UDC 004.415.5 **doi: 10.32620/reks.2024.1.11**

Yuriy MANZHOS, Yevheniia SOKOLOVA

National Aerospace University "Kharkiv Aviation Institute", Kharkiv, Ukraine

A TYPE SYSTEM FOR FORMAL VERIFICATION OF CYBER-PHYSICAL SYSTEMS C/C++ SOFTWARE

The subject: *This study focuses on improving the quality of Cyber-Physical System (CPS) software by eliminating incorrect usage of units of measurement and orientation in C/C++ programs. Incorrect usage often leads to critical errors that conventional systems cannot effectively prevent. Manual examination of code using dimensional and orientation analysis can detect these errors in physical equations, but these methods become impractical when dealing with complex physical computations.* ***Objectives****: As suggested by Siano, the proposed approach uses physical quantities and prefixes defined by the International System of Units and orientation operations on physical objects. The elaborated system incorporates dimensional and orientation analysis and metaprogramming techniques. The methods used are dimensional & orientational analysis and metaprogramming. The following* ***results*** *were obtained: ensuring consistency of the units, incorporating orientation operations into the programming model for accurately handling physical object rotations and alignments, and using Siano's work to precisely manipulate object orientation, thereby reducing the likelihood of orientation-related errors. Checking physical dimensions and orientations during the compilation stage identifies potential software defects before code execution, thereby reducing debugging time and lowering the cost of addressing issues later in development. The elaborated system represents a crucial step towards safer and more dependable Cyber-Physical System applications. This approach allows us to identify approximately 90% of incorrect usage of program variables; additionally, it detects over 50% of erroneous operations during compilation and execution of large-scale programs in real-world conditions.* ***Conclusions.*** *Scientific novelty: it proposed and developed a specialized C++-type library for formal compile-time software verification of Cyber-Physical Systems software. The proposed C++-type library leverages dimensional and orientational analysis to enhance software quality, reliability, and real-time formal verification. Although the proposed method for formal verification is not tailor-made for cyber-physical objects and systems, given its primary focus on software-level concerns, it does exhibit adaptability for verifying general-purpose software that incorporates various physical parameters. This versatility extends to diverse domains such as educational, gaming, and simulation software.*

Keywords: *Cyber-Physical Systems; dimensional analysis; formal verification; orientational analysis; software quality; type system.*

1. Introduction

Cyber-Physical Systems (CPS) are pivotal in modern society, delivering many benefits and applications that profoundly impact our daily lives. Notably, they drive the transformation of manufacturing processes, propelling Industry 4.0 [1] and setting the stage for the impending Industry 5.0 [2] revolution through automation [3], connectivity [4], and data-driven decision-making [5]. This leads to increased productivity, reduced downtime, and more agile production processes [6]. This can lead to better resource allocation, cost savings, and improved outcomes [7]. Therefore, the proposed study aims to utilize a specialized library for formal software verification.

1.1. Motivation

Implementing compile-time orientation and dimensional checking can be challenging, particularly as software reliability and quality requirements increase. This challenge motivated our study, which focuses on a specialized C++ type library that enables compile-time checking for the usage of various System International (SI) units physical quantities, including different orientations and SI decimal prefixes.

1.2. State of the art

CPS technologies: facilitate monitoring and managing environmental factors such as air quality, water usage, and energy consumption, promoting sustainability and minimizing environmental impact [8]; are pivotal in healthcare, enabling remote patient monitoring, telemedicine, and wearable health devices for more effective delivery [9], personalized treatment plans, and improved patient outcomes [10]; are pivotal in advancing smart infrastructure for cities and homes [11], integrating transportation, energy, waste management, and public services for more efficient urban living; enhance automation, energy efficiency, and remote control, providing heightened convenience and comfort; and in agriculture

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

monitor soil and weather and optimize practices for higher yields and resource efficiency, while CPS enables precise control of irrigation, fertilization, and pest management, further enhancing productivity and resource use [12]; in educational environments to create interactive learning experiences and research platforms. It provides hands-on learning opportunities in robotics, automation, and control systems [13].

CPS is at the heart of autonomous vehicle technology, enabling self-driving cars, drones, and other forms of transportation [14]. This can revolutionize mobility, improve safety, and reduce traffic congestion [15]. Aviation systems such as aircraft and Unmanned aerial vehicles (UAV) are CPS with several interacting automated control modules [16]. In aerospace, CPS is crucial for the operation of modern aircraft, including fly-by-wire systems, autopilots, and flight control systems. In UAVs, managing energy consumption is vital. This often involves employing adaptive data compression methods to minimize transmission overhead [17]. It ensures safe and efficient air travel.

CPS enhances energy efficiency in smart grids [18], reduces environmental impact, and seamlessly integrates renewable sources [19]. CPS is applied in defense systems, including UAVs, military robotics, and surveillance, to enhance situational awareness, intelligence gathering, and mission execution [20].

Ensuring the accuracy and reliability of CPS software is crucial for secure and reliable operation, as CPS systems involve a combination of hardware, firmware, communication protocols, and cloud services [21]. The following steps and techniques can be employed for the verification of CPS software [22]: Requirements Specification [23]; Functional Testing; Formal Verification (FV) [24]; Simulation and Testing; Code Reviews and Inspections; Static Code Analysis; Documentation Verification; Compliance with Standards and Regulations.

FV is a rigorous mathematical technique that proves that a system (including its hardware and software components) satisfies specific properties or requirements. When applied to CPS software, FV offers several benefits: FV instills high confidence in critical safety properties, which are vital for safety-critical CPS applications [25]; FV methods validate real-time properties, ensuring critical tasks meet timing requirements [26]; FV provides stability and performance of control algorithms in CPS [27]; Coupling formal methods with model-based design ensures accurate representation of system requirements and design [28]; FV spots flaws, inconsistencies, and ambiguities in specifications pre-implementation saving time and resources [29]; safety-critical domains often necessitate FV for demonstrating compliance with standards [30].

FV boosts confidence by detecting errors beyond testing, vital in complex systems such as autonomous vehicles. It complements testing methods and is applied selectively to critical components due to time and expertise constraints.

CPS applications, as well as scientific applications, heavily rely on the use of measurement units such as meters, seconds, kilograms, and so on (as specified in the SI system). Some software faults occurred during development because of incorrect physical quantities and system unit usage [31]. An illustrative instance of a software fault arising from the erroneous use of physical quantities occurred with the Therac-25 radiation therapy machine in the 1980s. In the 1983 Gimli Glider incident, an Air Canada Boeing 767 experienced a fuelling error during maintenance when converting from imperial to metric units. The incorrect conversion factor resulted in the aircraft being loaded with only half the necessary fuel. During the 1991 Gulf War, the Patriot missile defence system experienced a critical software fault due to the system's internal clock measurement in tenths of a second, resulting in cumulative errors and inaccurate missile position calculations [32]. In the 1999 Mars Climate Orbiter mission, a critical software fault arose from incorrect unit conversion, causing the spacecraft to enter the atmosphere of Mars at an inadequate altitude and destroy it [33].

Physical unit and Orientation checking is of utmost importance in satellite and UAV (Unmanned Aerial Vehicle) software testing for several critical reasons: Ensures consistent and correct usage of units of measurement, preventing errors that may result from incompatible units in calculations; Unit and orientation checking enhances code readability and maintainability by clearly defining units and orientations, making it easier for developers to understand and debug the code; Physical unit checking is crucial in avoiding costly errors in critical systems, as incorrect units can lead to catastrophic failures, and implementing this check early in development helps prevent potentially life-threatening situations; Compile-time checking for correct usage reduces debugging efforts by addressing potential issues before deployment, saving time and effort in later stages of development. Ensuring accurate mathematical operations with physical quantities is crucial in precision-dependent scientific and engineering applications.

Accurate orientation checking is vital in aerospace and defence for tasks such as flight simulation, missile guidance, and satellite positioning, as well as for preventing drift in inertial navigation systems. It is essential for precise positioning and heading of satellites and UAVs, ensuring that they stay on the intended trajectories and avoid collisions. Proper orientation checking is critical for safety and equipment preservation, playing a fundamental role in stabilizing and controlling the attitude of satellites and UAVs relative to Earth or other reference

points, and is crucial for risk reduction in congested or restricted airspace.

Incorporating physical unit, orientation, and dimension checking is crucial for verifying software correctness, reliability, and safety, especially in domains reliant on precise measurements. This practice plays a vital role in satellite and UAV development, ensuring platform and payload compatibility, safety, and reliability, which is pivotal for aerospace mission success. As stated in [34], simple dimensional analysis is essential for identifying relevant quantities in specific problems.

Our article [35] stated that GitHub hosts over 2 GB of UAV-related C/C++ source code. Physical unit checking in software ensures consistent and accurate use of measurement units throughout the program. Numerous specialized libraries are available for implementing this process:

Boost.Units is a C++ library [36] designed to precisely manage physical quantities and units, enabling custom unit definitions with compile-time error detection for unit-related concerns. It was first included in Boost 1.36.0, which was released in 2008. Benri, a C++ library created by Jan in 2018, focuses on compile-time checking of physical quantities. Benri provides extensive support for various systems of units, physical constants, mathematical operations, and affine spaces [37]. Mp-units, a compile-time enabled Modern C++ library by Mateusz Pusz in 2020, provides compile-time dimensional analysis and unit/quantity manipulation [38]. PHYSUNITS-CT-CPP11, a C++ library based on the work of Michael Kenniston from 2001, expanded, and adapted for C++11 and C++14 by Martin Moene [39] in 2020, is a header-only library that provides compile-time dimensional analysis and unit/quantity manipulation and conversion.

Orientation checking, which is essential in computer graphics and robotics, involves operations such as unit quaternion normalization to maintain the validity of orientation representations. Here are C/C++ libraries/tools for this purpose:

Eigen is a C++ template library for linear algebra, written in 2021, which includes support for quaternions. It provides functions for quaternion operations, including normalization [40]; GLM (OpenGL Mathematics) is a C++ library for graphics programming, offering functions, structures, and support for quaternions, including normalization [41]; Quaternion library for C is A basic quaternion library written in C by Martin Weigel in 2018. This library implements the most basic quaternion calculations [42].

We need new tools for formal verification based on diverse principles because diversifying verification methods allows for an increase in software quality.

Unfortunately, standard-type systems do not enforce the proper use of physical dimensions and orientations, leaving room for potential errors and inconsistencies. To address this challenge, physicists and engineers often employ dimensional analysis [43] to verify the dimensional unit correctness of quantities in equations. Dimensional analysis assumes that each physical quantity has a well-defined, fixed unit of measure, requiring the units on both sides of the equation to match. While dimensional analysis is useful, it can be challenging for non-physicists. Many physical equations involve complex computations, making it difficult to accurately track the flow of units throughout the calculations. The manual application of dimensional analysis to programs that involve such equations can further intensify the complexity.

According to [44], 38 of the most comprehensive and well-developed open-source libraries provide built-in support for units and dimensions in software development. In [45], Steve McKee classified the many software solutions measurement units checking. Steve proposed software development based on Units of Measurement.

Similarly, orientational analysis [46, 47] assumes that each physical quantity has a meaningful, fixed orientation in space, and it requires the orientations on both sides of an equation to align [48]. Similar to dimensional analysis, orientational analysis can be challenging, especially for individuals without a strong physics background. Complex computations in physical equations make it difficult to accurately trace the flow of units and orientations. Therefore, manually applying dimensional and orientational analysis to programs involving such equations can be even more daunting.

In summary, ensuring the correctness of physical dimensional units and orientations is essential for CPS and scientific applications. The limitations of the standard-type system in enforcing these constraints make it necessary to incorporate methods like dimensional analysis and orientational analysis. However, manually applying these analyses to programs that involve complex equations can be intricate and time-consuming, highlighting the need for more efficient and automated approaches to ensure the accuracy and reliability of these applications.

1.3. Objectives and methodology

This paper presents a novel specialized type library (TL) to facilitate the formal verification of C++ software at compile-time and run-time. By leveraging this library, developers can enhance the reliability and correctness of their C++ programs through rigorous verification techniques. Benefits of the proposed library:

– Extensibility: The library allows for easy expansion to accommodate new domains and physical quantities;

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

– Addition of orientation to quantities: The library enables the inclusion of orientation information for the quantities;

– Adaptive use of prefixes: The library allows for the adaptive use of prefixes for units, enabling flexibility in expressing quantities.

As Siano suggested, the proposed approach uses physical quantities and prefixes defined by SI and orientation operations on physical objects. The elaborated C++ type system incorporates dimensional and orientation analysis and metaprogramming techniques. The methods used are dimensional & orientational analysis and metaprogramming.

The TL introduces a set of specialized types that enforce specific constraints and properties during program execution. These types enable compile-time verification by utilizing static analysis techniques to detect potential errors and inconsistencies in the code before execution. By catching these issues early, developers can prevent runtime errors and enhance the overall robustness of their software.

Furthermore, the TL extends its functionality to run-time (real-time) verification. During program execution, the library dynamically checks the validity of the program's state and behavior against the defined constraints. This dynamic verification process provides an additional safety net, ensuring that the software adheres to the intended specifications and behaves as expected.

The library's formal verification capabilities enable developers to reason about their C++ programs more rigorously and systematically. Providing a higher level of assurance make it possible to detect and prevent various types of errors, including type mismatches, undefined behavior, and violations of specified invariants. This significantly reduces the risk of bugs, improves the code quality, and enhances the overall reliability of the software. In addition to its verification features, TL integrates seamlessly into the C++ development workflow. It provides clear and expressive interfaces that allow developers to specify constraints and properties concisely and readably. The library also offers extensive documentation and support, making it accessible to developers with varying levels of expertise in formal verification.

By adopting this TL, C++ developers can elevate the quality of their software by incorporating formal verification techniques into their development process. By combining the advantages of compile-time and run-time verification, the library offers a comprehensive approach to ensure the correctness and robustness of C++ software, ultimately leading to increased confidence in its reliability and improved software quality.

The objectives of the investigation are:

– develop a novel specialized type library (TL) for formal verification of C++ software at compile-time and run-time, enhancing reliability and correctness, while ensuring extensibility by accommodating new domains, physical quantities, and orientation information;

– incorporate dimensional and orientation analysis, along with metaprogramming techniques, into the C++ type system;

– implement both compile-time and run-time verification to detect errors, check program validity, and elevate software quality;

– seamlessly integrate the TL into the C++ development workflow for rigorous reasoning about programs and improved code quality.

The structure of this paper is as follows:

1. Describes the software verification model used in the study, including its key components and methodologies (subsection 2.1).

2. Explains the fundamental principles underlying the specialized TL, including its design considerations and core functionalities (subsection 2.2).

3. Details the implementation of operators and function wrappers within the TL framework, highlighting the use of templates for flexibility and efficiency (subsection 2.3).

4. Provides guidelines and examples for using the TL in C++ software development, demonstrating its practical application and benefits (subsection 2.4).

5. Outlines the verification process for the TL, including both compile-time and run-time verification techniques employed to ensure its correctness and reliability (subsection 2.5).

6. This section presents the findings of the study, including empirical results and insights gained from the implementation and verification of the TL (section 3).

7. Summarizes the key findings of the study and discusses their implications for C++ software development and formal verification practices (section 4).

8. Suggests potential avenues for future research and development in the field of formal verification and C++ software engineering, building upon the findings of the current study (section 5).

2. Materials and Methods

2.1. Software verification model

The proposed TL will enable the formal verification of the embedded software. Integrating TL into the software development process ensures the correctness and reliability of the software's behavior and its interactions with the physical world. TL's capability to manage physical dimensions, units, and orientations offers a potent tool for static checking and validation, thereby reducing the risk of errors and enhancing the overall quality of the embedded software.

The software formal verification model employs invariants checking to ensure dimension and orientation homogeneity (see Fig. 1). By enforcing these invariants, and the model proves that the software components and operations maintain consistent physical dimensions and orientations throughout their execution. This approach guarantees the integrity of calculations and prevents incompatible combinations of quantities, resulting in more reliable and accurate software behavior. The TL is crucial in supporting this verification process by providing the necessary tools and mechanisms to enforce and validate dimension and orientation consistency.

The software verification process involves several steps. Based on the technical documentation, the first step is to set the physical dimensions and orientations of the input and output SW variables. This is done by using the source code of every function as the body of a method of a special testing class.

The second step involves modifying the C++ source code by overriding standard data types to match the physical dimensions and orientation of the input and output variables.

In the third step, a standard compiler detects SW defects, such as violations of dimensional and/or orientational homogeneity. After modifying the SW units, the modified units are compiled again.

In the fourth step, the test cases are used for software testing after compiling and linking editing. The test cases' negative results indicate dimensional and/or orientational homogeneity violations. This allows for modification of the source code to correct any detected issues. The test cases also allow for checking the correctness of different pointer operations during dynamic linking in C++.

In the fifth and final step, after software verification, invariant checking can be performed in real time to ensure that the software is operating correctly.

Overall, this software verification process provides a reliable and effective method for detecting and correcting dimensional and orientational homogeneity violations in C++ code, ensuring high-quality and reliable software.

2.2. Key Principles of Type Library

The International System of Units (SI) consists of seven base units: the amount of substance, current, length, luminous intensity, mass, time, and thermodynamic temperature and dimensionless symbol. Each base unit is associated with a unit symbol and a dimension symbol. According to the Siano convention [46, 47], length is commonly understood to possess a fixed orientation in space and is categorized as orientationless, x-oriented, y-oriented, or z-oriented. This orientation is defined by the symbol "O" (Table 1). In SI, each physical quantity is defined as the product of base units raised to certain powers. These base units serve as the fundamental building blocks for expressing various measurements.

When SI units are used, physical values are associated with specific subject areas (SA). In cases where a dimension has an orientation, the corresponding value can have one of the following directions: {0, X, Y, Z}. A value of 0 indicates an orientationless quantity, while X, Y, and Z denote values oriented in the respective directions. For instance, quantities such as force and acceleration fall into this category. On the other hand, certain physical values, including mass and current, are considered to be orientationless. This means that they do not possess a specific direction associated with them.

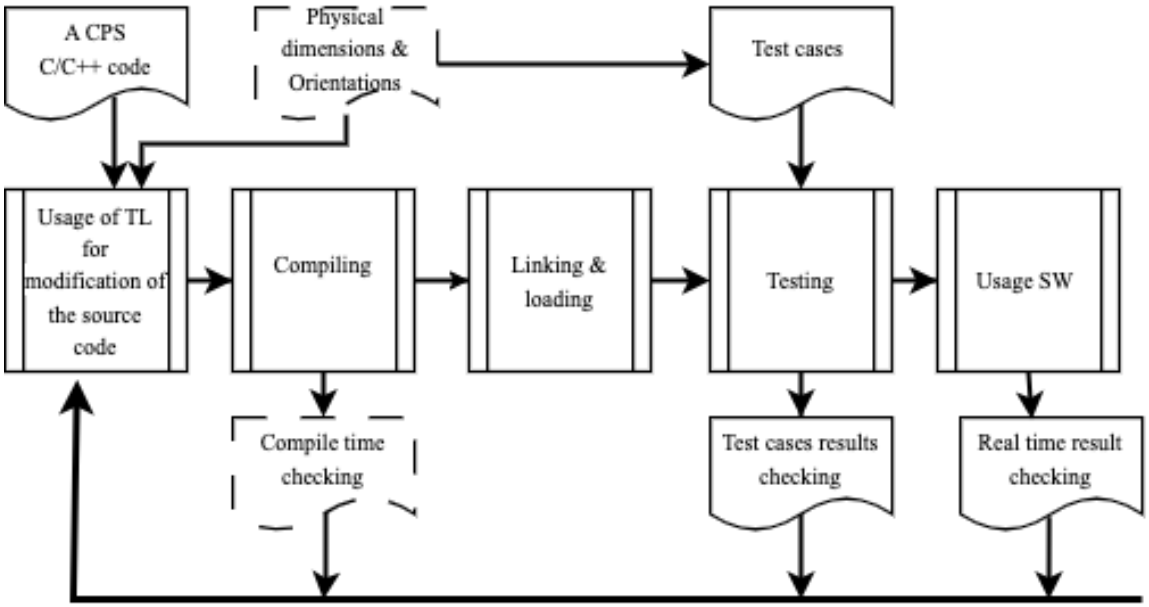

Fig. 1. Functional model of software formal verification

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

Table 1

Base Units of SI & Dimensionless Unit

SI unit	Unit [Symbol]	Dimension symbol
Amount Of Substance	[mol]	N
Current	[A]	I
Length	[m]	LO
Luminous Intensity	[cd]	J
Mass	[kg]	M
Time	[s]	T
Thermodynamic Temperature	[K]	Θ
Dimensionless	[1]	O

Table 2 and Table 3 offer a comprehensive compilation of the common units of measurement used across diverse subject areas

Table 2

Units of Measurement for Quantities in the Subject Area of "Geometry"

Quantity	Unit [Symbol],{Dimension}
Area	[m^2], {L^2 O}
Aspect Ratio	[1], {1}
Curvature	[1/m], {L^{-1} O}
Perimeter	[m], {L}
Plane Angle	[rad], {O}
Solid Angle	[Sr], {1}
Surface Area	[m^2], {L^2 O}
Volume	[m^3], {L^3}

Table 3

Units of Measurement for Quantities in the Subject Area of "Optics and Photometry"

Quantity	Unit [Symbol], {Dimension}
Illuminance	[lx], {L^{-2}JO}
Luminance	[cd/m^2], {L^{-2}JO}
Luminous Efficacy	[lm /W], {$T^3L^{-4}M^{-1}$J}
Luminous Energy	[lm s], {TJ}
Luminous Exposure	[lx s], {TL^{-2}JO}
Luminous flux or Luminous power	[cd sr], {J}
Optical Power	[1/m], {L^{-1}O}
Refractive Index	[1], {O}

These tables serve as valuable references, allowing users to conveniently access and apply the appropriate units in their respective fields of study or work. We can create similar tables for different subject areas covered, encompassing Chemistry, Density & Concentration, Electricity & Magnetism, Flow, Physics and Thermodynamics, etc. Importantly, these tables should be considered a foundation, providing a starting point for users to expand upon based on their specific requirements and subject areas of interest. Users are encouraged to augment the list to accommodate their unique needs and ensure comprehensive coverage within their chosen domain.

According to these subject areas, we have more than 80 different orientationless physical quantities and more than 60 oriented physical quantities.

To implement compile-time software formal verification, we need to create a total of 320 (80 + 60 x 4) different C++ classes. Considering both dimension and orientation homogeneity, we must create overloading operators for product and division between these 320 classes, resulting in a total of 320 x 320 x 2 operators. When dealing with expressions that involve the product of multiple values (n-values), it is necessary to overload the product and division operators to accommodate the varying number of operands. In total, 2 x 320 x n overloads will be required for these operators. Clearly, such a TL would be considerably complex and substantial in size. However, the advantage of using this TL is that it enables the execution of formal verification during compile time.

A common approach to representing units is to utilise exponent vectors based on base units (as shown in Table 1) and unit factors. For instance, the dimension of an orientationless unit of force is $T^{-2}L^1M^1$, which can be represented as [-2, 1, 1, 0, 0, 0, 0] using exponent vectors. On the other hand, an oriented force has dimension $T^{-2}L^1M^1O$, where O can take values of l_x, l_y, or l_z, representing the orientation in the $x \equiv 1$, $y \equiv 2$, or $z \equiv 3$ direction, respectively. For example, the x-oriented force has vector = [-2, 1, 1, 0, 0, 0, 0, 1]; the orientationless quantity, such as mass, has an orientation of l_0. We can distinguish between quantities with the same dimension but different orientations by employing both orientational and dimensional analysis. For example, it considers energy [Newton x meter] and torque [Newton x meter].

Thus, arithmetic operations on units can be simplified to vector additions, subtractions, or comparisons. These operations can be performed only on physical quantities with the same dimension or corresponding to vectors with identical coordinates. In other words, units with matching exponent vectors can be directly added, subtracted, or compared using these operations.

However, complications arise when dealing with the product and division of physical quantities. For orientationless quantities, we can simply add or subtract their corresponding exponent vectors to obtain the resulting vector. However, for orientated quantities, determining the resulting vector requires more than simply adding or subtracting the exponent vectors. We also need to consider Siano's rules, which help define the orientation of the resulting vector [46, 47]. Siano demonstrated that ori-

entational symbols have an algebra defined by the multiplication table for the orientation symbols, which is as follows:

	l_0	l_x	l_y	l_z
l_0	l_0	l_x	l_y	l_z
l_x	l_x	l_0	l_z	l_y
l_y	l_y	l_z	l_0	l_x
l_z	l_z	l_y	l_x	l_0

(1)

and rules: $l_0 = \frac{l}{l_0}$ $l_x = \frac{l}{l_x}$ $l_y = \frac{l}{l_y}$ $l_z = \frac{l}{l_z}$

Based on the above, the product of two orientated physical quantities has an orientation as follows:

$$l_0 l_x = l_x l_0 = l_x, l_0 l_y = l_y l_0 = l_y, \quad l_0 l_z = l_z l_0 = l_z, l_x l_x = l_y l_y = l_z l_z = l_0. \tag{2}$$

However, a common approach to representing units is to utilise exponent vectors based on base units and unit factors, which has some constraint: we may realize software formal verification only in run time.

To enable the use of compile-time formal verification, we present TL that encompasses the key components designed to enhance the verification process. The TL comprises several essential classes, including the "PNSD_SI" class for facilitating operations with SI prefixes (e.g., nano, milli, giga, etc.). This class enables seamless handling of units of different magnitudes.

Another integral component of TL is the "Printing" class, which provides comprehensive control over output formatting. This class empowers developers to customize the display of results, ensuring a clear and informative representation of data.

Additionally, the TL incorporates a template class called "PhysicalVariable" that plays a vital role in dimensional analysis and orientational analysis operations. This class allows for the precise handling of physical quantities, considering both their dimensions and orientations. Moreover, the "PhysicalVariable" class inherits essential functions from the "Printing" class, enabling seamless integration of output control capabilities.

The architecture of the TL is depicted in Fig. 2, which showcases the relationships and dependencies between the classes. The template class "PhysicalVariable" serves as a powerful tool for creating various C++ classes that correspond to specific physical quantities (

Table 2 and Table 3) [49]. This template class, combined with metaprogramming [50] techniques, forms the foundation of a library that enables the generation of new types at compile time [51].

To simplify the creation of these classes, the library provides special pre-processor macros that leverage the "PhysicalVariable" template. These macros facilitate the generation of C++ classes during the compilation process. The use of these macros is straightforward: developers define the desired quantity name, unit name, unit symbol, and a vector representing the dimensions based on the basis dimensions T (time), L (length), M (mass), I (electric current), θ (thermodynamic temperature), N (amount of substance), and J (luminous intensity) (see Table 1).

For instance, the following macro invocation creates a class named "Density," representing the orientationless physical quantity of density:

```
createSomeUnit(Density, "kilogram per cubic
metre", "kg / m3", 0, -3, 1, 0, 0, 0, 0).
```

The next macro allows the creation of four C++ classes: "Area" (representing orientationless quantities), "AreaX" (representing x-oriented quantities), "AreaY" (representing y-oriented quantities), and "AreaZ" (representing z-oriented quantities):

```
createSomeUnit0XYZ(Area, "square meter",
"m2", 0, 2, 0, 0, 0, 0, 0).
```

Developers can easily generate the necessary C++ classes for their desired physical quantities by employing these macros in the development process. This streamlined approach leverages the power of metaprogramming and compile-time generation to create a comprehensive library of types that accurately represent physical quantities and their orientations.

Table 2 and Table 3 show that each generated class is mapped to a specific subject area. Each subject area is associated with a C++ namespace. These namespaces are included within the SI namespace (see Fig. 3).

The mapping of each generated class to a specific subject area, as described in

Table 2 and Table 3, is achieved through association with a dedicated C++ namespace. This approach ensures that classes related to Chemistry, Optics, Photometry, etc. These subject-specific namespaces are encapsulated within the SI namespace to maintain a well-structured codebase. This hierarchical structure promotes modularity, clarity, and ease of navigation within the codebase, facilitating efficient development and maintenance of the library.

2.3. Implementation of Operators and Function Wrappers Using Templates

The proposed library adheres to the principles of homogeneity in physical dimensions and orientations by ensuring that the left and right operands have equal dimensions and orientations. The assignment operators (=, +=, -=) are function members within the template class.

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

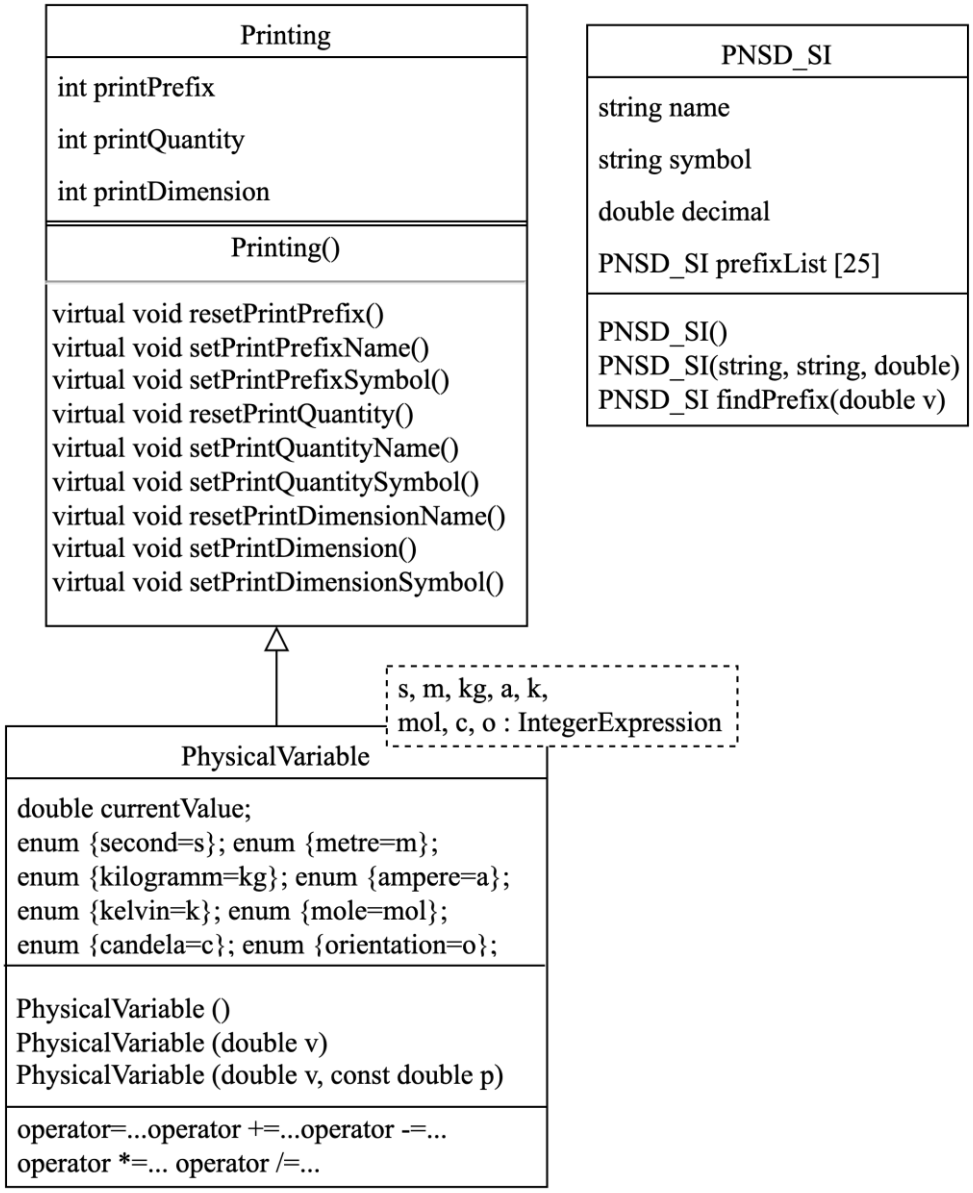

Fig. 2. Architecture of the Specialized Type Library

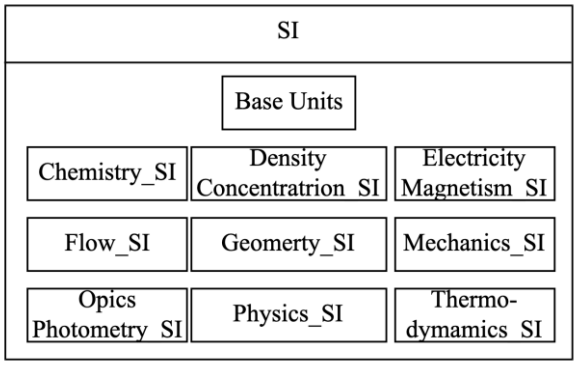

Fig. 3. SI namespace structure

For the conditional operators (>, >=, ==, !=, <=,<), as well as the addition (+) and subtraction (-) operators, template functions are employed with template class arguments. These operators utilize the Siano conventions (see expressions 1 and 2).

The multiplication and division operators are also defined as template functions that utilize template class arguments and adhere to the Siano conventions. These operators enable proper handling of physical quantities during calculations.

The compound assignment operators (*=, /=) require a dimensionless right operand. To preserve the dimension and orientation of the result, these operators are defined as template function members.

The implementation of this library provides a significant advantage in terms of unit testing, integration testing, and regression testing. The effectiveness of static checking during compilation reduces the time required

for these testing activities. By catching errors and inconsistencies at compile time, developers can identify and address issues early in the development process, improving software quality and reducing debugging efforts.

Overall, the proposed library ensures consistency, accuracy, and efficiency by enforcing physical dimensions and orientations homogeneity and leveraging static checking during compilation.

Wrapper functions are utilized with template arguments to accommodate the dimensionless and orientationless arguments of the exp and log functions. These wrapper functions aim to ensure that the correct version of the exp and log functions is called for the PhysicalVariable instances. Here are the improved versions of the wrapper functions:

```
double exp(PhysicalVariable <0, 0, 0, 0, 0, 0, 0, 0>pv)
{ return ::exp(pv.value()); }
double log(PhysicalVariable <0, 0, 0, 0, 0, 0, 0, 0>pv)
{ return ::log(pv.value()); }
```

By specifying the template argument <0, 0, 0, 0, 0, 0, 0, 0>, the wrappers ensure that only dimensionless and orientationless instances of PhysicalVariable can be passed as arguments to the exp and log functions. In this way, the correct mathematical operations can be applied to these specific instances, guaranteeing the accuracy and integrity of the calculations.

These wrapper functions are crucial in maintaining the consistency and correctness of operations involving dimensionless and orientationless quantities within the proposed library.

To handle the square root function (sqrt) within the template framework, we created a wrapper function that correctly takes the dimensions of the result.

```
template < int T, int L, int M, int I, int θ, int N, int J>
PhysicalVariable < T/2, L/2, M/2, I/, θ /2, N/2, J/2, 0, 0>
sqrt(PhysicalVariable < T, L, M, I, K, N, J, 0> p)
{ return
 PhysicalVariable < T/2, L/2, M/2, I/, K/2, N/2, J/2, 0>
(::sqrt(p.value()));}
```

This wrapper function takes a PhysicalVariable instance as an argument, where the dimensions are represented by the template parameters T, L, M, I, θ, N, J, and O. The sqrt function calculates the square root of the value stored in the PhysicalVariable instance. Creates a new PhysicalVariable instance with dimensions halved for each base unit.

To create a function wrapper for x^n or pow(x, n), where x represents a dimensioned and orientational value and n is an integral number, the following struct template, function template, and macro can be used:

```
template <int num>
struct powN{ enum { np = num }; };
template
< int T, int L, int M, int I, int θ, int N, int J, int O, int n>
PhysicalVariable
< T*n, L*n, M*n, I*n, θ*n, N*n, J*n, (O*n)%2>
    powPhysicalVariable(PhysicalVariable
    < T, L, M, I, θ, N, J, O > left, powN< n>)
    { return  PhysicalVariable
    < T*n, L*n, M*n, I*n, θ*n, N*n, J*n, (O*n)%2>
    (::pow(left.value(), n)); }
#define pow(x,y) powPhysicalVariable( x, powN<y>())
```

This implementation allows the calculation of the power of a dimensioned and orientational value (x) raised to an integral exponent (n).

To handle trigonometric functions (sine, cosine, tangent, arcsine, arccosine) within the template framework, we created wrapper functions that correctly take the dimensions of the result.

```
#define X SI::Geometry_SI::PlaneAngleX
#define Y SI::Geometry_SI::PlaneAngleY
#define Z SI::Geometry_SI::PlaneAngleZ
#define cosXYZ(L) double cos(L pv)
{ return ::cos(pv.value());}
cosXYZ(X)  cosXYZ(Y) cosXYZ(Z)
#define FXYZ(F,L)\
Dimensionless##L
F(SI::Geometry_SI::PlaneAngle##L pv)\
{ return Dimensionless##L (::F(pv.value())) ;}
FXYZ(sin, X) FXYZ(sin, Y) FXYZ(sin, Z)
FXYZ(tan, X) FXYZ(tan, Y) FXYZ(tan, Z)
double acos(Dimensionless pv)
{ return ::acos(pv.value()); }
X  asinx(double v){ return X(::asin(v)); }
Y  asiny(double v){ return Y(::asin(v)); }
Z  asinz(double v){ return Z(::asin(v)); }
X  asinx(Dimensionless v)
{ return X(::asin(v.value())); }
Y  asiny(Dimensionless v)
{ return Y(::asin(v.value())); }
Z  asinz(Dimensionless v)
{ return Z(::asin(v.value())); }
```

2.4. Using the Type Library

Using TL is a straightforward process. In your C++ file, include the necessary namespaces to access the desired subject areas. Here is an example of SI namespace and its subnamespaces:

```
using namespace SI;
using namespace SI::Optics_Photometry_SI;
using namespace SI::Electricity_Magnetism_SI;
using namespace SI::Thermodynamics_SI;
```

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

By including these namespaces, you can access the classes and functionality related to each subject area. It allows you to use the units, perform calculations, and leverage the features provided by the TL.

We can create an alias for a long or nested namespace using the namespace aliasing feature as follows:

```
namespace ZSI = SI::Electricity_Magnetism_SI;
int main() {
   ZSI::Capacitance c;
   // Use the alias to access the classes
    return 0;
}
```

This modular approach allows users to extend and organize subnamespaces structured, resulting in enhanced code reusability and maintainability.

By dividing the functionality into subnamespaces, developers can logically group related classes and functions, making it easier to locate and reuse code across different projects. In addition, it promotes a modular design where each subnamespace can be independently extended or modified without affecting other parts of the codebase.

Furthermore, this approach enhances the code organization making it more intuitive and understandable. Developers can navigate through the codebase more efficiently by understanding the purpose and scope of each subnamespace.

This modular approach fosters better code management, encourages code reuse, and facilitates future modifications and enhancements. It is a powerful technique for structuring and maintaining complex projects effectively.

2.5. Verification of the C++ Type Library

Based on Table 1, we have eight base units. Furthermore, our TL covers specific Subject Areas (see Table 4). Every physical quantity corresponds to a distinct C++ class generated by TL. The distinct physical quantities used in the program determines the number of generated classes. These classes have various overloaded operators that facilitate arithmetic, mathematical, and logical operations.

To verify the functionality of the TL, dedicated C++ units were created for each Subject Area. Special Test Cases were generated for verification operations with different physical quantities. Cartesian products of Subject Areas defined the operands of this operation. Additional calculation errors based on dimensionality checks were eliminated using operation templates and classes. Further 24 tests were conducted to verify the use of SI prefixes.

Table 4

Units of Measurement for Quantities in the Subject Areas

Subject Areas	Numer of quatities
Chemistry	11
Density & Concentration	7
Electricity & Magnetism	31
Flow	7
Geometry	8
Mechanics	31
Optics and Photometry	8
Physics	15
Thermodynamics	21

Each physical quantity in our software system corresponds to a C++ class specifically designed for this purpose. These classes are generated using a special template. Each object of these classes occupies 24 bytes of memory. Of these, 8 bytes are allocated for storing the current value, whereas an additional 16 bytes are used to store information about the physical dimension and orientation. A detailed breakdown of memory usage by the executable file is provided in Table 5.

Table 5

Memory usage by the executable file

Subject Areas testing	Executable file, KB
Without physical values	67
Usage TL without physical values	1198
Base SI quantity	1254
Geometry	1319
Optics and Photometry	1444
Chemistry	1540
Density & Concentration	1612
Flow	1659
Mechanics	2010
Thermodynamics	2202
Physics	2288
Electricity & Magnetism	2758
Additional tests of cout	2771
Software 1	1694
Software 2	1697

During Subject Areas testing (see Table 5), subject area physical quantities were used as extended test variables. For example, for testing Geometry quantities, use SI base units and Geometry quantities, etc. The number of physical quantities in the subject area rather than by the number of program variables with the same physical quantity determines increasing the executable file size. All tested programs used 6800 KB of RAM. Software 1 encompasses all SI base units and incorporates all physical quantities within the subject areas of Geometry and

Physics. Software 2 includes all SI base units and encompasses all physical quantities in the subject areas Geometry and Physics, along with an additional 30,000 variables corresponding to different physical quantities, requiring 7500 KB of RAM.

This section will focus on verifying the Geometry_SI namespace as an example. Let us consider the operations with the Length and Area classes within the Geometry_SI namespace as follows:

```
Length0 l0(2); // orientationless l0=2[m]
LengthX lx(3); // X-oriented lx =3[m ]
LengthY ly(4); // Y-oriented ly =4[m ]
LengthZ lz(5); // Z-oriented lz =5[m ]

Area a(100),b; // orientationless a =100[m2],b
AreaX ax(200); // X-oriented ax =200[m2 ]
AreaY ay(300); // Y-oriented ay =300 [m2]
AreaZ az(400); // Z–oriented az =400 [m2]

// calculation of new values of areas
a = l0 * l0;    // a= 100 is correct
ax = ly * lz; // ax= 20 is correct
ay = lx * lz; // ay= 15 is correct
az = lx * ly; // az = 12 is correct
b = lx *lz; // orientational error is not correct!!!
double k = 10;
az *= k;      // is correct az= 120
ay /= k;      // is correct az= 1.5
```

In the above code, we create instances of the Area and Length classes, representing quantities with different orientations.

In the next example, we check the operations involving the Curvature class (Optics Photometry_SI namespace) and Length class (Geometry_SI namespace).

```
Curvature c(1); //orientationless c=1[1/m]
CurvatureX cx(1);//X-oriented cx=1[1/m]
CurvatureY cy(2);//Y-oriented cy=1[1/m]
CurvatureZ cz(3); //Z-oriented cz=1[1/m]
Length0 L0(2); // orientationless L0=2[m]
LengthX Lx(3); // X-oriented Lx =3[m ]
cx =L0 / LX; //dimension error!!
cx = l.0 / Lx;
lx = 1.0 * Lx;
```

Now let us proceed with the verification of trigonometric functions within the TL:

```
Dimensionless dl(1), dlx(1), dly(1), dlz(1);
PlaneAngle pa0(1);  //[radian        rad l=0]
PlaneAngleX pax(1); //[radian        rad l=x]
PlaneAngleY pay(2); //[radian        rad l=y]
PlaneAngleZ paz(1); //[radianrad l=z]

double lcosd = cos(1.25), lsind = sin(1.25),
ltan = tan(1.25),
acosd = acos(1.25), asind = asin(1.25);

Dimensionless lcosx = cos(pax),
lcosy = cos(pay), lcosz = cos(paz);

DimensionlessX lsinx = sin(pax), ltanx = tan(pax);
DimensionlessY lsiny = sin(pay), ltany = tan(pay);
DimensionlessZ lsinz = sin(paz), ltanz = tan(paz);

PlaneAngleX acosx = acos(dlx),
asinxd = asinx(dl);
PlaneAngleY acosy = acos(dly),
asinyd = asiny(dl);
PlaneAngleZ acosz = acos(dlz),
asinzd = asinz(dl);

double d0 = sin(pa);  //compile errors!!!
double dx = sin(pax); //compile errors!!!
double dy = sin(pay); // compile errors!!!
double dz = sin(paz); // compile errors!!!
```

The provided code snippet focuses on verifying specific classes within the Geometry_SI namespace. However, similar checks were conducted for all other classes and operations within the TL to ensure their correctness and adherence to the defined rules and principles.

SI defines a set of prefixes of physical values. The provided code defines a list of prefixes used in the SI to denote physical values. Each prefix is associated with a name, symbol, and the corresponding decimal factor:

```
const PNSD_SI prefixList[24] = {
{"quetta","Q", 1e30 },  {"ronna", "R", 1e27},
{"yotta", "Y", 1e24},…
{"zepto", "z", 1e-21},  {"yocto",  "y", 1e-24},
{"ronto", "r", 1e-27},  {"quecto", "q", 1e-30} };
```

In the following code example, we observe the use of prefixes for initializing physical quantities:

```
Mass m(15., prefix::micro);// m=1.5 * 10^-5 [kg]
Mass m2(20.,nano); // m2=2 10^-8 [kg]
```

The proposed library has special methods for printing physical values and prefixes.

In the following Fig. 4, you can observe the output of physical quantities with different settings: printing prefixes (without prefix, name, symbol), printing quantities (without quantity, name, symbol), and printing dimensions (without dimension and dimension vector).

This comprehensive output process ensures the reliability verification of the CPS-embedded software.

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

The proposed TL was developed with the assistance of Microsoft Visual Studio Community 2022 Version 17.1.2. This widely used development environment played a crucial role in the elaboration and creation of the TL, ensuring compatibility and leveraging the powerful features and tools provided by Visual Studio for efficient development and implementation.

```
i1=1 h["A"]i2=2 nano["ampere"][ s0 m0 kg0 A1 K0 mol0 cd0 ]
i2=4 nano["ampere"][ s0 m0 kg0 A1 K0 mol0 cd0 ]
i2=1 hecto["ampere"][ s0 m0 kg0 A1 K0 mol0 cd0 ]
edf=1.2 ["coulomb per square metre"]
b=2.2e-11
r1=1 quecto["ohm"][ s-3 m2 kg1 A-2 K0 mol0 cd0 ]
r2=0.01 quecto["ohm"][ s-3 m2 kg1 A-2 K0 mol0 cd0 ]
r1=1 kilo["ohm"][ s-3 m2 kg1 A-2 K0 mol0 cd0 ]
```

Fig. 4. Fragment physical quantities output

The thorough verification process ensures the reliability, accuracy, and effectiveness of the TL implementation across diverse subject areas and namespaces. This process instills confidence in the TL's functionality and usability, making it a dependable tool for software development and related tasks.

3. Results and Discussion

This article introduces a groundbreaking C++-type library based on metaprogramming. This innovative library incorporates SI prefixes and dimensional analysis and integrates orientational analysis, demonstrating its remarkable effectiveness in identifying software defects. The proposed method requires additional memory. For instance, a program without TL requires only 100 KB, whereas using TL extends the executable file size to 1-3 MB.

Notably, it has demonstrated the capability to identify over 60% of software defects [17], including those stemming from incorrect usage of variables, operations, SI prefixes, and C++ functions. These results suggest that this library has substantial potential as a valuable tool for software development.

However, it is essential to acknowledge that no single library can detect all software defects. Therefore, while the proposed library, based on SI, shows promise, it requires thorough evaluation and comparison with other library types based on different system units.

Furthermore, this study proposed a software verification model that leverages the type library for formal CPS software verification during compile time and runtime. This approach represents a significant advancement in ensuring the reliability and safety of CPS software, making a crucial contribution to this evolving field.

Although the proposed method for formal verification is not specifically tailored for cyber-physical objects and systems, given its primary focus on software-level concerns, it does demonstrate adaptability for verifying general-purpose software that incorporates various physical parameters. This versatility extends across diverse domains, such as educational, gaming, and simulation software.

4. Conclusions

This article introduces a novel C++ type library based on software invariants for formal verification. The proposed TL leverages both dimensional and orientational analysis to enhance the software quality. Employing two independent formal software verification methods offers diverse and robust verification capabilities, leading to improved software quality.

The proposed software verification model relies on the use of software invariants. Although this approach has certain drawbacks, such as the requirement to determine the physical dimensions and orientation of variables during compile time and increased compilation time, it still provides significant advantages over human manual error detection. TL empowers compilers to efficiently identify errors, making it a valuable tool in software development.

On the other hand, the proposed model has several notable advantages. It enhances programmer productivity by eliminating the need to troubleshoot dimensional and orientational errors during runtime. TL enables comprehensive analysis of the software's dimensional and orientational correctness, covering the compile and runtime phases. It ensures the correct usage of software variables and operations and verifies the arguments of functions and procedures.

The proposed TL seamlessly integrates with any modern C++ compilers, enabling formal software verification at compile-time. It enhances software reliability by introducing additional checks during dynamic linking and facilitates real-time formal verification.

Although the proposed method for formal verification is not customized explicitly for cyber-physical objects and systems, as its primary emphasis lies on software-level considerations, its adaptability shines when verifying general-purpose software that integrates a spectrum of physical parameters. This versatility traverses many domains, including educational, gaming, simulation software, and beyond, demonstrating its broad applicability across diverse industries and applications.

The effectiveness of the proposed TL was demonstrated through the analysis of real-world software for unscrewed aerial vehicles (drones) on GitHub. It successfully detected 90% of incorrect uses of software variables and over 50% of incorrect operations, resulting in an overall conditional probability of defect detection of 60% [17]. Because using TL extends the executable file size

by 1-3 MB, further research is necessary to explore methods for reducing this memory extension.

These results highlight the efficacy of the proposed software verification model in identifying software defects and reinforcing software reliability.

5. Directions for further research

Overall, the proposed C++ type library demonstrates a high detection rate, potentially reducing testing time and improving reliability and software quality. This approach is effective for formal verification during compile time and supplementary verification in real-time scenarios. The library shows promise in enhancing the reliability of custom software; however, further research and the development of additional methods are necessary to comprehensively evaluate the reliability of custom software. Furthermore, additional research is required to explore strategies for reducing memory usage.

Contributions of authors: conceptualization – **Yuriy Manzhos;** methodology, software, validation, formal analysis, resource – **Yuriy Manzhos, Yevheniia Sokolova;** data curation – **Yuriy Manzhos;** writing – original draft preparation, writing – review and editing, visualization **– Yuriy Manzhos, Yevheniia Sokolova;** supervision – **Yuriy Manzhos;** project administration – **Yevheniia Sokolova.**

Conflicts of interest

The authors declare no conflict of interest.

Financing

This study received no external funding.

Data availability

Data will be made available upon reasonable request

Use of Artificial Intelligence

The authors confirm that they used artificial intelligence technologies solely to check the grammar of the English text.

All authors have read and agreed to the published version of this manuscript.

References

1. Vasylenko, O., Ivchenko, S., & Snizhnoi, H. Design of information and measurement systems within the Industry 4.0 paradigm. *Radioelectronic and Computer Systems*, 2023, no. 1, pp. 45-54. DOI: 10.32620/reks.2023.1.04.

2. Valette, E., El-Haouzi, H. B., & Demesure, G. Industry 5.0 and its technologies: A systematic literature review upon the human place into IoT - and CPS-based industrial systems. *Computers & Industrial Engineering*, 2023, vol. 184. DOI: 10.1016/j.cie.2023.109426.

3. Schneider, G. F., Wicaksono, H., & Ovtcharova, J. Virtual engineering of cyber-physical automation systems: The case of control logic. *Advanced engineering informatics*, 2019, no. 39, pp. 127-143. DOI: 10.1016/j.aei.2018.11.009.

4. Manzos, Y., & Sokolova, Y. The method of data compression in Internet of Things communication. *Radioelectronic and Computer Systems*, 2020, no. 4, pp. 57-67. DOI:10.32620/reks.2020.4.05 (In Ukrainian)

5. Olaniyi, O., Okunleye, O. J., & Olabanji, S. O. Advancing Data-Driven Decision-Making in Smart Cities through Big Data Analytics: A Comprehensive Review of Existing Literature. *Current Journal of Applied Science and Technology*, 2023, vol. 42, iss. 25, pp. 10-18. DOI: 10.9734/CJAST/2023/v42i254181.

6. Maskuriy, R., Selamat, A., Ali, K. N., Maresova, P., & Krejcar, O. Industry 4.0 for the Construction Industry – How Ready Is the Industry? *Applied Sciences*, 2019, vol. 9, no.14, article no. 2819. DOI: 10.3390/app9142819.

7. Miśkiewicz, R., & Wolniak, R. Practical Application of the Industry 4.0 Concept in a Steel Company. Sustainability, 2020, vol. 12, no. 14, article no. 5776. DOI: 10.3390/su12145776.

8. Mane, V. Environmental Monitoring Using Internet of Things. *International Journal of Electrical and Computer Engineering*, 2022, vol. 11, iss. 1, pp. 2-9. DOI: 10.15662/IJAREEIE.2022.1101015.

9. Rayan, R. A., Tsagkaris, C., & Iryna, R. B. The Internet of Things for Healthcare: Applications, Selected Cases and Challenges. *IoT in Healthcare and Ambient Assisted Living. Studies in Computational Intelligence*, 2021, vol. 933, pp. 1-15. DOI: 10.1007/978-981-15-9897-5_1.

10. Strelkina, A. Information technology for dependability assessment and providing of healthcare IoT systems. *Radioelectronic and Computer Systems*, 2019, no. 3, pp. 48-54. DOI:10.32620/reks.2019.3.05. (In Ukrainian).

11. Syed, A. S., Sierra-Sosa, D., Kumar, A., & Elmaghraby, A. IoT in Smart Cities: A Survey of Technologies, Practices and Challenges. *Smart Cities*, 2021, no. 4(2), pp. 429-475. DOI: 10.3390/smartcities4020024.

12. Deepak, V., Mishra, A., & Mishra, K. Role of IOT in introducing Smart Agriculture. International Research. *Journal of Engineering and Technology (IRJET)*, 2022, no. 9, pp. 883-887.

13. Mourtzis, D., Vlachou, K., Dimitrakopoulos, G., & Zogopoulos, V. Cyber-Physical Systems and Education 4.0 – The Teaching Factory 4.0 Concept. *Procedia*

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

Manufacturing, 2018, no. 23, pp. 129-134. DOI: 10.1016/j.promfg.2018.04.005.

14. Tsiatsis, V., Karnouskos, S., Höller, J., Boyle, D., & Mulligan, C. Chapter 16 - Autonomous Vehicles and Systems of Cyber-Physical Systems. *Internet of Things*, 2019, pp. 299-305. DOI: 10.1016/B978-0-12-814435-0.00029-8.

15. Alsulami, A. A., Abu Al-Haija, Q., Alturki, B., Alqahtani, A., & Alsini, R. Security Strategy for Autonomous Vehicle Cyber-Physical Systems Using Transfer Learning. *Journal of Cloud Computing*, 2023, vol. 12, article no. 181. DOI: 10.21203/rs.3.rs-2301648/v1.

16. Banerjee, A., Maity, A., Gupta, S. K., & Lamrani, I. Statistical Conformance Checking of Aviation Cyber-Physical Systems by Mining Physics Guided Models. *Proceedings of the 2023 Aerospace Conference*, Big Sky, MT, USA, IEEE, 2023, pp. 1-8, DOI: 10.1109/AERO55745.2023.10115613.

17. Manzhos, Y., & Sokolova, Y. A Method of IoT Information Compression. *International Journal of Computing*, 2022, vol. 21, iss. 1, pp. 100-110. DOI: 10.47839/ijc.21.1.2523.

18. Fursov, I., Yamkovyi, K., & Shmatko, O. Smart Grid and wind generators: an overview of cyber threats and vulnerabilities of power supply networks. *Radioelectronic and Computer Systems*, 2022, no. 4, pp. 50-63. DOI: 10.32620/reks.2022.4.04.

19. Smadi, A. A., Ajao, B. T., Johnson, B. K., Lei, H., Chakhchoukh, Y., & Abu Al-Haija, Q. A Comprehensive Survey on Cyber-Physical Smart Grid Testbed Architectures: Requirements and Challenges. *Electronics*, 2021, vol. 10, no. 9, article no. 1043. DOI: 10.3390/electronics10091043.

20. Kang, B., Seo, K.-M., & Kim, T. G. Model-Based Design of Defense Cyber-Physical Systems to Analyze Mission Effectiveness and Network Performance. *IEEE Access*, 2019, vol. 7, no. 1, pp. 42063-42080. DOI: 10.1109/ACCESS.2019.2907566.

21. Wisniewski, R., Bazydło, G., Szcześniak, P., Grobelna, I., & Wojnakowski, M. Design and Verification of Cyber-Physical Systems Specified by Petri Nets – A Case Study of a Direct Matrix Converter. *Mathematics*, 2019, vol. 7, no. 9, article no. 812. DOI: 10.3390/math7090812.

22. Cordeiro, L. C., de Lima Filho, E. B., & Bessa, I. V. Survey on automated symbolic verification and its application for synthesizing cyber-physical systems. *IET Cyber-Physical Systems: Theory & Applications*, 2019, vol. 5, iss. 1, pp. 1-24. DOI: 10.1049/iet-cps.2018.5006.

23. Grobelna, I., Wiśniewski, R., & Wojnakowski, M. Specification of Cyber-Physical Systems with the Application of Interpreted Nets. *Proceedings of the IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society*, Lisbon, Portugal, IEEE, 2019, pp. 5887-5891. DOI: 10.1109/IECON.2019.8926908.

24. Luckeneder, C., & Kaindl, H. A case study of systematic top-down design of cyber-physical models with integrated validation and formal verification. *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19). Association for Computing Machinery*, New York, NY, USA, 2019, pp. 1828-1836. DOI: 10.1145/3297280.3297460.

25. Bernardeschi, C., Domenici, A., & Saponara, S. Formal Verification in the Loop to Enhance Verification of Safety-Critical Cyber-physical Systems. *Electronic Communications of the EASST*, 2019, vol. 77, pp. 1–9. DOI: 10.14279/tuj.eceasst.77.1106.1050.

26. Misson, H. A., Gonçalves F. S., & Becker, L. B. Applying Integrated Formal Methods on CPS Design, *Proceedings of the IX Brazilian Symposium on Computing Systems Engineering (SBESC)*, Natal, Brazil, 2019, pp. 1-8. DOI: 10.1109/SBESC49506.2019.9046084.

27. Grobelna, I. Formal Verification of Control Modules in Cyber-Physical Systems. *Sensors*, 2020, vol. 20, no.18, article no. 5154. DOI: 10.3390/s20185154.

28. Garro, A., Vaccaro, V., Dutré, S., & Stegen, J. Cyber-Physical Systems engineering: model-based solutions. *Proceedings of the SummerSim-SCSC 2019*, Berlin, Germany, 2019, Society for Modeling and Simulation International (SCS). Available at: https://scs.org/wp-content/uploads/2020/02/CYBER-PHYSICAL-SYSTEMS-ENGINEERING-MODEL-BASED-SOLUTIONS.pdf (accessed 24 July 2019).

29. Wisniewski, R., Bazydło, G., Szcześniak, P., Grobelna, I., & Wojnakowski, M. Design and Verification of Cyber-Physical Systems Specified by Petri Nets – A Case Study of a Direct Matrix Converter. *Mathematics*, 2019, vol. 7, no. 9, article no. 812. DOI: 10.3390/math7090812.

30. Nikolakis, N., Maratos, V., & Makris, S. A Cyber-Physical System (CPS) approach for safe human-robot collaboration in a shared workplace. *Robotics and Computer-Integrated Manufacturing*, 2019, vol. 56, pp. 233-243. DOI: 10.1016/j.rcim.2018.10.003.

31. The Incredible Story of the Gimli Glider. *Simple Flying*. Available at: https://thedailywtf.com/articles/the-therac-25-incident (accessed 6 August 2023).

32. *The Patriot Missile Failure*. Available at: https://www-users.cse.umn.edu/~arnold/disasters/patriot.html (accessed 23 August 2000).

33. Stephenson, A. G., LaPiana, L. S.; Mulville, D. R., Rutledge, P. J., Bauer, F. H., Folta, D., Dukeman, G. A., Sackheim, R., & Norvig, P. *Mars Climate Orbiter Mishap Investigation Board Phase I Report NASA*. Available at: chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf. (accessed 10 November 1999).

34. Hall, B. Software representation of measured physical quantities. Series on *Advanced in Mathematics*

for Applied Sciences. Advanced Mathematical and Computational Tools in Metrology and Testing XII, 2021, vol. 90, pp. 273-284. DOI: 10.1142/9789811242380_0016.

35. Manzhos, Y., & Sokolova, Y. A Software Verification Method for the Internet of Things and Cyber-Physical Systems. *Computation*, 2023, no. 11 (7), article no. 135. DOI: 10.3390/computation11070135.

36. Schabel, M. C., & Watanabe, S. *Chapter 42. Boost.Units 1.1.0*. Available at: https://www.boost.org/doc/libs/1_83_0/doc/html/boost_units.html. (accessed 17 August 2003).

37. *Benri is a C++ library for compile time checking of physical quantities.* Available at: https://github.com/jansende/benri (accessed 4 October 2019).

38. Pusz, M. *A C++ Approach to Physical Units.* Available at: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1935r2.html#biblio-nic_units (accessed 13 January 2020).

39. Moene, M., Huebl, A., Reinhold, S., & Pilz, T. *PhysUnits-CT-Cpp11 (compile time).* Available at: https://github.com/martinmoene/PhysUnits-CT-Cpp11. (accessed 24 May 2020).

40. *Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.* Available at: https://eigen.tuxfamily.org/index.php?title=Main_Page. (accessed 18 August 2021).

41. *OpenGl Mathematics (GLM).* Available at: Available at: https://github.com/g-truc/glm (accessed 13 April 2020).

42. Weige, M. *Quaternion Library for C.* Available at: https://github.com/MartinWeigel/Quaternion (accessed 16 May 2022).

43. Mahoney, J. F. Dimensional Analysis. *Procedia Manufacturing*, 2019, vol. 38, pp. 694-701. DOI: 10.1016/j.promfg.2020.01.094.

44. McKeever, S. Unit of measurement libraries, their popularity and suitability. *Software: Practice and Experience*, 2021, vol. 51, iss. 4, pp. 711-734. DOI: 10.1002/spe.2926.

45. McKeever, S. Acknowledging Implementation Trade-Offs When Developing with Units of Measurement. *Communications in Computer and Information Science*, 2023, vol. 1708, pp. 25-47. DOI: 10.1007/978-3-031-38821-7_2.

46. Siano, D. B. Orientational Analysis – A Supplement to Dimensional Analysis. *Journal of the Franklin Institute*, 1985, vol. 320, iss. 6, pp. 267-283. DOI: 10.1016/0016-0032(85)90031-6.

47. Siano, D. B. Orientational analysis, tensor analysis and the group properties of the SI supplementary units. *Journal of the Franklin Institute*, 1985, vol. 320, iss. 6, pp. 285-302. DOI: 10.1016/0016-0032(85)90032-8.

48. Dos Santos, L. F. Orientational Analysis of the Vesic's Bearing Capacity of Shallow Foundations. *Soils Rocks*, 2020, vol. 43, pp. 3-9. DOI: 10.28927/SR.431003.

49. Sutter, H. *Metaclasses: Generative C++*. Available at: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0707r3.pdf. (accessed 11 February 2018).

50. Sutton, A. *Metaprogramming.* Available at: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2237r0.pdf. (accessed 15 October 2020).

51. *Working Draft, Standard for Programming Language C++*. Available at: https://isocpp.org/files/papers/N4928.pdf. (accessed 22 May 2023).

Received 23.10.2023, Accepted 20.02.2024

СИСТЕМА ТИПІВ ДЛЯ ФОРМАЛЬНОЇ ВЕРИФІКАЦІЇ С/С++ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КІБЕРФІЗИЧНИХ СИСТЕМ

Юрій Манжос, Євгенія Соколова

Некоректне використання одиниць вимірювання та орієнтації у програмах С/С++ для кіберфізичних систем часто призводить до критичних помилок, які звичайні системи типів не можуть ефективно запобігти. Ручний аналіз коду за допомогою розмірного та орієнтаційного аналізу може виявити ці помилки в фізичних рівняннях, але при роботі зі складними фізичними обчисленнями, ці методи стають непрактичними. Ми запропонували підхід, який базується на використанні фізичних величин, визначених Міжнародною системою одиниць та операціями з орієнтацією фізичних об'єктів, як це запропонував Сіано. Цей підхід забезпечує однорідність одиниць. Додатково, включення операцій з орієнтацією в модель програмування є важливим для точного управління обертанням і вирівнюванням фізичних об'єктів. Практичні рекомендації, надані роботою Сіано дозволяють точно маніпулювати орієнтацією об'єктів, зменшуючи ймовірність помилок, пов'язаних з орієнтацією. Шляхом перевірки фізичних розмірностей і орієнтацій на етапі компіляції, потенційні дефекти програмного забезпечення виявляються до виконання коду. Це зменшує час налагодження та знижує витрати на виправлення проблем на пізніших етапах розроблення. Запропонована система типів, яка включає в себе розмірний та орієнтаційний аналіз, а також методи метапрограмування, представляє собою важливий крок у напрямку більш безпечних та надійних кіберфізичних систем. Цей підхід дозволяє виявити приблизно 90%

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

неправильного використання змінних програми та понад 50% помилкових операцій як під час компіляції, так і під час виконання великомасштабних програм в реальних умовах. Запропонований метод формальної верифікації не спеціально адаптований для кіберфізичних об'єктів та систем, враховуючи його основну увагу до проблем на рівні програмного забезпечення, він демонструє адаптивність для перевірки загальнопризначеного програмного забезпечення, яке включає різноманітні фізичні параметри. Ця універсальність поширюється на різні сфери, такі як освітнє, ігрове та симуляційне програмне забезпечення, серед інших.

Ключові слова: формальна верифікація; аналіз розмірностей; орієнтаційний аналіз; система типів; кіберфізичні системи; якість програмного забезпечення.

Манжос Юрій Семенович – канд. техн. наук, доц., доц. каф. інженерії програмного забезпечення, Національний аерокосмічний університет ім. М. Є. Жуковського «Харківський Авіаційний Інститут», Харків, Україна.

Соколова Євгенія Віталіївна – канд. техн. наук, доц., доц. каф. інженерії програмного забезпечення, Національний аерокосмічний університет ім. М. Є. Жуковського «Харківський Авіаційний Інститут», Харків, Україна.

Yuriy Manzhos – Ph.D. in Information Technologies, Associate Professor at the Department of Software Engineering and Business, National Aerospace University “Kharkiv Aviation Institute”, Kharkiv, Ukraine,
e-mail: y.manzhos@khai.edu, ORCID: 0000-0002-4910-7285.

Yevheniia Sokolova – Ph.D. in Information Technologies, Associate Professor at the Department of Software Engineering and Business, National Aerospace University “Kharkiv Aviation Institute”, Kharkiv, Ukraine,
e-mail: y.sokolova@khai.edu, ORCID: 0000-0002-1497-4987.