quantifiability. This offers various ways of specification. An example is the approach of Knoth et al. [45], which introduced resource-guided synthesis to derive programs from a functional specification and symbolic resource bounds. Lopez et al. [52] presented an approach of defining intervals to describe the energy consumption of a program (R1). Our ideas of energy annotations presented in Section 4 show the option of combining the refinement rules of CbC and energy annotations as seen in the examples of *linear* and *binary search* (R6). If other sub-properties of performance efficiency, such as memory consumption or runtime, are taken into account in this context, trade-offs can occur, for example, that a program can be designed to be more memory efficient, but may therefore have a higher runtime [65] (R3). As discussed above, dependencies to hardware- or energy models can be abstracted, but need to be considered at a later stage of development, e.g., by instantiating the abstract symbols with concrete models. For this, as discussed in Section 4.1, the proposed energy annotations need to be refined to more fine-grained annotations to match the set of possible instructions of energy models and to receive more exact estimations. Besides hardware dependencies, cross-cutting concerns within a software system can occur that affect the modularity and, thus, compositionality of a property [44] (R5). Assume-guarantee reasoning is one approach to tackle these concerns. We discuss details on both, dependencies to hardware and cross-cutting concerns, in Section 6.2.

Reliability. In ISO 25010, *reliability* is defined as the extent to which a system, product, or component performs its intended functions under defined conditions for a specified duration. Reliability of software is particularly relevant in safety-critical systems and can be split in the following sub-properties: *faultlessness*, *availability*, *fault tolerance*, and *recoverability*. Reliability can be integrated into XbC as a probabilistic property [56], for example, by introducing failure probabilities, ensuring reliability can become relevant.

With respect to the defined requirements for integrating properties into XbC, in the literature, various approaches quantify reliability based on different metrics [40, 60, 61] (R1-2). Some of these metrics can be combined with the refinement rules of functional CbC as well as on component level, e.g., in ArchiCorC (R6). With some of the metrics, compositionality is given implicitly (R5), however, evaluating these metrics can become difficult [77, 78] (R3-4).

Safety. ISO 25010 defines *safety* as the extent to which a product can prevent situations that endanger human life, health, or the environment. It is composed of the main sub-properties *fail safeness* and *operational constraint*. Safety was

Table 2. Assessment of non-functional properties of ISO 25010. Check marks in brackets indicate that a requirement is fulfilled under certain circumstances.

Non-functional Property	Requirements					
	R1	***R2***	***R3***	***R4***	***R5***	***R6***
Performance eff.	✓	✓	(✓)	✓	(✓)	✓
Reliability	✓	(✓)	(✓)	(✓)	✓	✓
Safety		(✓)	✓		(✓)	
Compatibility			(✓)			(✓)
Maintainability	✓	✓	(✓)			✓

already considered long time ago, motivated by catastrophes such as the Ariane-5 disaster [63] or the crash of Lufthansa flight 2904 in 1993 [53].

Existing work provides approaches to assure the safety of software [1, 33], as well as analyzing the safety assurance in existing systems [39, 50]. However, quantification is a major hurdle and, thus, also establishing formal specifications (R1-2), a program's safety can be constructed from and evaluated against (R3-R6). Integration of safety as non-functional property into XbC requires preparatory work to tackle these challenges.

Compatibility. ISO 25010 defines *compatibility* as the degree to which a system is capable of exchanging information with other systems or components while executing its intended functions within a shared environment and utilizing common resources. Thus, it is not applicable directly to by-Construction engineering (R3). Scaling CbC to component level using ArchiCorC might offer opportunities to ensure compatibility between correct-by-construction components (R5-6). Open questions regarding specifications and a formal verification remain, as there is only very few literature on this aspect (R1-2, R4). Existing work mainly focusses on ensuring compatibility before or while designing software systems [18, 64].

Maintainability. According to ISO 25010, *maintainability* reflects the extent to which a system can be effectively and efficiently modified for the purposes of improvement, correction, or adaptation. It is composed of the sub-properties *modularity*, *reuseability*, *modifiability*, and *testability*. A commonly used methodology to determine maintainability is the *maintainability index* (MI) [85] which defines the grade of maintainability of a software based on different metrics. Further work considers maintainability models to assess the maintainability of software [35]. Based on these techniques, quantification, specification, and combination with the refinement rules of functional CbC might be feasible in the scope of XbC (R1, R5-6). However, it is questionable whether those metrics can be applied compositionally and if they can be ensured while constructing a program (R2-4).

In conclusion, besides the already integrated information flow properties, performance efficiency, reliability, and, to a certain degree, safety and maintainability, fulfill the requirements defined above. Compatibility and the properties that we did not consider in this discussion, flexibility and interaction capability, are unlikely to be integrated into XbC, as they do not meet certain requirements necessary for a successful integration into by-Construction engineering.

6 Our Vision

In the previous section, we evaluated the suitability of the non-functional properties of ISO 25010 for a constructive software development process. Following the existing approaches for functional correctness and information flow properties, we want to generalize X-by-Construction for enhancing future software engineering.

We envision XbC as a software development approach for constructing high-quality software and imagine it as a method to develop software that fulfills correctness guarantees as well as non-functional guarantees already in the development process. Thus, the quality of the software will be guaranteed as early as possible.

Driven by the main application areas of functional CbC, safety-critical systems, generalizing XbC by supporting further non-functional properties will not only improve these systems, but pave the way for developing high-quality software for all domains that can benefit from using by-Construction engineering. To fit the individual needs of different systems, we envision a software development paradigm, where developers are able to set the focus to one or another property or trade-offs. Aligned with abstract, thus, hardware-independent, specifications for the chosen non-functional properties, a program will be constructed using the set of CbC-refinement rules without being limited to certain systems, ensuring functional correctness and the selected non-functional properties.

We define a procedure for integrating non-functional properties into by-Construction engineering. The steps for the integration of non-functional properties are as follows:

1. Development of a specification for the considered non-functional property. Depending on the characteristics of the non-functional property, this can be done symbolically or by quantification. In general, the format of the specification needs to be independent of the structure of a program.
2. Establishment of validity, such that we can ensure that the non-functional property can be validated during the construction of a program.
3. Establishment of compositionality on program level, such that a program property follows from properties of its parts. A property might need to be localized and modularized by assume-guarantee reasoning. In general, the property needs to be relativized towards an abstract, symbolic level [75].
4. Combination of the specifications for the non-functional property with the refinement rules of functional CbC which serves as the basis for every program construction, such that side conditions preserve the validity of the specifications.
5. Proof of soundness of formalized refinement rules w.r.t. specification and validity.

6.1 Tool Support

To be able to use our envisioned software development paradigm for high-quality software development in practice, we imagine tool support, called XCorC. For reasons of accessibility and usability, XCorC shall be based on the current web-framework, WebCorC (see Section 2.3). Currently, WebCorC supports software development using CbC in an interactive web-framework. For XCorC, we envision a parameterizable, extensible development environment in which non-functional properties can be considered in addition to functional correctness using by-Construction engineering. Developers will be able to determine properties that fit their use-case by selecting them in the web tool, following our vision of XbC described above. Thereby, we aim to keep the abstract, symbolic representation of non-functional properties, especially performance properties, to enable a hardware-independent construction of programs.

XCorC will integrate functionality of the existing Eclipse-plugins CorC and ArchiCorC to facilitate integration into existing projects using the round-trip engineering process on both single software program and component level. Further, we aim to extend existing approaches on supporting projects that consist of CbC- and non-CbC parts. This is to broaden the applicability in practice and to offer higher suitability for real-world software systems.

Recent and ongoing work on simplifying and increasing efficiency of the CbC-process using both, static methods, such as testing as first step towards correct software, and LLM-based support, are to be continued and integrated into XCorC. This will result in a tool that is designed in a way that provides opportunities for all developers needs to build high-quality software in a refinement-based environment to leverage future software engineering.

6.2 Discussion

In this section, we discuss challenges and issues arising when integrating energy consumption and other non-functional properties of software quality into XbC.

Limitations of Energy Annotations. Our ideas towards energy consumption in XbC result in symbolic, hardware-independent energy annotations which can be used to compare programs from algorithmic families and can be instantiated with a concrete energy model. The examples of *linear*

and *binary* search and the assessment of performance efficiency as non-functional property in XbC uncovered certain limitations of our approach. The first limitation concerns mapping the energy annotations to energy models. For now, the chosen symbolic energy consumptions are very coarse-grained, e.g., E_{binop}, and need more detailed distinction and generalization to receive sufficient results when instantiating them with any energy model. Further, we need to take certain dependencies into account. Commonly, the energy consumed by some program highly depends on the structure of the system it is executed on and cannot be evaluated as modular as it is done in CbC and IFbC for functional correctness and information flow properties, respectively. For these approaches, specifying, constructing, and proving the desired properties is possible symbolically in a modular way on program level without dependencies to the remainder of a system. In the context of other non-functional properties, different issues, e.g., virtualization, caching, pipelining, or nondeterminism, arise which need to be handled in a uniform manner [15, 44]. These properties need to be abstracted from the underlying system to reason about them as information on hardware is not available at construction time. The abstract symbols are only to be concretized when instantiating them with a specific hardware-model.

A possibility to address the dependencies to other program parts is assume-guarantee-reasoning (AGR). The assumptions of AGR can be discharged at a later point in time. Dependencies to the underlying hardware or specific languages are already addressed in our concept of energy annotations by abstraction and delayed concretization when instantiating the abstract symbols. Defining concepts for specific languages or systems could streamline the application of performance properties in by-Construction engineering even further, however, simultaneously restricting the applicability to certain systems. Current experiments focus on a specific, platform-dependent extension for performance properties and investigate on introducing CbC and XbC to the low-level programming language *Pancake*[11].

Application in Practice. In Section 2.4, we discussed the applicability of by-Construction engineering in practice. We introduced ArchiCorC which scales CbC to component level, and presented tool support for CbC that allows constructing large scale systems, such as object-oriented projects and product lines. We aim to use these approaches as a basis to build our concepts and tooling for future XbC on. This is to enable constructing not only single programs but real-world systems using CbC and XbC. By considering ArchiCorC, we also allow the construction of systems consisting of CbC and non-CbC parts. Recent work on scaling IFbC shows how non-functional properties in by-Construction engineering can be scaled to component level [74], building a first step towards our vision of using XbC in real-world scenarios.

[11] https://trustworthy.systems/projects/pancake, [accessed 2025/06/30]

Usability of by-Construction Engineering. A critical concern of by-Construction engineering is its usability. By extending XbC, the process of constructing programs using CbC and XbC is becoming increasingly complex. Thus, we need to ensure that the process stays comprehensible and does not demand disproportionate effort, in the worst case exceeding the effort needed in traditional software-quality assurance techniques. As presented in Section 2.3, recent work focuses on enabling CbC development in CorC for unexperienced developers by introducing *testing* as intermediate level between *specified* and *verified* [8], and LLM-based approaches aim to automate tasks of the development process to speed up constructing programs using CbC [47]. However, for future extensions of XbC and its tool support, we need to ensure that specification and implementation effort maintain feasible. Otherwise, by-Construction engineering might not be applicable in practice. Thereby, the main focus is on tool support and the automation of development steps.

7 Conclusion

CbC is an approach to incrementally develop functionally correct programs using sound refinement rules. XbC is an extension of CbC covering the by-Construction engineering of software with guaranteed non-functional properties. So far, XbC has only been realized for security using information flow properties. In this paper, we presented our vision of generalizing XbC for developing high-quality software using by-Construction engineering, not only guaranteeing functional correctness, but selected non-functional properties. We proposed our vision of the generalization of XbC to support multiple aspects of software quality, resulting in a comprehensive software development technique. We discussed which requirements have to be fulfilled to enable XbC for a non-functional property and defined a roadmap on how to integrate further non-functional properties. Further, we addressed challenges arising when extending XbC. We discussed our vision of accessible, extensible tool support for future software development using by-Construction engineering. Finally, we showed first ideas of symbolic energy annotations in CbC-programs to support energy efficiency as non-functional property going beyond information flow. Using these energy annotations, we are able to compare the programs and program parts of algorithmic families regarding their energy efficiency. Ongoing and future work focuses on the formalization of energy annotations, the generalization of XbC by supporting further non-functional properties, performance efficiency and reliability in particular, and addressing the challenges discussed in this paper.

Acknowledgments

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

References

- [1] Asim Abdulkhaleq and Stefan Wagner. 2014. A Software Safety Verification Method Based on System-Theoretic Process Analysis. In *Computer Safety, Reliability, and Security*. Springer International Publishing, Cham, 401–412. doi:10.1007/978-3-319-10557-4_44
- [2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification – The KeY Book*. Lecture Notes in Computer Science, Vol. 10001. Springer International Publishing, Cham. doi:10.1007/978-3-319-49812-6
- [3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. 2007. Cost Analysis of Java Bytecode. In *Programming Languages and Systems*. Springer, Berlin, Heidelberg, 157–172. doi:10.1007/978-3-540-71316-6_12
- [4] Ralph-Johan Back and Joakim Wright. 1998. *Refinement Calculus*. Springer. doi:10.1007/978-1-4612-1674-2
- [5] Thomas Ball and James R. Larus. 1994. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.* 16, 4 (July 1994), 1319–1360. doi:10.1145/183432.183527
- [6] Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo. 2016. Energy-Aware Scheduling for Real-Time Systems: A Survey. *ACM Trans. Embed. Comput. Syst.* 15, 1 (Jan. 2016), 7:1–7:34. doi:10.1145/2808231
- [7] Jürgen Becker et al. 2021. XANDAR: X-by-Construction Design framework for Engineering Autonomous & Distributed Real-time Embedded Software Systems. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. 382–383. doi:10.1109/FPL53798.2021.00075
- [8] Tabea Bordis. 2025. *Scaling Correctness-by-Construction*. Ph. D. Dissertation. Karlsruhe Institute of Technology, Karlsruhe. Forthcoming.
- [9] Tabea Bordis, Loek Cleophas, Alexander Kittelmann, Tobias Runge, Ina Schaefer, and Bruce W. Watson. 2022. 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, 80–104. doi:10.1007/978-3-031-08166-8_5
- [10] Tabea Bordis, Maximilian Kodetzki, and Ina Schaefer. 2024. From Concept to Reality: Leveraging Correctness-by-Construction for Better Algorithm Design. *Computer* 57, 7 (July 2024), 113–119. doi:10.1109/MC.2024.3390948
- [11] Tabea Bordis, Tobias Runge, Alexander Knüppel, Thomas Thüm, and Ina Schaefer. 2020. Variational Correctness-by-Construction. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. ACM, Magdeburg Germany, 1–9. doi:10.1145/3377024.3377038
- [12] Richard Bubel, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, Olaf Owe, Ina Schaefer, and Ingrid Chieh Yu. 2016. Proof Repositories for Compositional Verification of Evolving Software Systems. In *Transactions on Foundations for Mastering Change I*. Springer International Publishing, Cham, 130–156. doi:10.1007/978-3-319-46508-1_8
- [13] Radu Calinescu, Carlo Ghezzi, Kenneth Johnson, Mauro Pezzé, Yasmin Rafiq, and Giordano Tamburrelli. 2016. Formal Verification With Confidence Intervals to Establish Quality of Service Properties of Software Systems. *IEEE Transactions on Reliability* 65, 1 (March 2016), 107–125. doi:10.1109/TR.2015.2452931
- [14] Eugenio Capra, Chiara Francalanci, and Sandra A. Slaughter. 2012. Measuring Application Software Energy Efficiency. *IT Professional* 14, 2 (March 2012), 54–61. doi:10.1109/MITP.2012.39
- [15] S. Chaki, E. Clarke, D. Giannakopoulou, and C. S. Pasareanu. 2004. Abstraction and Assume-Guarantee Reasoning for Automated Software Verification. https://ntrs.nasa.gov/citations/20050185529
- [16] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. PÄsÄreanu. 2003. Learning Assumptions for Compositional Verification. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, 331–346. doi:10.1007/3-540-36577-X_24
- [17] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. 2012. Energy types. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 831–850. doi:10.1145/2384616.2384676
- [18] D.C. Craig and W.M. Zuberek. 2006. Compatibility of Software Components - Modeling and Verification. In *2006 International Conference on Dependability of Computer Systems*. 11–18. doi:10.1109/DEPCOS-RELCOMEX.2006.13
- [19] Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457. doi:10.1145/360933.360975
- [20] Edsger Wybe Dijkstra. 1976. *A discipline of programming*. Englewood Cliffs, N.J. : Prentice-Hall.
- [21] Elizabeth D. Dolan and Jorge J. Moré. 2002. Benchmarking optimization software with performance profiles. *Mathematical Programming* 91, 2 (Jan. 2002), 201–213. doi:10.1007/s101070100263
- [22] Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. 2014. Probabilistic model checking for energy analysis in software product lines. In *Proceedings of the 13th international conference on Modularity (MODULARITY '14)*. Association for Computing Machinery, New York, NY, USA, 169–180. doi:10.1145/2577080.2577095
- [23] Tobias Dörr et al. 2024. XANDAR: An X-by-Construction Framework for Safety, Security, and Real-Time Behavior of Embedded Software Systems. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6. doi:10.23919/DATE58400.2024.10546852
- [24] Michael Ferdman et al. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *SIGPLAN Not.* 47, 4 (2012), 37–48. doi:10.1145/2248487.2150982
- [25] Alcides Fonseca and Guilherme Espada. 2022. Type Systems in Resource-Aware Programming: Opportunities and Challenges. doi:10.48550/arXiv.2205.15211
- [26] Stefano Gallotti, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. 2008. Quality Prediction of Service Compositions through Probabilistic Model Checking. In *Quality of Software Architectures. Models and Architectures*. Springer, Berlin, Heidelberg, 119–134. doi:10.1007/978-3-540-87879-7_8
- [27] D. Giannakopoulou, C.S. Pasareanu, and H. Barringer. 2002. Assumption generation for software component verification. In *Proceedings 17th IEEE International Conference on Automated Software Engineering,*. 3–12. doi:10.1109/ASE.2002.1114984
- [28] Daniel H. Greene and Donald E. Knuth. 1990. *Mathematics for the Analysis of Algorithms*. Birkhäuser, Boston, MA. doi:10.1007/978-0-8176-4729-2
- [29] David Gries. 1981. *The Science of Programming*. Springer New York, New York, NY. doi:10.1007/978-1-4612-5983-1
- [30] Juan Pedro Gutiérrez Hermosillo Muriedas et al. 2023. perun: Benchmarking Energy Consumption of High-Performance Computing Applications. In *Euro-Par 2023: Parallel Processing*. Springer Nature Switzerland, Cham, 17–31. doi:10.1007/978-3-031-39698-4_2
- [31] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance debugging in the large via mining millions of stack traces. In *2012 34th International Conference on Software Engineering (ICSE)*. 145–155. doi:10.1109/ICSE.2012.6227198
- [32] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. 2013. Estimating mobile application energy consumption using program analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 92–101. doi:10.1109/ICSE.2013.6606555
- [33] Richard Hawkins, Ibrahim Habli, and Tim Kelly. 2013. The principles of software safety assurance. In *31st International System Safety Conference*. The International System Safety Society Boston, Massachusetts USA, 12–16.
- [34] Ian J. Hayes. 2002. The Real-Time Refinement Calculus: A Foundation for Machine-Independent Real-Time Programming. In *Application*

and Theory of Petri Nets 2002. Springer, Berlin, Heidelberg, 44–58. doi:10.1007/3-540-48068-4_3

[35] Ilja Heitlager, Tobias Kuipers, and Joost Visser. 2007. A Practical Model for Measuring Maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*. 30–39. doi:10.1109/QUATIC.2007.8

[36] J.L. Henning. 2000. SPEC CPU2000: measuring CPU performance in the New Millennium. *Computer* 33, 7 (July 2000), 28–35. doi:10.1109/2.869367

[37] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. 2014. GreenMiner: a hardware based mining software repositories software energy consumption framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 12–21. doi:10.1145/2597073.2597097

[38] C.E. Hrischuk, C. Murray Woodside, and J.A. Rolia. 1999. Trace-based load characterization for generating performance software models. *IEEE Transactions on Software Engineering* 25, 1 (Jan. 1999), 122–135. doi:10.1109/32.748921

[39] Eunkyoung Jee, Shaohui Wang, Jeong Ki Kim, Jaewoo Lee, Oleg Sokolsky, and Insup Lee. 2010. A Safety-Assured Development Approach for Real-Time Software. In *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*. 133–142. doi:10.1109/RTCSA.2010.42

[40] Z. Jelinski and P. Moranda. 1972. Software Reliability Research. In *Statistical Computer Performance Evaluation*. Academic Press, 465–484. doi:10.1016/B978-0-12-266950-7.50028-1

[41] Timo Johann, Markus Dick, Stefan Naumann, and Eva Kern. 2012. How to measure energy-efficiency of software: Metrics and measurement results. In *2012 First International Workshop on Green and Sustainable Software (GREENS)*. 51–54. doi:10.1109/GREENS.2012.6224256

[42] Eva Kern, Lorenz M. Hilty, Achim Guldner, Yuliyan V. Maksimov, Andreas Filler, Jens Gröger, and Stefan Naumann. 2018. Sustainable software products—Towards assessment criteria for resource and energy efficiency. *Future Generation Computer Systems* 86 (Sept. 2018), 199–210. doi:10.1016/j.future.2018.02.044

[43] Rody Kersten, Paolo Parisen Toldin, Bernard van Gastel, and Marko van Eekelen. 2014. A Hoare Logic for Energy Consumption Analysis. In *Foundational and Practical Aspects of Resource Analysis*. Springer International Publishing, Cham, 93–109. doi:10.1007/978-3-319-12466-7_6

[44] Gregor Kiczales and Mira Mezini. 2005. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering - ICSE '05*. ACM Press, St. Louis, MO, USA, 49. doi:10.1145/1062455.1062482

[45] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 253–268. doi:10.1145/3314221.3314602

[46] Alexander Knüppel, Tobias Runge, and Ina Schaefer. 2020. Scaling Correctness-by-Construction. In *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles: 9th International Symposium on Leveraging Applications of Formal Methods, Part I*. Springer-Verlag, Berlin, Heidelberg, 187–207. doi:10.1007/978-3-030-61362-4_10

[47] Maximilian Kodetzki, Tabea Bordis, Michael Kirsten, and Ina Schaefer. 2025. Towards AI-Assisted Correctness-by-Construction Software Development. In *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering Methodologies*. Springer Nature Switzerland, Cham, 222–241. doi:10.1007/978-3-031-75387-9_14

[48] Derrick G. Kourie and Bruce W. Watson. 2012. *The Correctness-by-Construction Approach to Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg. doi:10.1007/978-3-642-27919-5

[49] G. Latif-Shabgahi, J.M. Bass, and S. Bennett. 2004. A taxonomy for software voting algorithms used in safety-critical systems. *IEEE Transactions on Reliability* 53, 3 (Sept. 2004), 319–328. doi:10.1109/TR.2004.832819

[50] N.G. Leveson and P.R. Harvey. 1983. Analyzing Software Safety. *IEEE Transactions on Software Engineering* SE-9, 5 (Sept. 1983), 569–579. doi:10.1109/TSE.1983.235116

[51] J. N. Lyness. 1978. *Performance profiles and software evaluation*. Technical Report CONF-781233-1. Argonne National Lab., IL (USA). https://www.osti.gov/biblio/6411646

[52] Pedro López-García, Luthfi Darmawan, Maximiliano Klemen, Umer Liqat, Francisco Bueno, and Manuel V. Hermenegildo. 2018. Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption. *CoRR* abs/1803.04451 (2018). http://arxiv.org/abs/1803.04451

[53] Main Commission Aircraft Accident Investigation Warsaw. 1994. A320-211 Warsaw Accident Report.

[54] Javier Mancebo, Félix García, and Coral Calero. 2021. A process for analysing the energy efficiency of software. *Information and Software Technology* 134 (June 2021), 106560. doi:10.1016/j.infsof.2021.106560

[55] Leonard Masing et al. 2022. XANDAR: Exploiting the X-by-Construction Paradigm in Model-based Development of Safety-critical Systems. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), Antwerp, Belgium, 14-23 March 2022*. 814. doi:10.23919/DATE54114.2022.9774534

[56] Annabelle McIver and Carroll Morgan. 2006. Developing and Reasoning About Probabilistic Programs in pGCL. In *Refinement Techniques in Software Engineering: First Pernambuco Summer School on Software Engineering, PSSE 2004*. Springer, Berlin, Heidelberg, 123–155. doi:10.1007/11889229_4

[57] Peter David Miller. 2006. *Applied Asymptotic Analysis*. American Mathematical Soc.

[58] Carroll Morgan. 1990. *Programming from specifications*. Prentice-Hall, Inc., USA.

[59] San Murugesan. 2008. Harnessing Green IT: Principles and Practices. *IT Professional* 10, 1 (Jan. 2008), 24–33. doi:10.1109/MITP.2008.10

[60] John D. Musa. 1979. Software reliability measurement. *Journal of Systems and Software* 1 (Jan. 1979), 223–241. doi:10.1016/0164-1212(79)90023-2

[61] Ann Marie Neufelder. 2018. *Ensuring Software Reliability*. CRC Press, Boca Raton. doi:10.1201/9781315217758

[62] Hanne Riis Nielson. 1987. A hoare-like proof system for analysing the computation time of programs. *Science of Computer Programming* 9, 2 (Oct. 1987), 107–136. doi:10.1016/0167-6423(87)90029-3

[63] B. Nuseibeh. 1997. Ariane 5: Who Dunnit? *IEEE Software* 14, 3 (May 1997), 15–16. doi:10.1109/MS.1997.589224

[64] Osinachi Deborah Segun-Falade et al. 2024. Developing crossplatform software applications to enhance compatibility across devices and systems. *Computer Science & IT Research Journal* 5, 8 (Aug. 2024), 2040–2061. doi:10.51594/csitrj.v5i8.1491

[65] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2021. Ranking programming languages by energy efficiency. *Science of Computer Programming* 205 (May 2021), 102609. doi:10.1016/j.scico.2021.102609

[66] Gustavo Pinto, Francisco Soares-Neto, and Fernando Castor. 2015. Refactoring for Energy Efficiency: A Reflection on the State of the Art. In *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*. 29–35. doi:10.1109/GREENS.2015.12

[67] Meikel Poess and Chris Floyd. 2000. New TPC benchmarks for decision support and web commerce. *SIGMOD Rec.* 29, 4 (2000), 64–71. doi:10.1145/369275.369291

[68] Vijay Janapa Reddi et al. 2020. MLPerf Inference Benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 446–459. doi:10.1109/ISCA45697.2020.00045

[69] Swapnoneel Roy, Atri Rudra, and Akshat Verma. 2013. An energy complexity model for algorithms. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science (ITCS '13)*. Association for Computing Machinery, New York, NY, USA, 283–304. doi:10.1145/2422436.2422470

[70] Tobias Runge, Alexander Kittelmann, Marco Servetto, Alex Potanin, and Ina Schaefer. 2022. Information Flow Control-by-Construction for an Object-Oriented Language. In *Software Engineering and Formal Methods*. Springer International Publishing, Cham, 209–226. doi:10.1007/978-3-031-17108-6_13

[71] Tobias Runge, Alexander Knüppel, Thomas Thüm, and Ina Schaefer. 2020. Lattice-Based Information Flow Control-by-Construction for Security-by-Design. In *Proceedings of the 8th International Conference on Formal Methods in Software Engineering (FormaliSE '20)*. Association for Computing Machinery, New York, NY, USA, 44–54. doi:10.1145/3372020.3391565

[72] Tobias Runge, Ina Schaefer, Loek Cleophas, Thomas Thüm, Derrick Kourie, and Bruce W. Watson. 2019. Tool Support for Correctness-by-Construction. In *Fundamental Approaches to Software Engineering*. Vol. 11424. Springer International Publishing, Cham, 25–42. doi:10.1007/978-3-030-16722-6_2

[73] Tobias Runge, Thomas Thüm, Loek Cleophas, Ina Schaefer, and Bruce W. Watson. 2020. Comparing Correctness-by-Construction with Post-Hoc Verification—A Qualitative User Study. In *Formal Methods. FM 2019 International Workshops*. Vol. 12233. Springer International Publishing, 388–405. doi:10.1007/978-3-030-54997-8_25

[74] Rasmus C. Rønneberg, Tabea Bordis, Christopher Gerking, Asmae Heydari Tabar, and Ina Schaefer. 2025. Scaling Information Flow Control By-Construction to Component-Based Software Architectures. In *Formal Techniques for Distributed Objects, Components, and Systems*. Vol. 15732. Springer Nature Switzerland, Cham, 55–74. doi:10.1007/978-3-031-95497-9_4

[75] Ina Schaefer, Dilian Gurov, and Siavash Soleimanifard. 2012. Compositional Algorithmic Verification of Software Product Lines. In *Formal Methods for Components and Objects*. Springer, Berlin, Heidelberg, 184–203. doi:10.1007/978-3-642-25271-6_10

[76] Ina Schaefer, Tobias Runge, Alexander Knüppel, Loek Cleophas, Derrick Kourie, and Bruce W. Watson. 2018. Towards Confidentiality-by-Construction. In *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*. Springer International Publishing, Cham, 502–515. doi:10.1007/978-3-030-03418-4_30

[77] Natasha Sharygina, James C. Browne, and Robert P. Kurshan. 2001. A Formal Object-Oriented Analysis for Software Reliability: Design for Verification. In *Fundamental Approaches to Software Engineering*. Springer, Berlin, Heidelberg, 318–332. doi:10.1007/3-540-45314-8_23

[78] Natasha Sharygina and Doron Peled. 2001. A Combined Testing and Verification Approach for Software Reliability. In *FME 2001: Formal Methods for Increasing Software Productivity*. Springer, Berlin, Heidelberg, 611–628. doi:10.1007/3-540-45251-6_35

[79] Maurice H. ter Beek, Loek Cleophas, Axel Legay, Ina Schaefer, and Bruce W. Watson. 2020. X-by-Construction. In *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*. Springer International Publishing, Cham, 211–215. doi:10.1007/978-3-030-61362-4_11

[80] Maurice H. ter Beek, Loek Cleophas, Ina Schaefer, and Bruce W. Watson. 2018. X-by-Construction. In *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*. Springer International Publishing, Cham, 359–364. doi:10.1007/978-3-030-03418-4_21

[81] V. Tiwari, S. Malik, and A. Wolfe. 1994. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 2, 4 (Dec. 1994), 437–445. doi:10.1109/92.335012

[82] Bernard van Gastel, Rody Kersten, and Marko van Eekelen. 2016. Using Dependent Types to Define Energy Augmented Semantics of Programs. In *Foundational and Practical Aspects of Resource Analysis*. Springer International Publishing, Cham, 20–39. doi:10.1007/978-3-319-46559-3_2

[83] Shinan Wang, Hui Chen, and Weisong Shi. 2011. SPAN: A software power analyzer for multicore computer systems. *Sustainable Computing: Informatics and Systems* 1, 1 (March 2011), 23–34. doi:10.1016/j.suscom.2010.10.002

[84] Bruce William Watson. 1995. *Taxonomies and toolkits of regular language algorithms*. Eindhoven University of Technology, Dept. of Mathematics and Computing Science, Eindhoven.

[85] Kurt D. Welker, Paul W. Oman, and Gerald G. Atkinson. 1997. Development and Application of an Automated Source Code Maintainability Index. *Journal of Software Maintenance: Research and Practice* 9, 3 (1997), 127–159. doi:10.1002/(SICI)1096-908X(199705)9:3<127::AID-SMR149>3.0.CO;2-S

[86] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. 2000. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Proceedings of the 37th Annual Design Automation Conference (DAC '00)*. Association for Computing Machinery, New York, NY, USA, 340–345. doi:10.1145/337292.337436

[87] Qi Zhao, Hailong Yang, Zhongzhi Luan, and Depei Qian. 2013. POIGEM: A Programming-Oriented Instruction Level GPU Energy Model for CUDA Program. In *Algorithms and Architectures for Parallel Processing*. Springer International Publishing, Cham, 129–142. doi:10.1007/978-3-319-03859-9_10

Received 2025-04-24; accepted 2025-08-11