Formal Verification of Aerospace Cyber-Physical System Software*

Yuriy Manzhos[1,†] and Yevheniia Sokolova[1,*,†]

[1] National Aerospace University "Kharkiv Aviation Institute", Vadyma Manka St, 17, Kharkiv, 61070 Kharkiv, Ukraine

Abstract

The growing complexity of aerospace cyber-physical systems demands rigorous methods to ensure software correctness, reliability, and compliance with safety standards. Traditional verification techniques often fail to detect dimensional inconsistencies that can lead to critical failures. This paper presents a formal verification approach based on dimensional analysis, specifically tailored for aerospace software. The method employs mathematical models derived from the statistical characteristics of C/C++ source code to identify dimensional defects in computations, data flows, and control algorithms. The proposed approach provides several benefits: early compile-time detection of defects, reduced testing effort and duration, cost savings through the elimination of latent defects, and improved software reliability, robustness, and performance. By integrating dimensional analysis with formal specification and verification frameworks, the method enables early detection of inconsistencies and mitigates defect propagation across system components. Case studies show that the method detects up to 90% (with an average of 66%) of incorrect uses of software variables and operations at both compile time and run time. This detection rate depends on the distribution of operations and dimensions in the C++ source files. The results demonstrate the method's effectiveness in uncovering errors overlooked by conventional testing. Overall, the proposed technique serves as a complementary tool for aerospace software verification, enhancing quality assurance and supporting certification processes.

Keywords

aerospace cyber-physical systems, formal verification, physical dimension, software defect model

1. Introduction

Aerospace cyber-physical systems (CPS) are highly integrated environments where physical processes such as flight dynamics are tightly coupled with computational algorithms, sensors, actuators, and control systems [1]. These systems are increasingly central to modern aerospace engineering, improving efficiency, safety, performance, and real-time decision-making. Applications of aerospace CPS span autonomous aircraft, spacecraft and satellites, commercial and military aviation, and predictive maintenance. In these contexts, CPS enable autonomy, advanced flight control, digital twins, sensor fusion, and adaptive diagnostics [2].

Despite their advantages, aerospace CPS face several critical challenges. Real-time constraints demand processing of large volumes of sensor data with strict latency requirements, especially for navigation, flight control, and emergency response. Safety and reliability are paramount, as these systems are safety-critical and must be fault-tolerant and resilient to avoid catastrophic failures. Integration and testing also remain complex, requiring approaches such as model-based design (MBD) to ensure subsystem interoperability [3]. Finally, increasing reliance on digital communication exposes aerospace CPS to cybersecurity threats, necessitating robust protection mechanisms[4].

Addressing these challenges necessitates advanced verification methods. Classical approaches to software verification, including Hoare's seminal work on proving compiler correctness [5], demonstrate that formal verification can provide guarantees about program behavior beyond

* ProfIT AI'25: 5th International Workshop of IT-professionals on Artificial Intelligence, October 15–17, 2025, Liverpool, UK

* Corresponding author.

† These authors contributed equally.

y.manzhos@khai.edu (Y. Manzhos); y.sokolova@khai.edu (Y. Sokolova)

0000-0002-4910-7285 (Y. Manzhos); 0000-0002-1497-4987 (Y. Sokolova)

syntactic correctness [6]. Building on these principles, this paper explores the use of formal verification [7] based on dimensional analysis (DA) [8] as a novel approach to enhance the reliability and safety of aerospace CPS software. Prior work by the authors [9] [10] has demonstrated the feasibility of applying DA to C/C++ code for detecting latent errors and ensuring physical consistency.

This paper explores the use of formal verification based on DA as a novel approach to improve the reliability and safety of aerospace CPS software.

2. The Formal Verification Method of CPS software

The proposed approach leverages natural software invariants, defined as the physical dimensions of variables corresponding to real-world quantities. By integrating these invariants into program specifications, program expressions can be reformulated into a set of lemmas requiring formal proof. This process enables verification of both dimensional homogeneity and program conciseness.

As noted by Martínez-Rojas et al. [11], DA is a well-established methodology in physics and engineering, used to identify or validate relationships among physical quantities based on their dimensions. Within the International System of Units (SI) [12], each physical quantity is expressed as a combination of seven base dimensions: length (meter, m), time (second, s), amount of substance (mole, mol), electric current (ampere, A), temperature (kelvin, K), luminous intensity (candela, cd), and mass (kilogram, kg) [13]. Derived units are defined as products of powers of base units, and when the numerical factor of such a product equals one, they are classified as coherent derived units. Together, SI base and coherent derived units form a coherent system, where equations involving numerical values mirror the structure of the underlying physical relationships. This property ensures consistency and accuracy in computations involving physical quantities, making DA a reliable foundation for software verification.

Some coherent derived units in the SI are assigned specific names and, together with the seven base units, form the foundation of the SI system. All other units are expressed as combinations of these. The central principle of DA is that physical laws must remain valid regardless of the units used. According to the rule of dimensional homogeneity, every physically meaningful equation must balance dimensions on both sides. This principle underpins the use of DA across physics and engineering.

Conventional software analysis tools primarily check syntax and semantics, but not the physical correctness of code. When program code is treated as a set of expressions involving variables, constants, and operations, DA can be integrated into the compilation or verification process. This enables the detection of mismatches in variable usage, inconsistent unit conversions, and incorrect dimensional operations directly at the software level. By embedding dimensional checks into program specifications, aerospace and safety-critical applications can benefit from early error detection, reduced defect propagation, and improved reliability of C/C++ implementations [14].

The correctness of program expressions can be evaluated by analyzing the dimensionality of their values. When expressions preserve dimensional homogeneity, they are likely to represent physically meaningful relationships. Conversely, violations of homogeneity signal incorrect use of program variables or operations. Unlike conventional checks, DA can be applied not only to simple expressions but also to procedure and function calls, enabling a broader validation of software behavior.

DA thus provides a systematic way to ensure the physical correctness of software code. By embedding physical dimensions into program specifications, it becomes possible to verify that computations remain consistent with the physical laws governing the modeled system. In this view, software can be regarded as a model of a physical process, and DA serves as a validation tool to confirm the model's adherence to physical principles.

Incorporating DA into the software development and verification process enables the detection of errors caused by inconsistent or incorrect unit usage [15]. This contributes to the development of safer, more reliable, and physically accurate software—particularly in safety-critical domains such as aerospace cyber-physical systems.

A software system can be viewed as a hierarchy of interacting components. At the highest level, it is composed of subsystems; each subsystem consists of software units, and each unit is defined by a set of operators. Operators, in turn, are expressed as ordered sequences of statements or expressions. This hierarchical structure allows systematic verification of interactions and operations across different levels of the system.

To establish dimensional homogeneity within the system, verification must proceed step by step: the homogeneity of the overall system depends on the homogeneity of its subsystems; each subsystem's homogeneity depends on that of its software units; and unit-level homogeneity requires the homogeneity of individual statements or expressions. This layered approach provides a structured pathway for verifying dimensional consistency throughout the code.

Within this framework, we distinguish between two categories of operations. Multiplicative operations (*, /, etc.) generate new physical dimensions, while additive operations (+, −, =, <, ≤, >, ≥, !=, etc.) act as checkpoints, enforcing dimensional homogeneity. When program variables are associated with specific physical dimensions, this property can be treated as a software invariant. Each additive operation then serves as a basis for generating lemmas, which collectively support the formal verification of dimensional correctness across the system.

DA enables verification of the physical dimensions of program variables, allowing the detection of errors caused by inconsistent unit usage, incorrect dimensional relationships, or improper application of operations, variables, and procedures. Nonetheless, challenges arise when different quantities share the same dimensions. For example, moments of inertia and angular velocity both involve combinations of mass and length, yet represent fundamentally different physical concepts. Detecting defects in such cases requires careful analysis of expressions.

3. Software Defect Detection Models

Software defect detection models designed to identify errors, inconsistencies, and potential faults in software systems before deployment. These models aim to predict, locate, and prevent defects by analyzing code structure, execution patterns, or software behavior.

Proposed defect detection models incorporate probabilistic methods to improve predictive accuracy, leveraging defect data. In safety-critical domains, such as aerospace and automotive systems, defect detection models are essential for ensuring reliability and compliance with dimensional homogeneity.

3.1. General Software Defect Detection Model

To simplify the analysis, we assume that each software statement may contain at most one defect, occurring with a probability of P_{def}. The model begins with the initial state labeled "Software", which branches into two possible outcomes: "Software has a defect" with probability P_{def}, and "Software does not have a defect" with probability $1-P_{\text{def}}$.

Decision trees provide a structured way to visualize sequences of decisions or events along with their probabilities and outcomes. In the context of software defect detection, they can be enhanced by incorporating DA. By assigning physical dimensions as invariants to program variables, DA allows the detection of defects arising from inconsistent or incorrect use of units and operations. Each node in the decision tree can represent not only the presence or absence of a syntactic or semantic defect but also violations of dimensional homogeneity.

This combined approach enables early identification of errors that might remain undetected by conventional testing. For example, a branch could represent a statement where a variable's

dimensional type conflicts with an operation, triggering a defect detection alert. Figure 1 illustrates a decision tree model for software defect detection enhanced with dimensional analysis, demonstrating how this methodology supports systematic verification of both traditional and physics-based software correctness.

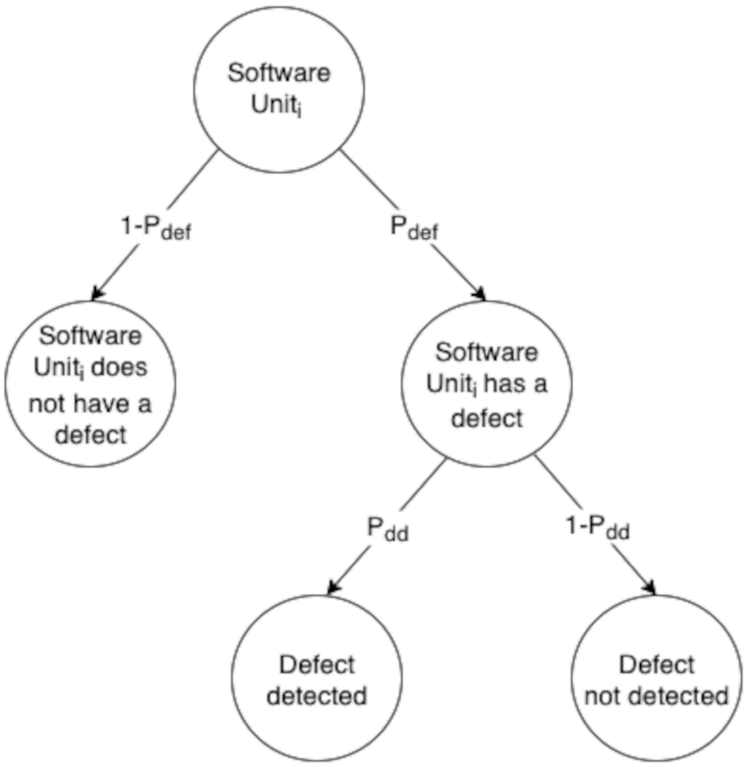

Figure 1: General software defect detection model.

Software has N software units. In the state 'Software Unit$_i$ has a defect', our focus shifts to detecting the defect. The model branches out into two possible outcomes: 'Defect detected' and 'Defect not detected', with probabilities of P_{DD} and $1-P_{DD}$, respectively. Let us define that software units has equal P_{def} Here P_{def} represents a probability of a software defect in the code. The conditional probability of software unit$_i$ defect detection defined as:

$$\eta_i = \frac{P_{\text{def}}}{P_{\text{def}} P_{DDi} + P_{\text{def}}(1-P_{DDi})} = P_{DDi}. \tag{1}$$

The total conditional probability of software defect detection $\eta = \sum_{i=1}^{N} P_{DDi}\omega_i = \sum_{i=1}^{N} \eta_i \omega_i$, where ω_i is a weight of software unit$_i$, e.g. $\omega_i = \frac{S_i}{\sum_{i=1}^{N} S_i}$, where S_i is a size of software unit$_i$ and $\sum_{i=1}^{N} \omega_i = 1$.

We can extend the defect detection model to account for two types of defects - variable defects and operation defects—while still assuming that at most one defect occurs per software statement (see Figure 2).

A variable defect arises when a program variable is used incorrectly, such as referencing the wrong variable or using one with an incompatible physical dimension or orientation. An operation defect occurs when an operator or function is applied incorrectly, for example, using the wrong arithmetic or logical operator within a software unit$_i$.

By distinguishing these defect types, the model provides a more detailed representation of potential errors in software statements, enabling targeted detection and analysis. Despite the increased complexity, the assumption of a single defect per statement simplifies the probabilistic modeling, allowing the systematic application of decision tree methods and, when integrated with DA, facilitates the detection of both conventional coding errors and violations of physical correctness.

In this more complex model, the initial event state is 'Software unit$_i$.' At the branching point, the model expands into two possible outcomes: 'Variable' and 'Operation,' with probabilities of P_{vari} and $1-P_{vari}$, respectively which defined for every software unit$_i$. The 'Variable' state has two potential outcomes at the next level: 'Correct use of variable' and 'Incorrect use of variable', with probabilities of $1-P_{\text{def}}$ and P_{def}, respectively. The 'Incorrect use of variable' state has two possible outcomes at the next level: 'Variable defect detected' and 'Variable defect not detected', with probabilities of P_{VDI} and $1-P_{VDI}$, respectively. Here, P_{VDI} represents the probability of detecting a variable defect in the unit$_i$.

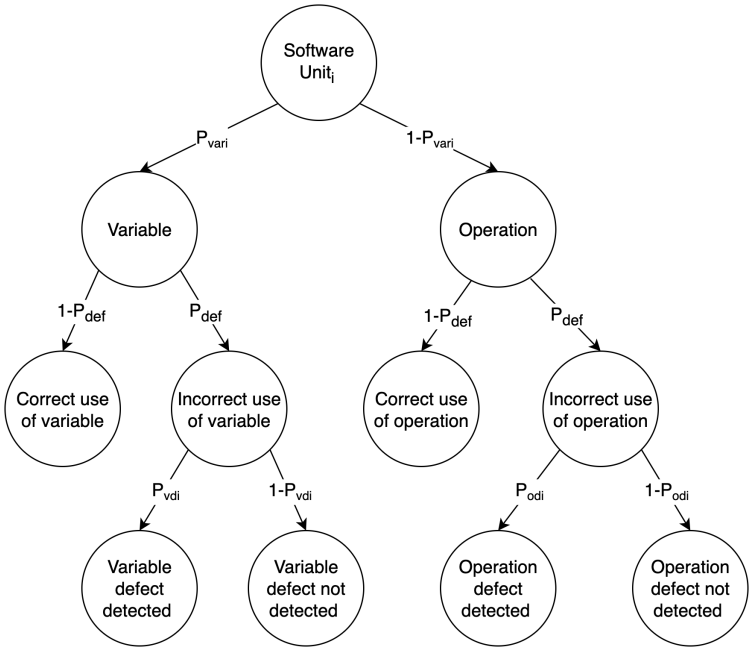

Figure 2: Complex software defect detection model.

In addition to the 'Variable' state, the model also has an 'Operation' state, which has two possible outcomes at the next level: 'Correct use of operation' and 'Incorrect use of operation', with probabilities of $1-P_{\text{def}}$ and P_{def}, respectively.

The 'Incorrect use of operation' state has two possible outcomes: 'Operation defect detected' and 'Operation defect not detected', with probabilities of P_{ODi} and $1-P_{ODi}$, respectively. Here, P_{ODi} represents the probability of detecting an operation defect in the source code of software unit$_i$. The conditional probability of a software defect in software unit$_i$ can be defined as follows:

$$\eta_i = \frac{P_{VDD_I} + P_{ODD_i}}{P_{VDD_i} + P_{VDND_i} + P_{ODD_i} + P_{ODND_i}},$$

here $P_{VDD_i} = P_{vari} P_{\text{def}} P_{VDi}$, $P_{ODD_i} = (1-P_{vari}) P_{\text{def}} P_{ODi}$,

$P_{VDND_i} = P_{vari} P_{\text{def}} (1-P_{VDi})$, $P_{ODND_i} = (1-P_{vari}) P_{\text{def}} (1-P_{ODi})$

and we have:

$$\eta_i = P_{vari} P_{VDi} + (1-P_{vari}) P_{ODi} \tag{2}$$

As per Expression (2), for software unit$_i$ the conditional probability of software defect detection depends on the probability of the software variables used in the source code of software unit and the conditional probabilities of detecting defects (defects of operations and defects of variables).

The total conditional probability of a software defect in software which includes N unit can be defined as follows: $\eta=\sum_{i=1}^{N}\eta_i\omega_i=\sum_{i=1}^{N}\left(P_{vari}P_{VDi}+\left(1-P_{vari}\right)P_{ODi}\right)\omega_i$, where ω_i is a weight of software uniti $\omega_i=\dfrac{N_{vi}+N_{Oi}}{\sum_{k=1}^{N}\left(N_{vk}+N_{Ok}\right)}$,

where N_{vi} - number of variable used in software unit$_i$, N_{Oi} - - number of operation used in software unit$_i$, N - total number of software units.

We can determine the value of P_{vari}, N_{vi}, N_{Oi} by analyzing the software code statically, i.e., without executing the code. However, to determine the values of P_{VDi} and P_{ODi}, we would need to build additional software defect detection models.

3.2. Software Defect Detection Model for Incorrect Variable Usage.

The proposed model allows the detection of incorrect variable usage (see Figure 3).

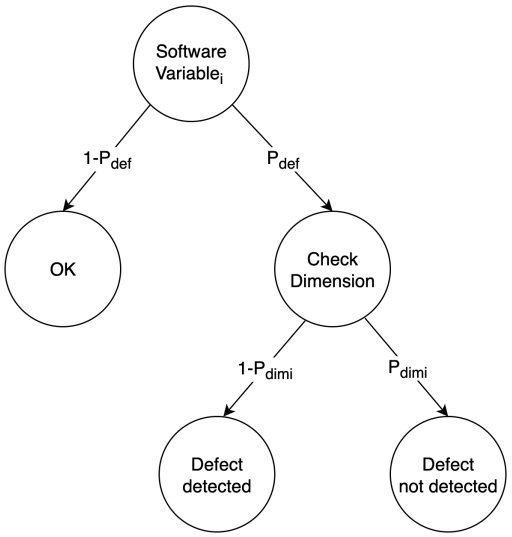

Figure 3: Model for Detecting Incorrect Variable Usage in Software.

This model has an initial state of 'Software Variable$_i$' and is applied to each software unit. The initial state has two transitions to states 'OK' and 'Check Dimension', with probabilities $1-P_{\text{def}}$ and P_{def}, respectively. In the state 'Check Dimension', we can evaluate the required physical dimension of the variable using dimensional analysis.

If the actual physical dimension is equal to the required physical dimension, we cannot detect the software defect. However, if the dimensions differ, a software defect can be identified. In this case, the probabilities are P_{dimi} and $1-P_{\text{dimi}}$, where P_{dimi} represents the probability that two randomly selected variables in software unit have the same physical dimension.

Let us define the conditional probability of defect detection of incorrect use of a program variable in the software unit$_i$ as follows:

$$P_{VDi}=\frac{P_{DD_i}}{P_{DD_i}+P_{DND_i}}.$$

Here $P_{DD_i} = P_{\text{def}}(1 - P_{dimi})$ and $P_{DND_i} = P_{\text{def}} P_{dimi}$,

$$P_{VDi} = 1 - P_{dimi} \tag{3}$$

Let us consider a set of distinct software variables $\{var_1, \ldots, var_{Nvi}\}$ and a set of diverse physical dimensions $\{dim_1, \ldots, dim_{Ndi}\}$, where N_{vi} represents the cardinality of set $\{var_i\}$ and N_{Di} represents the cardinality of set $\{dim_i\}$. To depict the relationship between these variables and dimensions, we can make use of an n_{ijk}-matrix (4) which defined for every software i-unit:

	dim_1	dim_2	dim_3	dim_4	dim_5	dim_6	dim_7	$\cdots$	dim_{Nd-1}	dim_{NDi}
var_1	n_{i11}	0	0	0	0	0	0	$\cdots$	0	0
var_2	n_{i21}	0	0	0	0	0	0	$\cdots$	0	0
var_3	0	n_{i31}	0	0	0	0	0	$\cdots$	0	0
var_4	0	0	n_{i43}	0	0	0	0	$\cdots$	0	0
var_5	0	0	0	n_{i54}	0	0	0	$\cdots$	0	0
var_6	0	0	0	n_{i64}	0	0	0	$\cdots$	0	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
var_{Nvi-}	0	0	0	0	0	0	0	$\cdots$	0	$n_i, N_{vi}-1, N_{Di}$
var_{Nvi}	0	0	0	0	0	0	0	$\cdots$	0	n_i, N_{vi}, N_{Di}

(4)

e.g. n_{i11} - the total number usage of var_1 which has physical dimension dim_1

n_{i21} - the total number usage of var_2 which has physical dimension dim_1 etc.

The equation for the total number of usages of all software variables with dimension j in software unit i can be written as follows:

$$N_{VUsageij} = \sum_{k=1}^{N_{vi}} n_{ikj}. \tag{5}$$

where n_{ikj} represents the total number of usages of k-variable which has a *j*-physical dimension which used in the i-unit; N_{vi} is the cardinality set of software variables in the i-unit. Equation (6) shows the total number of variable usages in the i-unit code:

$$N_{VARi} = \sum_{k=1}^{N_{vi}} \sum_{j=1}^{N_{Di}} n_{ikj}. \tag{6}$$

To define the probability of choosing *l*-variable and *m*-variable with the same dimensions, we can use the total number of usages of variables with the *j*-physical dimension and the total number of usages of all variables in the i-unit code:

$$D_{ilm} = \frac{n_{ilm}}{N_{VARi}} \frac{\left(\sum_{k=1}^{N_{Vi}} n_{ikl}\right) - n_{ilm}}{N_{VARi} - n_{ilm}} \tag{7}$$

According to (7), the probability of choosing two random variables that have the same physical dimension in i-unit is given by the following equation:

$$P_{dimi}=\sum_{l=1}^{N_{Vi}}\sum_{m=1}^{N_{Di}}\left(\frac{n_{ilm}}{N_{VARi}}\frac{\left(\sum_{k=1}^{N_{Vi}}n_{ikl}\right)-n_{ilm}}{N_{VARi}-n_{ij}}\right), \tag{8}$$

here n_{ikj} represents the number of usages for the k-variable with the *j*-physical dimension in the i-unit; N_{VARi} represents the total number of variable usages in the i-unit; N_{Di} represents the total number of different dimensions of variables in the i-unit; and N_{Vi} represents the total number of variables in the code.

The total conditional probability of software defect detection in a system consisting of N units can be defined as follows: $\eta_D=\sum_{i=1}^{N}P_{VDi}\omega_i$, where $P_{VDi}=1-P_{dimi}$, ω_i is a weight of software i-unit $\omega_i=\frac{N_{vi}}{\sum_{k=1}^{N}N_{vk}}$, where N_{vi} - number of variable usages in i-unit, N - total number of software units

$$\eta_D=\sum_{i=1}^{N}\left(1-P_{dimi}\right)\omega_i. \tag{9}$$

Expression (9) defines the total conditional probability of detecting a software defect in software comprising N units.

3.3. Model for Detecting Incorrect Usage of Operations and Variables.

Consider three subsets of C/C++ operations:

- Additive operations (A) include arithmetic, assignment, comparison, increment/decrement, member access, and concatenation operators.
- Multiplicative operations (M) include multiplication, division, modulo, and their compound assignment forms.
- Other operations (O) cover logical, bitwise, shift, scope, conditional, and compound assignment operators not included in A or M.

This classification supports dimensional analysis by distinguishing operators that generate new dimensions (multiplicative) from those that check or preserve dimensional homogeneity (additive), while other operators are treated separately for consistency checks. In addition, we are given three probabilities associated with the utilization of this operation in the source code, namely, P_A, P_M, and P_O. let us define the sum of these probabilities as the full group probability:

$$P_A+P_M+P_O=1. \tag{10}$$

Let us define P_A, P_M, and P_O as follows:

$$P_A=\frac{N_A}{N_A+N_M+N_O},\ P_M=\frac{N_M}{N_A+N_M+N_O},\ P_O=\frac{N_O}{N_A+N_M+N_O} \tag{11}$$

Here, N_A represents the total number of "additive" operations in a file, N_M represents the total number of "multiplicative" operations in a file, and N_O represents the total number of "other" operations in the file.

In this case, we can build a decision tree for the detection of incorrect use of operations based on dimensional analysis. The model allows us to define the conditional probability of operation defect detection (see Figure 4).

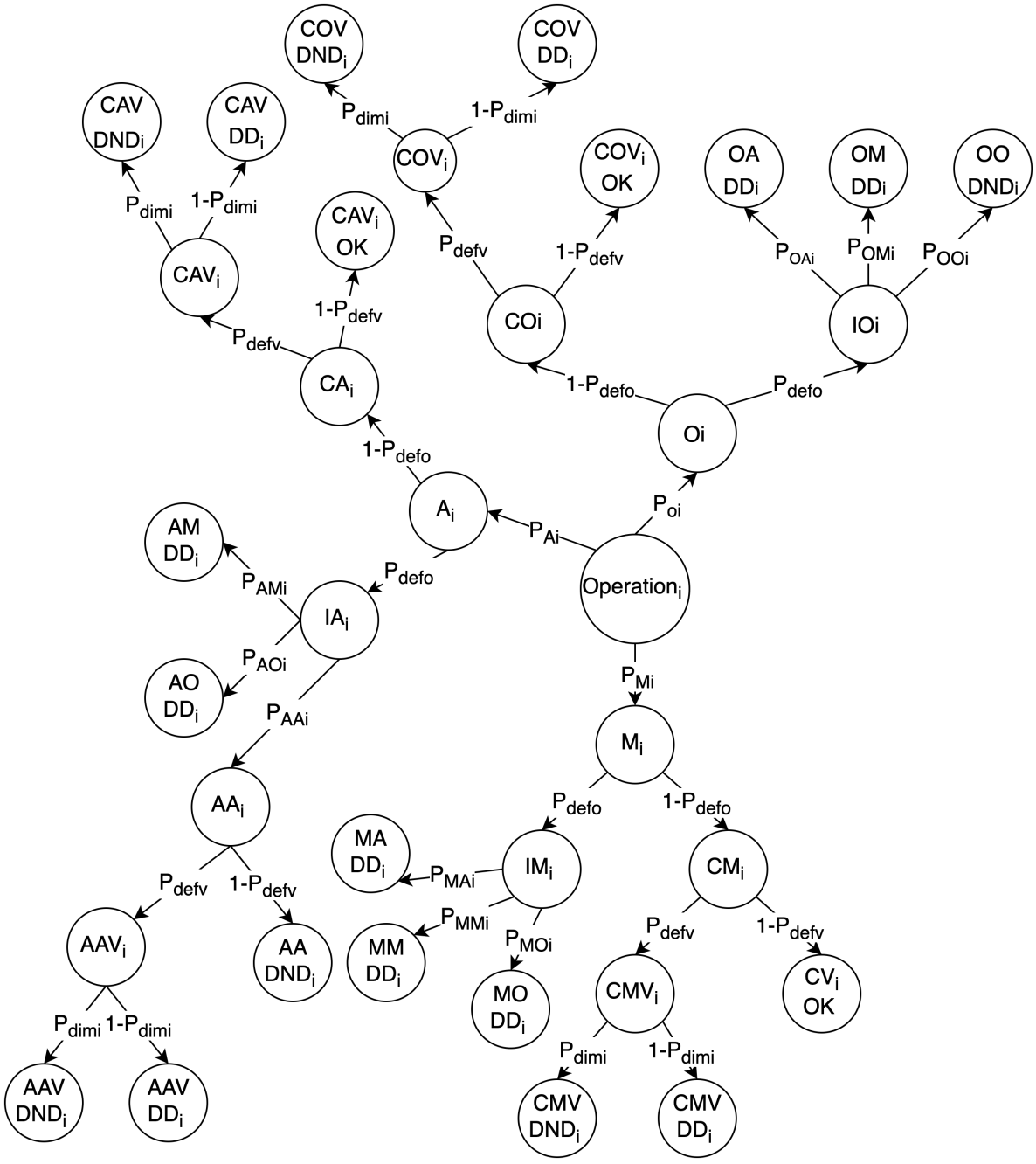

Figure 4: Model for Detecting Incorrect Usage of Operations and Variables.

According to Figure 4, the model begins with an initial state Operation, which branches into three additional states: A, M, and O, as defined in Expressions (10). The M state further splits into two states: IM_i (incorrect M operation, with probability P_{defO}) and CM_i (correct M operation, with probability $1-P_{defO}$). The CM_i state then divides into CMV_i (correct M operation with a Variable defect, probability P_{defv}) and CV_i (correct M operation with a correct variable, probability $1-P_{defv}$).

The CMV_i state expands further into two cases: $CMVDND_i$ (correct M operation with an incorrect variable of the same dimension, probability P_{dimi}, defect not detected) and $CMVDD_i$ (correct M operation with an incorrect variable of a different dimension, probability $1-P_{dimi}$, defect detected). Other nodes of the proposed model follow the same structural logic and are defined with analogous probability values.

Let us define the probability of a software defect as $P_{\text{def}}=\frac{N_{\text{def}}}{N_O+N_v}$. where $N_{\text{def}}=N_{defO}+N_{defV}$ is the total number of incorrect usages of operations and variables, and N_O+N_v is the total number usages of operations and variables. That is why we can define the probabilities of a software defect as: $P_{\text{defO}}=\frac{N_{defO}}{N_O}$, $P_{\text{defv}}=\frac{N_{defv}}{N_v}$.

Because most operations have two operands, we can define that $2N_O=N_v$ and:

$$P_{\text{def}}=\frac{N_{defO}+N_{defV}}{N_O+2N_O}=\frac{2}{3}P_{defv}+\frac{2}{3}\frac{N_{defO}}{N_v}.$$

Because $P_{\text{defO}}\ll 1$, $P_{\text{defv}}\ll 1$, $P_{\text{def}}\ll 1$, then $\frac{N_{defV}}{3N_O}\ll 1$ $\frac{2N_{defO}}{3N_v}\ll 1$ $P_{\text{def}}\approx\frac{P_{defO}}{3}\approx\frac{2}{3}P_{defv}$, $P_{\text{defO}}\approx 3P_{\text{def}}$, $P_{\text{defv}}\approx\frac{3}{2}P_{\text{def}}$.

According to Figure 4:

the conditional probability of software defect detection in the i-th unit can be defined as: $\eta_i=\frac{\sum P_{DD}}{\sum P_{DD}+\sum P_{DND}}$;

the conditional probability of absence of software defects in the i-th unit after performing formal verification can be defined as: $\varphi_i=\frac{\sum P_{DD}+\sum P_{OK}}{\sum P_{DD}+\sum P_{DND}+\sum P_{OK}}$.

According (10) we have: $\eta_i=1+\frac{\sum P_{DND}}{1-\sum P_{OK}}$ and $\varphi_i=1-\sum P_{DND}$.

According to Figure 4: $P_{OK}=P_{COVOK}+P_{CAVOK}+P_{CMVOK}$,
$P_{DND}=P_{COVDND}+P_{OODND}+P_{CAVDND}+P_{AADND}+P_{CMVDND}+P_{AAVDND}$

After simplification $P_{OK}=1-\frac{3}{2}P_{\text{def}}-3P_{\text{def}}+\frac{9}{2}P_{\text{def}}^2$,

$$\sum P_{DND}=\frac{3}{2}(1-3P_{\text{def}})P_{\text{def}}P_{dim}+3P_O^2P_{\text{def}}+3P_{\text{def}}P_A^2(1-\frac{3}{2}P_{\text{def}})+\frac{9}{2}P_A^2P_{\text{def}}^2P_{dim}.$$

Because $P_{\text{def}}\ll 1$ we can define:

$$\varphi_i\approx 1-3(\frac{P_{dim}}{2}+P_O^2+P_A^2)P_{\text{def}}, \quad (12)$$

$$\eta_i=1-\frac{3}{4}(\frac{P_{dim}}{2}+P_O^2+P_A^2). \quad (13)$$

The results of statistical modeling of conditional probabilities were obtained for $P_{\text{dim}}=0{,}0001\ldots0{,}1$ and $P_{\text{def}}=0{,}00001\ldots0{,}01$, $P_O\in[0\ldots1]$, $P_A\in[P_O\ldots1]$, and are shown in Figures 5, 6, 7.

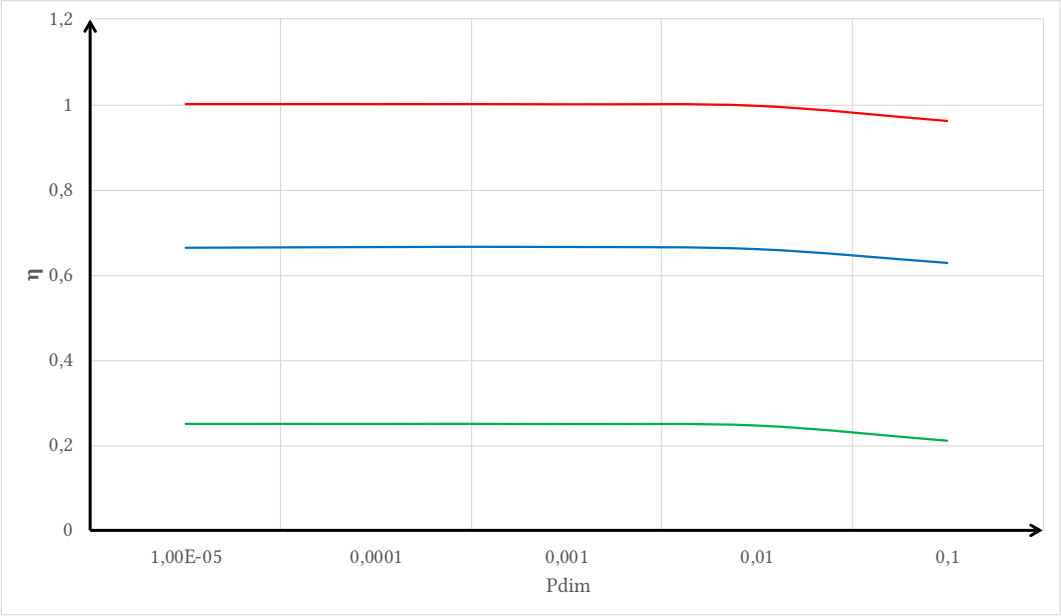

Figure 5: Conditional probability of software dimensional defects detecting as a function of P_{dim}. The green line represents the minimum value, the blue line – the expected value, and the red line – the maximum value of conditional probability.

The conditional probability of software defect detection is defined by the probabilities P_O and P_M. The average probability is approximately 0,66, with a minimum value of 0,26 and a maximum value of 1.0.

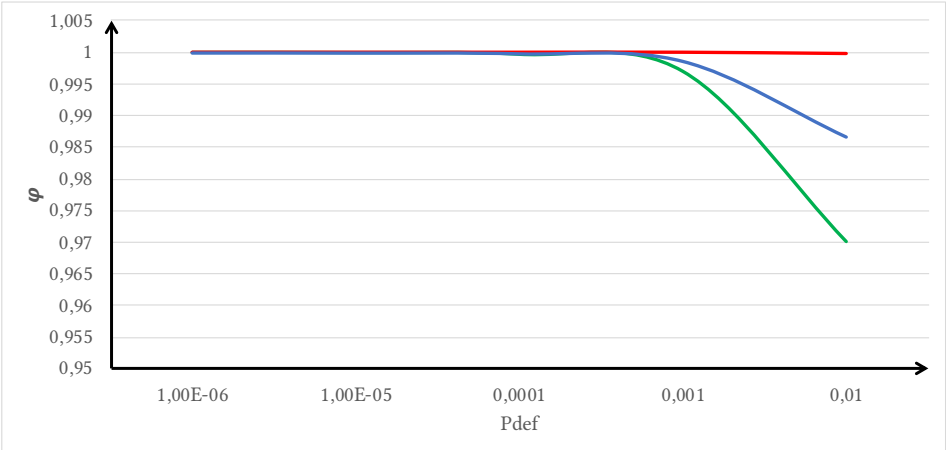

Figure 6: Conditional probability of software correctness after formal verification as a function of P_{def} for $P_{\text{dim}}=0{,}0001$. The green line represents the minimum value, the blue line – the expected value, and the red line – the maximum value of conditional probability.

The average probability of software correctness after formal verification for $P_{\text{dim}}=0{,}0001$ is approximately 1 when P_{def} varies within the range $0{,}000001\ldots0{,}001$.

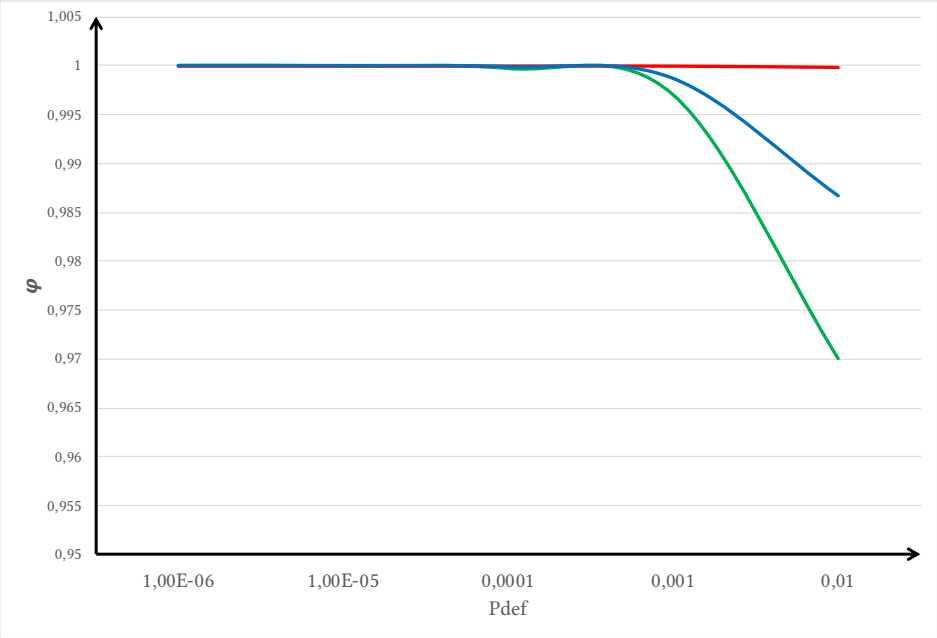

Figure 7: Conditional probability of software correctness after formal verification as a function of P_{def} for $P_{\text{dim}}=0{,}01$. The green line represents the minimum value, the blue line – the expected value, and the red line – the maximum value of conditional probability.

The average probability of software correctness after formal verification for $P_{\text{dim}}=0{,}01$ is approximately 1 when P_{def} varies within the range $0{,}000001\ldots0{,}001$.

The overall expected conditional probability of detecting dimensional defects in software.

$\eta=\sum_i \frac{\eta_i}{N_{Oi}}$, where N_{Oi} - total number of operations in the i-software unit.

The overall expected conditional probability of correctness after formal verification $\varphi=\sum_i \frac{\varphi_i}{N_{Oi}}$.

If all software units contain the same number of operations, the total probability is defined by expressions (12) and (13). For units of different sizes, the resulting probabilities decrease.

4. Conclusion

This paper has introduced a formal software verification method that leverages software invariants derived from dimensional analysis. According to [10], a formal type system defines software templates based on physical dimensions and fundamental numerical types. These templates enable the creation of dimensioned constants and variables and support operator overloading in C++. According to the C++, it is also possible to define numeric literals as constants of different dimension types.

The proposed approach offers several notable advantages. It enables early defect detection at compile-time, which helps reduce the likelihood of error propagation into later development stages. It also decreases testing overhead by supporting formal verification during both compile-time and run-time. As a result, development costs are lowered because early identification of defects minimizes the need for extensive debugging, maintenance, and post-release corrections. The method enhances software quality, demonstrating the capability to detect between 60% and 90% of latent defects related to incorrect use of physical dimensions in software variables and improper application of operations involving dimensioned variables. The proposed method allows checking the correct use of dimensioned arguments in C++ class methods. This detection rate depends on the distribution of operations and dimensions in the C++ source files [9]. It serves as a complementary verification technique that targets defect classes often missed by conventional

testing, based on the intrinsic characteristics of the software. Additionally, the method has the potential for continuous improvement, offering opportunities to further increase detection rates and overall reliability.

Together, these benefits underscore the method's effectiveness as a practical tool for software defect detection. Its high detection rate, along with reduced testing efforts and improved reliability, make it a compelling addition to the suite of formal verification techniques. The implementation of the proposed method using metaprogramming increases the reliability of software code, as the C/C++ compiler can perform formal verification at compile time [16].

However, the method does have certain limitations. Chief among them is the requirement for explicit knowledge of the physical dimensions and orientations of source variables at compile-time. Despite this constraint, the method enhances programmer productivity by automating the detection of dimensional and orientational inconsistencies. It also enables comprehensive dimensional correctness checks across variables, operations, functions, and procedures through argument verification.

While the method shows considerable promise in improving software reliability, its full potential in complex software systems will only be realized through further research and the development of specialized analysis tools.

Acknowledgements

We are grateful to Dr. D.I. Chumachenko for the invitation to participate in the conference, as well as to our colleagues from National Aerospace University for their support and assistance.

Declaration on Generative AI

During the preparation of this work, the authors used GPT-5 and Grammarly in order to: grammar and spelling check. After using these tools the authors reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

[1] Hamzah, M.; Islam, M.M.; Hassan, S.; Akhtar, M.N.; Ferdous, M.J.; Jasser, M.B.; Mohamed, A.W. Distributed Control of Cyber Physical System on Various Domains: A Critical Review. Systems (2023) 11, 208. doi:10.3390/systems11040208.

[2] Oks, S.J., Jalowski, M., Lechner, M. et al. Cyber-Physical Systems in the Context of Industry 4.0: A Review, Categorization and Outlook. Inf Syst Front 26 (2024): 1731–1772. doi:10.1007/s10796-022-10252-x

[3] Manzhos, Y., Sokolova, Y. The Software Development Lifecycle of Cyber-Physical Systems. Visnyk of Kherson National Technical University (2024), 1(88): 237-245. doi:10.35546/kntu2078-4481.2024.1.33

[4] Liubimov O, Turkin I, Pavlikov V, Volobuyeva L. Agile Software Development Lifecycle and Containerization Technology for CubeSat Command and Data Handling Module Implementation. Computation (2023): 11(9):182. doi:10.3390/computation11090182.

[5] Hoare, T. The Verifying Compiler: A Grand Challenge for Computing Research. In Modular Programming Languages; Böszörményi, L., Schojer, P. Eds.; Springer, Berlin, Heidelberg, 2003, pp. 25-35. doi:10.1007/978-3-540-45213-3_4.

[6] Boutekkouk, F. C Software Formal Verification. wipiec (2024): 10, 4.

[7] Krichen, M. A Survey on Formal Verification and Validation Techniques for Internet of Things. Appl. Sci. (2023): 13, 8122. doi:10.3390/app13148122.

[8] Longo, S.G. Principles and Applications of Dimensional Analysis and Similarity, Springer Cham, 2023. doi:10.1007/978-3-030-79217-6.

[9] Manzhos, Y.; Sokolova, Y. A Software Verification Method for the Internet of Things and Cyber-Physical Systems. Computation (2023), 11, 135. doi: 10.3390/computation11070135.
[10] Manzhos, Y.; Sokolova, Y. A type system for formal verification of cyber-physical systems C/C++ software. Radioelectronic and Computer Systems (2024), 1: 127-142. doi:10.32620/reks.2024.1.11.
[11] Martínez-Rojas, J.A.; Fernández-Sánchez, J.L. Combining dimensional analysis with model based systems engineering. Syst. Eng. (2022), 26: 71–87. doi:10.1002/sys.21646.
[12] Glavič P. Review of the International Systems of Quantities and Units Usage. Standards. (2021) 1(1) 2-16. doi:10.3390/standards1010002
[13] SI Units, 2023. URL: https://www.nist.gov/pml/owm/metric-si/si-units.
[14] Lischner, R. Programming at Compile Time. In Exploring C++20: The Programmer's Introduction to C++; Apress: Berkeley, CA, 2020, pp.643-653. doi:10.1007/978-1-4842-5961-0_73.
[15] Taylor, B.N. The Current SI Seen From the Perspective of the Proposed New SI. Journal of research of the National Institute of Standards and Technology (2011), 116(6): 797-807. doi:10.6028/jres.116.022
[16] Stavytskyi, P.; Voitko V.; Romanyuk, O. Analysis of metaprogramming capabilities in general-purpose programming language. Information Technology and Computer Engineering (2022), 55(3): 44-50. doi:10.31649/1999-9941-2022-55-3-44-50