The Physical Semantics of Code

by

Hannah Leung

Submitted to the School of Engineering and Applied Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

UNIVERSITY OF VIRGINIA

May 2019

Author .....
School of Engineering and Applied Science

Certified by .....
Kevin J. Sullivan
Associate Professor
Thesis Supervisor

Accepted by .....
Sebastian G. Elbaum
Chairman, Department Committee on Graduate Theses

The Physical Semantics of Code

by

Hannah Leung

Submitted to the School of Engineering and Applied Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Numerous mishaps involving cyber-physical systems have occurred due to divergences between software representations of the physical world and the actual phenomena being represented. An example is the failure of the Mars Polar Orbiter mission due to the fact that a physical *impulse* was calculated in two different unit systems: imperial and metric[1]. This error manifested the problem that a complete and explicit representation of physical quantities is often stripped from code. The problem in such cases is that software is over-abstracted and thus becomes under-constrained in its behavior. Such systems can perform operations without type errors that have no physical meaning, leading to the production of values that can cause major malfunctions. Modern compilers provide type-checking, but *physical type errors*, by which we mean operations that are inconsistent *in the world represented by the software*, elude ordinary type checking. This thesis contributes to recent work on imbuing software with physical semantics through the provision of separate, formal, computable *interpretations* that map terms in code to enriched entities in the language of mathematical physics. Unique to our approach is the use of a higher-order constructive logic proof assistant to formally represent the physical domain. We present an initial proof of concept system that mostly automatically constructs interpretations by mapping terms in C++ code that represent physical quantities to enriched terms expressed in the language of the Lean Prover. Type checking in this domain then reveals physical type errors. A human analyst provides information needed to complete this mapping. In our work to date, this information specifies distinct vector spaces to which different vectors represented in the code are assumed to belong. Lean's type checker then detects undefined operations involving vectors in different spaces. By augmenting cyber-physical systems code with physical semantics, we show that it is possible to detect physical type errors that cannot be detected by ordinary type checking. To help develop and to evaluate our concepts we developed a prototype system that finds and constructs interpretations of vector values and operations in C++ code.

Thesis Supervisor: Kevin J. Sullivan
Title: Associate Professor

Acknowledgments

I would like to give thanks to a number of people who helped me during my graduate studies at the University of Virginia. Without them, it would not have been the success of my completion on the Master's degree.

First, I want to give my appreciation and high respect to Professor **Kevin J. Sullivan**, who introduced me to the wonderful world of software logic and enriched my understanding in so many aspects as well as in depth. I'm also grateful for his generosity, kindness as well as his encouragement, for he spent lots of time mentoring me and witnessed my growth over the years.

Secondly, I want to thank my family members who gave me lots of love and support to help me achieve my dream and for all the warm hugs and big kisses that kept me going.

Also, I want to thank to Professor **Elbaum** who offered valuable suggestions and guidance on my project and this work. I also appreciate his work organizing the Software Engineering Reading Group, which also benefited me greatly. Professor **Davidson** also provided expertise on many aspects that are crucial to the future work of this project.

Last but not least, I want to thank all my friends who kept my company during my studies and research, as well as the staff members at the **University of Virginia**. Without their help, things could never have worked so smoothly.

Contents

1 Introduction **11**

2 Prior Work and Status Quo **15**

2.1 External Checkers: Real-World Types 15

2.2 Type-System Approach 17

2.3 Units and Dimensional Analysis 19

3 The Problem Redefined **21**

3.1 Understanding Physical Abstractions in ROS 22

3.1.1 Observations of Over-Abstracted Code in ROS 22

3.1.2 Potential Deficiencies 25

3.2 Problem Generalized 27

4 System Architecture **29**

4.1 Technical Documentation of Peirce 30

4.2 modularity design 40

5 Case Study on Concrete code **41**

6 Discussion and Future Work **51**

6.1 Discussion 51

6.1.1 A Physical Semantics of Code 51

6.1.2 Design Trade-offs 53

6.2 Future Work 55

6.2.1 Design to be refined 55
6.3 Conclusion 57

A The API design across system **59**

B Sample Concrete Code **61**

List of Tables

Chapter 1

Introduction

IoT and cyber-physical systems (CPS) software controls the behavior of physical objects in the world. Software embedded in such objects is often written with no guarantee of the absence of *physical type errors*, by which we mean computations that are inconsistent with the mathematical physics of the domain. An example would be the addition of quantities in the code that represent physically incompatible quantities in the real world. Therefore, it will be valuable to develop what we call a *physical semantics* of code so that software that can be checked mechanically for consistency with respect to the mathematical physics of the physical domain.

Multiple projects have addressed special cases of this issue with different mindsets and methodologies. We briefly outline related approaches here, and briefly summarize our approach in relation to earlier work.

Type System-Based. One approach to ensuring consistency of computation with the physics of the world is to use the type systems provided by production programming languages to more fully represent physical types in code. So, for example, as DeRose[2] showed, one can provide an abstract data type that fully represents all relevant aspects of multiple vector, affine, or Euclidean spaces, leaving it to the native type system to detect inconsistent operations, such as attempts to add two points in an affine space. One relies on the programming language type checker to prevent physical type errors.

This is a sensible approach, but in practice, it is often not used and a great deal of important code does not use such an approach. The Robot Operating Systems, ROS [3], for example, until recently, represented vectors as simple 3-tuples of floating point numbers. Now it also provides a way to represent frames of reference in terms of which coordinates are expressed, but it still has still no way to represent different vector spaces altogether. And while it also provides a way to represent points, it provides no fully developed representation of affine or Euclidean spaces, including points, vectors, frames, inner products, and all their relevant objects and operations.

Language-Based. A second approach to enforcing the physical consistency of code is to embed physical concepts, such as physical units (meters, feet, kilograms, etc), into the programming languages that developers use to program hardware systems.

For example, the F# programming language allows one to associate physical units with unsigned integer and floating point scalar values and variables. A problem with this approach is that the types of objects relevant to the physics of the domain are much richer than just units for scalar values, and it does not seem reasonable or practical to fold rich abstractions, such as those for vector, affine, Euclidean, and others spaces directly into the programming language. Yet, to describe the flying behavior of a plane, one must represent such quantities as locations, velocities, and angles in different 3-D coordinate systems. This will require much more than 1-D physical units annotations.

External Checkers. A third approach to enforcing physical consistency of code is to design additional pieces of software, ones that run outside of the main compiler, to checks all the entities used for describing and computing states within a program. Much work has been done on *pluggable type systems*[4], for example, to enable stronger type checking than a native type system supports. Such work, however, has not focused heavily on checking with respect to the intended real-world interpretations of terms in the code, but has rather focuses on lower-level issues internal to the code, such as properties of pointers being non-null, for example.

Physical Annotation Methods. The final approach we discuss here involves the annotation of code with information about physical interpretations, combined with the use of external checkers to detect inconsistencies. Much work on annotating code with physical units is of this kind, including recent work by Elbaum et al., for example [5].

In human subjects studies, Elbaum et. al. demonstrated the difficulty of accurate, cost-effective, human annotation of code with physical units. They addressed that problem with an annotation inference mechanism that propagated annotations using program flow analysis techniques. But that work did not generalize to the rich physics of Euclidean and other spaces.

Xiang, Knight, and Sullivan [6] showed that it was possible to associate an explicit interpretations with code, linking terms in code to so-called *real world types*, thereby enabling physical type checking with respect to frames of reference and other physically important quantities. However, that work took a simple, ad hoc approach to represent physical types.

Our Approach. Given the design constraints and the goal we aim to achieve, we propose a different approach: one that, at its core, formalizes rich mathematical models of physical domains embedded in the constructive logic of a modern proof assistant, and that then maps terms in code to meanings in such domains. We are currently using the logic of the Lean Prover [7], but Coq [8] or another such system would serve just as well.

The higher-order logics these languages are of deep interest not only to software engineers and programming language designers and implementors but to mathematicians. These logics are expressive enough to represent rich structures in many branches of mathematics, and they promise to enable the mechanized checking of theorems in many fields. Libraries in Lean have been written, for example, for topology, field theory, category theory, etc. These are mathematical realms in which it is feasible to abstractly and concisely represent rich, diverse concepts in mathematical physics.

Outline. In Chapter 2, we present a survey of the research state of the art in the broader field of real-world type checking and the physical consistency of code. We discuss how each of the related projects and design philosophies influenced our work. In Chapter 3, we define the boundary and the scope of the problem we are trying to tackle and the kind of solution we could expect using the approach proposed in this work. In Chapter 4, we discuss our proposed approach in greater detail, from the architecture of our demonstration system to detailed design preferences and implementation strategies. In Chapter 5, we present an initial proof of concept system and show how specific pieces of C++ code that manipulate a robot can be given a physical semantics and can be checked for consistency with the mathematical physics of the domain. In Chapter 6, we discuss design tradeoffs, our conclusions, and future work.

Chapter 2

Prior Work and Status Quo

In this chapter, we survey the relevant work in this field and report on research directions that have previously been explored. Each piece of work has inspired our work from a different perspective.

2.1 External Checkers: Real-World Types

The paper, *Synthesis of Logic Interpretations*, by Xiang et al. [9], pointed out that an explicit structure for documenting real-world interpretations of the code is essential. It also pointed out that there is a gap between the semantics of high-level languages and the interpretation of programs built on top of the high-level languages, where the latter attempt to model and ultimately control the behaviors of objects in the real world. Finally, Xiang et al. note that an intended interpretation of a software system is always present, at least in the minds of the developers, but it is not always either well conceived or documented in a manner that would allow it to be used for systematic consistency checking of code.

Often, these *intended interpretations* reside in descriptive comments, variable names, in other relevant identifiers, or in other documentation. Such information is of great importance. When software systems control objects in the real world, the scenario is often continuous and demanding in terms of control. Therefore, it is crucial for programmers to be clear about the laws that program executions must

obey. However, when the intended real-world interpretations of code elements are not precisely and completely documented in ways that can be mechanically analyzed, it is challenging for the programmer to ensure that such constraints are always satisfied. This is where additional computations could help the programmer to ensure the satisfaction of such additional physical world constraints.

The work of Xiang et al. supports programmer-assisted construction of interpretations for cyber-physical systems code. The idea is to add information about physical types represented by *machine types and values* to enable mechanical checking of machine-level operations for real-world consistency. A key question is *who or what provides the required additional information?*

Xiang et al. provided two answers. Their first demonstration system design supported interactive human annotation of code by linking variables and values in code to records representing corresponding *real-world types*. These types came with verysimple type signatures for the real-world operations that they support. An analyzer performs limited type inference to propagate real-world type annotations through the code, e.g., through assignment operations. It then determines whether certain operations in the code are unsupported at the level of the corresponding real-world types. If so, it issues a real-world type error. The work supports physical unit annotations as well as annotation of variables with limited richer types, including value ranges and coordinate frames of reference.

In follow-on work [6], Xiang et al. presented a method for easing the human code annotation burden by heuristically inferring candidate real-world types from identifier names. The system then presented the human analyst with possible interpretations of each relevant piece of code, leaving it to the human to decide which, if any, to use.

Xiang et al. carried out experimental work [6], using the *Kelpie Flight Planner* software [10] to test the proposition that the real-world type checking approach could find subtle real-world type errors in real cyber-physical systems code. They obtained positive empirical results. Information from this work provided a nice introduction to and overview of current progress in this field, and it inspired many of the concepts and design decisions in this work.

The overall thesis of the work of Xiang et al. is that software for cyber-physical systems comes today without checkable interpretations and is thus prone to contain physical type errors with potentially catastrophic implications. In their paper, [6], they propose that the lack of such interpretations is a fundamental shortcoming in the engineering of software for cyber-physical systems and that, in the future, what software engineers must deliver are pairs: ordinary programs, but now accompanied by explicit, checkable interpretations.

2.2 Type-System Approach

There are many packages and libraries designed for specific fields of physical computation. One library that influenced our thinking was a C++ library by Mann and DeRose *et al.*[2] developed for coordinate-free affine geometry for use in the design and implementation of computer graphics code.

The problem that they addressed was substantially similar to that which we are addressing. The issue was that programmers often wrote their code in terms of types over-abstracted from the mathematical geometry of the domain of physical space in which objects exist. They gave a simple example that showed there could be three distinct interpretations of a two-dimensional array: a transformation of a space, a change of coordinates for a space without a transformation, or as a mapping between spaces. Similarly, one-dimensional arrays represent either points or vectors but the distinction was often not explicit in the code. The lack of sufficient type information in the code made it too easy to make what we now call physical type errors.

This phenomenon entails the same problem discussed earlier in the programming world. Not all of the operations that are valid, as far as the machine-level type systems is concerned, have valid meanings in the domain of affine and Euclidean spaces. Often, it is the case that the links or connections from code to intended meanings are completely missing from the code.

To address this problem Mann and DeRose *et al.* proposed a geometric abstract data type (ADT) providing a complete abstraction of mutiple affine and Euclidean

spaces. An affine space is a pair, $(\mathcal{P}, \mathcal{V})$, in which $\mathcal{P}$ is a set of **points** and $\mathcal{V}$ is a set of **vectors**, along with certain operations on these objects that satisfy certain axioms. A Euclidean space is an affine space with an inner product that establishes a measure for distances and angles. Vectors can be understood as translations between points. The operations defined for affine spaces include standard vector space operations as well as subtraction of one point from another yielding a vector, and addition of a vector to a point yielding a new, translated point. All of these abstractions were defined in a *coordinate-free*, or *abstract*, form. For the purpose of computing, numerical coordinates for objects were maintained in an encapsulated form.

The library incorporated the concept of multiple spaces, and of multiple coordinate systems on any given space. A coordinate system on an affine space is defined by an *affine frame*. A frame comprises a point in $\mathcal{P}$, the *origin* of the frame, and a set of basis vectors in $\mathcal{V}$, in terms of which coordinates for any other point or vector can be expressed. Objects in the same space can be assigned coordinates with respect to different frames. These coordinates are coefficients of linear combinations of basis vectors. Euclidean geometry is defined as affine geometry plus an inner product that gives rise to absolute distances, lengths, and angles, which are not available in affine spaces.

Using this approach, each object has a specific geometric meaning so as to prevent ambiguous computations later on. This approach was effective because, in addition to defining fields or attributes of concrete representations of geometric objects, it also defined and enforced abstract, geometric, meanings. The interpretations of the concrete representations were thus made clear and enforceable. This library ensured that every object that it could compute represents a valid object in a Euclidean geometric space. It prevented execution of geometrically invalid operations on concrete, floating-point tuples that concretely represent different geometric objects.

This approach brought the mathematical rules of the domain into the program using the mechanism of user-defined types and the native type checking facilities of the C++ language. This approach resolved the problem they had identified. It also provided inspiration for this thesis. In particular, the concept of coordinate-free, or

abstract, representations of the mathematical physics domain is one that we intend to pursue aggressively. In the case of DeRose's work, there are of course coordinates, but they are hidden behind abstract interfaces. The interface presented to the programmer is *abstract* in the mathematical sense that one can operate on points, vectors, measures, transformations, and spaces, as abstract objects without having to see or thing about their underlying concrete, coordinate-based representations. The objects of an affine space are points and vectors, not tuples of coordinates. Coordinate free is the style in which mathematicians and physicists generally prefer to reason except when implementing computations.

While the approach of Mann and DeRose was sound, the fact is that much code is written without such care for representing mathematical abstractions in their full richness in code.

2.3 Units and Dimensional Analysis

Units and dimensional analysis can be viewed as a special case of physical type correctness. Decades of work have addressed the problem of the lack of physical units annotations in code, and the consequent ease with which physically erroneous code can be written. Elbaum et al. [11] studied this problem particularly in the context of robotics software. They examined code written against the Robot Operating System (ROS) libraries[12]. In their work, they found that the inconsistency type 'Assigning multiple units to a variable' accounts for 75% of inconsistencies in ROS code. Their work and previous work was done for representing one-dimensional SI units. While Xiang et al. did annotate code with richer abstractions (namely frames of reference), the approach was ad hoc and did not use or seek to enable the definition of mathematical spaces (geometric, algebraic) in terms of which proper physical interpretations of code could be defined. Richer abstractions, such as those of complete Euclidean spaces, are needed for current and future cyber-physical systems. And these spaces go beyond the geometric and temporal to include mathematical spaces for representing thermodynamics, electrodynamics, gravity, and so on.

Chapter 3

The Problem Redefined

As discussed in `Chapter 2`, much work has been done to regularize the use of the Standard International Units in programs. However, there is inadequate support for the much richer physical types that will be needed for the engineering of software-driven cyber-physical systems, including robotic systems. In this thesis, we generalize the idea and explore the possibilities that leverages coordinate-free approaches that transcend the realm of one-dimensional physical units so that the consistency of code with much richer abstractions of the mathematics of the physical world can be mechanically checked.

While compilers for modern computer languages often have rich type systems, they cannot do type checking based on information that is not in the code. That information often includes the intended physical interpretations of well typed terms in the code. Often one can not ascertain from the code whether operations in the code are valid with respect to what they are to represent in the *intended physical domain of discourse.* The work we discuss in this thesis builds on earlier work to connect concrete terms in programming code with physical interpretations by linking them to abstract algebraic structures in the language of the mathematical physics of the real world.

We decompose the task into subsections. There are many key questions that need to be answered to develop and evaluate a proof of this concept. Below are **4** essential aspects.

1. For which code elements do we want to establish interpretations? How might we best detect code elements of interest?

2. To what mathematical physics abstractions do we want to link such code elements, and how can we do this efficiently?

3. How should we represent the required abstractions derived from the mathematical structure and the physical world?

4. How might we best compose the essential components to assure the consistency of software with the physics of the domain?

3.1 Understanding Physical Abstractions in ROS

As a study case, we investigated the ROS code base to understand current practices for representing physical quantities in code written against this widely used library.

3.1.1 Observations of Over-Abstracted Code in ROS

We investigated the `ROS common_msg` stack[12]. According to the documentation on ROS wiki[12], Common_msgs contains messages that are widely used by other ROS packages. These include messages for actions (actionlib_msgs), diagnostics (diagnostic_msgs), geometric primitives (geometry_msgs), robot navigation (nav_msgs), and common sensors (sensor_msgs), such as laser range finders, cameras, point clouds. In fact, not all robotic system incorporated sensors, therefore that package will not be used as much as the geometric primitives stack, as every system needs to locate and control the motions of robots. We now present our observations, especially for the geometric primitives, and the deficiencies of over-abstracted, thus under-constrained, representations.

Initially, ROS designed the common_msg stack to isolate the messages for communicating between stacks in a shared dependency. This allows nodes in separate stacks to communicate without direct dependencies upon each other. This stack has

been designed to contain the most common messages passed between other stacks. This shared dependency eliminates problematic circular dependencies[12]. Representations of many physical quantities, such as acceleration, position, velocity, inertia, and other often used physical quantities are defined in this library.

Consider the following example: the ROS code that defines that ROS representations of **Intertia** and **Point** abstractions[13].

Inertia representation

```
# Mass [kg]
float64 m

# Center of mass [m]
geometry_msgs/Vector3 com

# Inertia Tensor [kg-m^2]
#     | ixx ixy ixz |
# I = | ixy iyy iyz |
#     | ixz iyz izz |
float64 ixx
float64 ixy
float64 ixz
float64 iyy
float64 iyz
float64 izz
```

The code snippet above defines a representation for the concept of inertia. Inertia is a concept well established in physics. It is vital to consider when exerting fine control over objects with mass. In this specific example, the mass is represented using a *float64 value*. It is a sufficient representation in the sense that it can express the possible scalar values of mass. However, we know that the mass is always greater than or equal to zero. The type, float64, is over-abstracted from the concept it is used

to represent, and is thus under-constrained in the sense that it can take on values that have no physical meaning (except in exotic versions of physics).

The center of mass is similarly represented by an array of 3 float64 value. In fact, many physical quantities are represented in this way, including Point (position), Vector (translation), etc. The mapping from physical types to machine types is non-injective (many to one), and thus lossy. The intended interpretations of the representations are ambiguous unless specified by additional information external to these representations. Moreover, programming language type-correct operations can now be performed on the representations that have no physical meaning. An example would be like adding a position to another position, or a mass to temperature, or subtracting two masses to obtain a negative mass. Considering the ROS representations of **Points** and **Vectors**. Here are the relevant definitions in the ROS code.

Point representation

```
# This contains the position of a point in free space
float64 x
float64 y
float64 z
```

Vector representation

```
# This represents a vector in free space.
# It is only meant to represent a direction. Therefore, it does not
# make sense to apply a translation to it (e.g., when applying a
# generic rigid transformation to a Vector3, tf2 will only apply the
# rotation). If you want your data to be translatable too, use the
# geometry_msgs/Point message instead.

float64 x
float64 y
float64 z
```

While ROS does have lightweight abstractions for physical locations, directions, and other such quantities, the information required to ensure physical consistency of operations involving such objects is not fully represented in the code. The concrete representations do not support the semantics imposed by their intended interpretations. The mathematical physical constraints expressed in the comments cannot be fully checked by machine because there is no formal representation.

3.1.2 Potential Deficiencies

While these message types do impose a degree of abstraction, in the sense that there are different Vector and Point messages types, they are not sufficiently rich abstractions to enforce consistency of code with the physics of the domain. For example, these represetations provide no way to record what space a given vector belongs to, and it was only very recently that it became possible to record coordiante frames of reference for points and vectors in ROS.

Such under-constrained representations require programmers to do *mental bookkeeping* to keep track of the details of the physical quantities and properties that the code is meant to represent. Programmers themselves then enforce the mathematical abstractions that the concrete data are meant to encode. Either those constraints remaining in programmers' minds or are documented in the comments or by giving the variable meaningful names, but in either case, it is not possible to mechanically check that they are enforced. There is no way for the programming type system to detect the physically meaningless operations because the mathematical and physical abstractions that the code is meant to represent are not represented in a machine-checkable form.

Concrete cases where things could go wrong Before discussing the abstractions derived from mathematics and physical world, in cases where the spatial dimension of the world has the structure of a Euclidean space, we want to demonstrate the concept in the weaker abstraction of affine space. One can summarise the valid set of operations for scalars, points, and vectors as follows:

- point - point = vector
- vector + vector = vector
- vector * scalar = vector
- point + vector = point

Operations other than the ones listed above are simply not part of the algebra of affine space. There is no operation for adding points to points, for example. Of course, if points are represented only by arrays of float64 values, then it becomes possible in the code to do so.

VectorA = VectorB + VectorC It is definitely feasible to do so in the code because the `C++ compiler` will not complain about this operation applied on the object **VectorB** and **VectorC** and assign it to **VectorA**. Since they are all represented using 3 tuple floating numbers, the `C++ compiler` will not detect any abstract type errors like this. However, this is definitely considered an **ERROR** of the program logic, the program that does the simulation of the real-world scenarios, the program that controls the system that interacts in reality. However, this is just a simple example of how the constraints derived from the mathematical abstractions and physical world can be easily violated when mechanized checkable procedures are missing.

Such physical type errors are easy to introduce into systems and hard to detect due to the absence of mechanisms for enforcing the physical abstractions that the concrete code is meant to represent. There are other cases. One is the case of two vectors reside in different spaces. Another is of two vectors in the same space but where the coordinates are expressed in terms of a different frame of reference. We note that different physical units for the same physical quantity (such as the distance of mass) are simply a special case of mismatched coordinate frames. Meter and foot are different *basis vectors* for a vector space of spatial displacements. More generally, the origin points of different frames for an affine space can differ, as can the basis vectors of the underlying vector space.

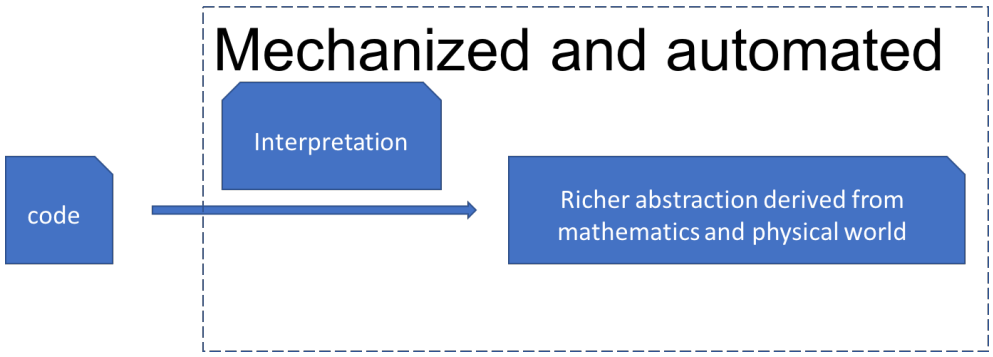

Figure 3-1: The code augmented with richer abstractions that provide more constraints

Operations on vectors represented in the same space but expressed in coordinates with respect to different frames should either be prevented, or change of basis operations must be applied so that coordinate-wise operations are valid. A problem related to different frames occurred during an attempt to dock Orbital's Cygnus spacecraft with the International Space Station (ISS)[14]. The two vehicles both represented GPS time in units of weeks plus seconds. Unfortunately, there were two standard frames for GPS time, with different origins, one in 1980, and one in 1999. The error was caught when the ISS checked time values transmitted by Cygnus. Unable to make sense of the data, the ISS aborted the docking attempt. The problem was not one of units inconsistency (weeks plus seconds). That is, the problem was not in mismatched bases for the vector space component of the affine space in which the vehicles were docking. It was in the fact that the two frames had inconsistent origins. While the units were the same, the coordinates were nevertheless not compatible. Clearly, units consistency checking alone is an insufficient safeguard.

3.2 Problem Generalized

As shown in `Figure3-1`, the code is simply the pure symbolic logic that lacks reliable mechanisms for abstract, or physical, type checking of the code. Therefore, **physical type errors** are easy to make and will go undetected by the compiler chance. Such an error occurs when operations are performed on data in ways that are inconsis-

tent with their intended physical interpretations. Clearly, even in so-called type-safe languages, such errors can easily occur and they will be impossible to detect mechanically. The type checker within the language is not specialized to check the validity of the operations applied, thus has no such capacity to capture the abstract errors. Critical machinery is missing in the current practice in which the code for cyber-physical systems is written.

Our aim is to provide software engineering concepts, tools, and methods for developing robotic application code with infrastructure to assist in constructing, representing, and exploiting physical interpretations of typical over-abstracted code. The abstractions to be checked come from the domain of discourse. An interpretation maps concrete code elements with their intended mathematical-physical meanings in the domain. An interpretation is the crucial missing ingredient needed to establish a checkable physical semantics for code. In the next Chapter, we will demonstrate the architecture of a system that implements such infrastructure that supports the mechanized abstract type checking with respect to such abstractions.

Chapter 4

System Architecture

We named this project after the American Logician, Charles Saunders Peirce (pronounced *purse*)[1]. Peirce studied and contributed to many fields. One was the field of semiotics, a general theory of signs and their meanings. We consider this name proper because, besides logic and philosophy, Peirce wrote voluminously on a wide range of other topics, as well, ranging from mathematics and mathematical logic to physics, geodesy, spectroscopy, and astronomy. Computer programs inhabit the realm of logical symbols, while all these other fields exist in the real-world. Peirce had the idea of connecting these worlds. Our project similarly aims to connect the symbols of the logical world of software to the physical world in which those symbols have their intended meanings. Making such connections explicit, precise, and computable is a key to ensuring that software logic and the behavior it specifies is true to real-world facts and laws. An interesting fact is that the initials of Charles Saunders Peirce (CSP) is the reverse of the abbreviation of the title of this thesis, the Physical Semantics of Code (PSC). In this section, we present the overall software architecture of our system for making such connections.

[1] `https://plato.stanford.edu/entries/peirce/`

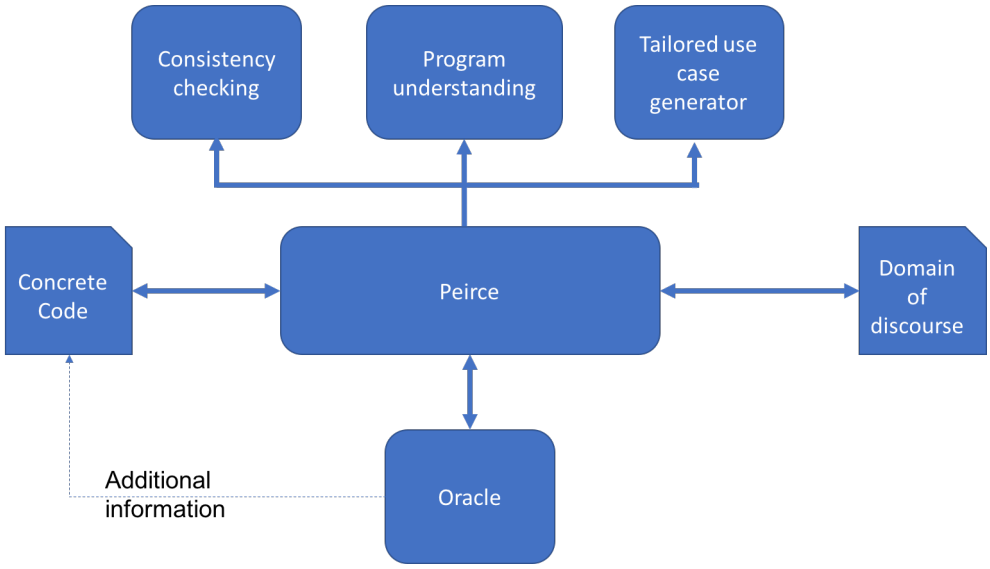

Figure 4-1: Peirce connects the over-abstracted code with the logic in the domain with information from the oracle. Peirce also provides service to the outside world.

4.1 Technical Documentation of Peirce

This section is designated to discuss the technical details of the Peirce System on how several important concepts that we introduced are implemented and why certain tools are selected to achieving the goal.

Figure 4-1 provides a blueprint illustrating how Peirce connects concepts expressed in concrete source code written in C++ with corresponding concepts in a domain of discourse, expressed in terms of domain abstractions embedded in the constructive logic of the Lean prover. We take terms in such a domain language to be mathematically complete representations corresponding directly to quantities in the physical world. For example, while a vector that represents an acceleration in with respect to a given frame of reference might be represented as little more than a three-tuple of floating point numbers in C++ code, it's corresponding representation in the domain language could represent the Euclidean space in which it is defined, the affine space in terms of which the Euclidean space is defined, with a frame on that space comprising a designated origin point and a basis for the corresponding vector space. The coordinates of the origin point and of the basis vectors could, in turn, be defined in terms of a different frame on the same space. In fact, a stack of frames corresponding

to a composition of change of basis functions is typical in such representations. The point is that what is represented efficiently but incompletely in code now has its full physical semantics expressed in the corresponding domain.

As should now be clear, critical information about the mathematical physics of physical quantities is often stripped in the translation from the world to code. To specify a physical semantics for such code, it is necessary to construct an interpretation and one that restores this missing information. Peirce must obtain this additional information from somewhere. There are many possible sources. To abstract from any particular choice, so as to allow a broad range of alternative future mechanisms, Pierce obtains it by calling an abstract *Oracle* module, from which it simply requests the missing information for any given code element. Peirce translates the code into expressions in the domain language, augmented with the additional information, and links the code element to its interpretation.

The main data structure that Peirce constructs for establishing such links is what we call an *interpretation*. In the current instantiation of the tool, it links each relevant *term* in the code (constant, variable, expression) to a corresponding term in the domain language. How Peirce builds this computable interpretation is by iteratively binding at different levels of an abstract syntax tree for the program source code based on a set of binding rules. Concretely, the parsing of C++ code and identification and dispatching of relevant code elements is done using functional programming-like pattern matching rules implemented using the *Clang Tooling* compiler-extension-building framework [15].

Once an interpretation is constructed, the checking phase of the process checks the interpretation for physical type errors. This phase depends on the definition of a set of types in the domain of mathematical physics and of operations involving these types, to which terms in the code are *lifted*, in such a way that the type checker of the Lean Prover will detect the errors of interest.

We assume that the programs to be checked have been developed carefully and that it is, therefore, feasible to determine which aspects of the code are intended to represent objects and operations in the real-world. In other words, we assume the

static analysis of the source code can determine which code elements require physical interpretations. This idea actually inspired the topic of this thesis. Since the set of assumed physical objects types and operations (e.g., point of vector in an affine space, and point-vector addition operations) are stable for a given application, therefore it is natural to formalize those types and the corresponding legal set of operations at the level of abstract geometry and in terms of laws derived from the physics of the control of flying objects or other physical systems.

A representation of the selected and augmented code is then rendered into a set of corresponding expressions in Lean in which the additional information is used to do stronger type checking, specifically to detect physical type errors in the code. Beyond just type *checking*, we also have short term plans to leverage Lean's *type inference* mechanism to minimize the number of queries to the human analyst that will be required to enable physical type checking. In future work, in order to make this process smarter, it is worthwhile to consider leveraging machine learning and related techniques ascertain appropriate interpretations without imposing an undue burden on the human developer, e.g., to find patterns that share high similarity and apply annotation propagation rules to the clusters found by machine learning algorithms.

The abstract *Oracle* component of our architecture can be implemented in many different ways, affording options to explore and adopt such alternative approaches. Eventually, we anticipate implementations that take advance of many sources of information, including programming source text, naming conventions within that domain, type inference mechanisms, analysis of reference materials, documentation, Artificial Intelligence, and many other related approaches. Such a structure could be used for synthesizing candidate real-world types, inferring real-world type bindings for program variables as well as synthesizing candidate type rules from verified or trusted programs.

Inside Peirce, there are several phases when the code element gets extracted, augmented, linked and transformed down to the logic expressed in Lean. For each of the phases, there exists a component designated to handle the transformation. Below is a detailed description of what each of the components is and what technique is

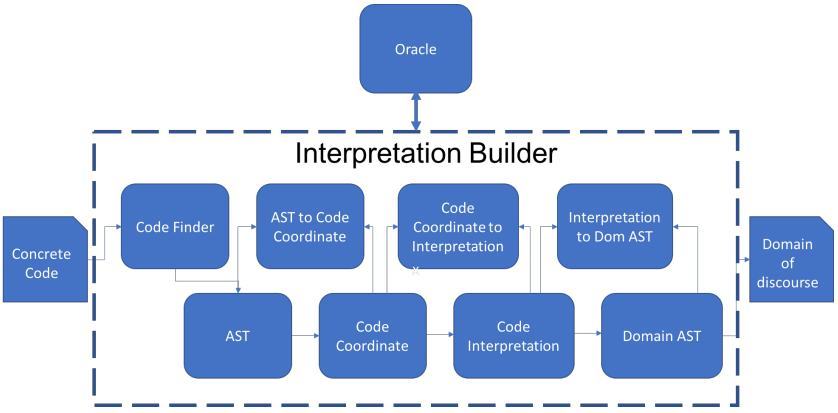

Figure 4-2: Caption

used and how they work together to achieve the goal of providing physical type error checking.

Code Finder In cyber-physical systems, the controlling software is often complex. Not all of the code represents or performs computations about the physical quantities that the system monitors, such as the velocity, location, acceleration and etc. We leverage the Clang / LLVM toolchain to find the code elements that require physical interpretation for a particular purpose. We used Clang *libtooling* as a library to develop our own tool to do static analysis on the source code that runs on cyber-physical systems.

Building a tool on a parsing mechanism other than that used by the compiler that will actually be used to compile the code could lead to some inaccuracies in the handling of the code. By contrast, Clang, on which we've built our implementation, is the same parser as that used by the LLVM compiler that we use to compile the C++ source code. This means there will be no false positives.

One of the main challenges we faced when developing our tool was to provide complete information to Clang to parse the source code for our purposes. Clang recently introduced its so-called *ASTMatcher* library to provide a simple, powerful, and concise way to describe specific patterns in the AST that should trigger actions by a tool such as ours. Implemented as a DSL and powered by macros and templates, matchers offer the feel of pattern matching with algebraic data types common to functional

programming languages[16]. Matcher expressions allow you to specify which parts of the AST are interesting for a certain task. In order to capture the code elements of interests, we designed ASTMatching patterns in that DSL to extract them precisely from the Abstract Syntax Tree parsed by Clang. In Chapter 5, we will give several concrete examples that demonstrate the power of the ASTMatcher in extracting both top-level and nested terms, the latter of which can require recursive matching and handling. Once an AST node of interest is found based on given matching patterns, a corresponding handler is called to perform the next phase. It can, in turn, execute pattern matching operations on subtrees of a given node.

In the current version of what Peirce is capable of, we define the following BNF grammar to describe the terms that Peirce can process. The experimental code that we test Peirce on is a simple vector library that contains the definition of Vector and one valid vector operation, which is 'vec_add' for adding two vectors.

```
-----------------------------
Vector_Def := VecIdent = VecExpr
VecIdent := VecVarExpr
VecVarExpr := string.
VecExpr := VecLitExpr | VecVarExpr | VecOpExpr | ParenExpr
Scalar := float
VecLitExpr := (Scalar, Scalar, Scalar)
ParenExpr := ( ParenExpr )
VecOpExpr := VecExpr.VecOp(VecExpr)
VecOp := "vec_add"
-----------------------------
```

At the high level, there exists the point when a new Vector is introduced into the system, then a definition of that Vector is formed. A vector definition is defined as the a vector identifier *VecIdent* assigned by a vector expression *VecExpr* and this vector identifier is a *VecVarExpr* and *VecVarExpr* is expressed as String. *VecEpxr* has several forms, such as the *VecLitExpr*, *VecVarExpr*, *VecVecAddExpr* and *ParenExpr*.

VecLitExpr is in the form of an array of 3 floats that represent the actual values. A *VecVarEpxr* represents the use of a variable, such as **v1** or **v2**, in an expression. *VecOpExpr* is defined as the binary operation of addition for vectors. It is expressed as two *VecExpr*s connected by the *VecOp* operation. In the current version of Peirce, we only support the add function, and we leave it in the future work to provide a full-fledged domain of discourse with complete, coordinate-free vector space operations (scalar multiplication is missing here), affine space concepts and operations (points, and point-vector operations are missing, as are representations of frames), and Euclidean geometry abstractions (we do not currently have ways to interpret terms in code as representing distances or angles; vectors are properly understood to represent differences between points).

Once the entities mentioned in the BNF grammar above were found, they are handled by the Interpretation module that fires up a sequence of actions to create the corresponding objects at different phases and build the connections between them that serve for searching and indexing purpose. More details will be included in the Interpretation section.

AST Once a node specified by our matching patterns is found, the Abstract Syntax Tree node is considered *raw*, as we only have the node in the representation in Clang AST hierarchy style. Moreover, additional relevant information about the node is stored in a separate Clang *ASTContext* data object. Information here includes variable names and source file, line, and column number locations. We need to use this additional information to provide the human analyst who will serve as an oracle with such information so that they know which code elements they are annotating. There is other information in the ASTContext node that is less relevant such as getting the value of refersToBitField(Returns true if this expression is a gl-value that potentially refers to a bit-field.) So we emitted that kind of information and coerced the ASTContext node only to provide information relevant to this project.

`AST.h` defines the alias for the AST node types in Clang naming convention to the naming in the domain of discourse, which are the terms that we used for describing

the BNF grammar. One concrete example for the alias of the VecIdent is given below.

```
using VecIdent = const clang::VarDecl;
```

Accordingly, we defined such alias out of engineering consideration and it is proven that such alias kept the Peirce system easy to maintain when it evolves along the way.

Code Coordinates This module wraps AST objects to abstract from the details of Clang's representation and to provide added behavior necessary and appropriate for each referenced code object. For example, code coordinate objects provide a uniform interface for obtaining source file names and line, and begin and end column numbers, for any code elements, along with a method for rendering any AST into a string suitable for presentation to a human analyst. Code coordinates objects also abstract from Clang. They provide for *ontology translation*, between the concrete Clang types used to represent pertinent code elements in a given programming language, and the abstract terms of a domain language, on which the rest of the interpretation pipeline is based. Here the code language is C++ as used to map applications built on our simple vector class (Vec). The domain language is one of simple vector space expressions and objects, albeit with gaps, as described above. We named it this way because this object provides enough information so it is easy to locate and identify the code element when needed.

We followed the Object Oriented Programming design principle to layout the structure of the Code Coordinate Objects. We define a superclass Coords that contains virtual functions to be implemented in each of its derived class to have different behaviors based on its own type. This superclass provides basic interfaces like deciding if two objects are equal or not base on the value of the given pointers, as the **coords** serves as the key in a two-way mapping from AST nodes to Code Coordinate objects. **Coords** class also provides interfaces for printing the current object to string(method *toString*) as well as getting the location where it appeared in the source location(method *getSourceLoc*). Both of these two methods are defined as

virtual functions as the derived classes require different implementations.

Interpretation This module is at the center location of the whole process. It coordinates between the entities in source code representation and how those entities are represented in the domain of discourse. It essentially builds mappings between *Coords* objects, element-wise *Interp* objects, and *Domain* Objects. Each *Interp* object is associated with a code *Coords* object on one hand, and with a *Domain* object, on the other. *Interp* objects link code and domain objects at the level of individual AST nodes. Because they have visibility to both (abstracted) code and the domain, they are able to perform functions such as rendering objects as strings that include both code and domain information.

Once all the entities in the source code get lifted to the domain of discourse, the Lean type checker will check the consistency of physical types. If there exists such physical type error detected by Lean, then it will be helpful to be able to trace back to the original code. This requires the transformation phases to be fully connected and maintained in both directions. We created additional auxiliary modules that serves for this purpose, including *ast2coords*, *coords2interp*, *interp2domain*. Taking *interp2domain* module as an example, it maintains the unordered map that contains pair whose key is the interpretation (*Interp*)object, and the value is the object in the domain. It also maintains a corresponding unordered map structure that contains the pairs whose key is the object in the domain, and the value is the interpretation object. For each of the auxiliary module, it maintains such connection for all relevant terms mentioned in the BNF grammar, including the *VecIdent*, *VecExpr*, *Vector*, *Vector_Def*.

How does the Interpretation module build the connection from end to end when a node in the AST is found based on the matching pattern provided? Once the AST node is retrieved from the top-level AST node, handlers call the **mk_Entity** functions of the *Interpretation* module. *Entity*, here, refers to the specific kind of entity found in the code. These syntactic entities include *VecIdent*, *VecExpr*, *Vector*, and *Vector_Def*. For each of the **mk_Entity** functions, it contains the following

actions to build the interpretation. We use one of the most representative function to explain what happens *under the hood*.

When the AST Matcher finds a *VecVecAddExpr*, it will trigger the following sequence of actions.

- The Interpretation module first gets the Coord objects from the the *ast2coords* module for both left and right operands, which are expressed as **VecExpr** terms. It is a pre-condition for constructing the top-level *VecVecAddExpr* that both the left and right operands already exists in the system.

- *ast2coords* module constructs the *VecVecAddExpr* via the function *mkVecVecAddExpr* to construct the coords object for *VecVecAddExpr*. The Coords *VecVecAddExpr* object gets constructed and stored.

- Interpretation module gets the space of the top-level *VecVecAddExpr* from the Oracle module.

- *coords2dom* module contains the mapping from coords objects to domain objects. By passing the coords object pointers of both left and right operand expressions, we get the corresponding domain object pointers.

- After obtaining the space of the top-level node, and both of the domain objects of the left and right operands, the *mkVecVecAddExpr* function in the *domain* module will construct the domain objects of the top-level *VecVecAddExpr* object.

- Once that's constructed, we store the pair (coords, domain) in the *coords2dom* module.

- Then we get teh *Interp* object from the *coords2interp*module for both the left and right operands expressions, together with the *coords* object and the *dom* object that we just constructed to create the *VecVecAddExpr* object in the *Interp* module.

- Now we finished constructing the corresponding objects of all phases, from *ast* to *dom*, for *VecVecAddExpr*, we then construct the mapping by adding the pair (coords,interp) and the pair (interp, dom) separately into *coords2interp* and the *interp2domain* module.

The example on how to construct the *VecVecAddExpr* demonstrated the workflow. It applies for constructing the interpretation for other terms as well, such as the *VecVarExpr*, *VecExpr*, *VecLitExpr*, *VecParenExpr*, *VecDef*.

Domain AST and Domain of Discourse We created the library in Lean that formalizes the construction of certain objects, in our case, the spaces, vectors, vector expressions, vector variables and the set of legal operations. Lean has a powerful mechanism to support abstract data type construction and type checking. Therefore, after extraction and augmentation of the code with the additional information from the Oracle module, the source code gets lifted to Lean. The code and the logic in terms of mathematical physics logic are no longer stripped away and Peirce provided a way for those code elements to carry the physical type with them.

How it actually gets done inside Peirce After the Interpretation module methods *mk_Entity* are triggered, the corresponding domain objects are constructed as well. Therefore, the *domain* object maintains all the available spaces, vector variables, vector expressions including vector literal expression, variable expressions and add expressions. These entities are isomorphic to the domain objects sets, but yet not exactly the term that we use in the logic formalized in Lean, therefore, we consider them more like a domain AST that could be rendered easily to the domain logic.

The library in Lean mentioned above is supposed to be the canonical description of the objects that inhabit in the specific domain of discourse. It requires engineering effort to build and implement the libraries. However, the good thing is that it is built once-and-for-all effort that will not be wasted. Because it is stable with respect to the algebra and mathematical properties that they carry intrinsically.

Main	Clang	Coords	AST	Interp	domain
					Space
		VecExpr	union	mkVecExpr	VecExpr
		VecLitExpr	VecLitExpr	(uses mkVector)	VecLitExpr
HandlerForCXXConstructLitExpr	CXXConstructExpr	VecIdent	VecIdent	mkVecIdent	VecIdent
HandlerForCXXMemberCallExprRight_DeclRefExpr	DeclRefExpr	VecVarExpr	VecVarExpr	mkVecVarExpr	VecVarExpr
HandlerForCXXAddMemberCall, handleMemberCallExpr	CXXMemberCallExpr	VecVecAddExpr	VecVecAddExpr	mkVecVecAddExpr	VecVecAddExpr
HandlerForCXXConstructAddExpr(recursive)	CXXConstructExpr	Vector	Vector	mkVector	Vector
VectorDeclStmtHandler, handleCXXDeclStmt (rec)	CXXConstructExpr	Vector_Def	Vector_Def	mkVector_Def	Vector_Def
CXXMemberCallExprArg0Matcher					
handle_arg0_of_add_call (recurse)					
CXXMemberCallExprMemberExprMatcher(paren)					
handle_member_expr_of_add_call					
CXXConstructExprMatcher (|lit | add)					

4.2 modularity design

Initially when we define Peirce, we not only care about enabling Peirce with the capability to analyze th complex code. As a software system that might evolve into a complex system itself, we designed Peirce in a principled manner. It needs to be flexible to change and evolve, modularized and easy to maintain.

A larger print of this diagram can be found in Appendix A.

Chapter 5

Case Study on Concrete code

Why we have application code like that? The aim of the work carried out and reported on in this thesis was to build a proof of concept for an interpretation builder, a software system that, when aided by an oracle, can mostly automatically construct interpretations for the kind of code now widely used to program robotic and other cyber-physical systems. In this chapter, we show such a system. While it is incomplete in ways that we have already discussed, it fully demonstrates the feasibility of establishing end-to-end interpretations, starting from source code and ending in a domain of physical abstractions formalized in a logic suitable for hosting the abstract languages of mathematical physics.

In this chapter, we present what our current demonstration prototype system can do and we argue that it establishes a path forward for an extensive research program in the physical semantics of the code and in code checking for practical robotic and other cyber-physical systems.

In a nutshell, we designed a concise library in Lean serving as a proxy for a complete Euclidean geometry library. Ours is currently limited to handling vectors that belong to distinct vector spaces, and for performing vector add and assignment operations, and the interpretation of vector literal, variable, add, and binding expressions in C++ code. We see no impediments to extending this work to full affine and Euclidean space abstractions, including support for interpretations of matrices, e.g., as representing transformations of drone locations and orientations in space.

It is a fact that most of the projects where the system interacts with the real-world or provides control to the entities of the real-world needs to have an accurate mapping of the objects and their physics properties, such as location, velocity, momentum, and time. The mathematics that supports such computations in non-relativistic settings is that of affine and Euclidean geometry and of Newtonian physics. Our simplified affine space library serves the purpose of enabling exploration and experimentation with our approach.

To date, we have applied it to hand-crafted C++ code. That said, we have designed the development platform for our system to enable its easy extension to thet handling of C++ code written against the ROS libraries. We have a focus of applying the Peirce system to analyze real ROS code and assist in checking of ROS code for physical type errors in the near future. Our choice of C++ as a source language for this project was driven by our aim to analyze ROS code in particular in the short term.

Introducing the library We built a very simple C++ library with a lightweight abstraction for representing vectors and vector addition operations. Here is the code. It provides a simple definition of the Vector class (Vec) with one primary operation: vec_add, taking 2 Vec arguments and returning a Vec as a result.

```
class Vec {

public:
  Vec(float i= 0.0, float j= 0.0, float k = 0.0):_x (i),_y (j),_z (k){};
        void set(float x, float y, float z)
        {
                _x = x;
                _y = y;
                _z = z;
        }
  float get_x() const{return _x;}
```

```
  float get_y() const{return _y;}
  float get_z() const{return _z;}

  Vec& vec_add(Vec& v)
    {
      set(v._x + _x, v._y + _y, v._z + _z);

      return *this;
    }
  ~Vec(){};

 private:
        float _x;
        float _y;
        float _z;
};
```

Simple Vector library Since Vector is an essential element in affine and Euclidean geometry, we have to module these mathematical concepts and for brevity, we use a tuple of floats to represent it. In the current experimenting code, the *Vector* only has one method, which is the "vec_add" function. We leave it in the future work to support the full-fledged Vector computation. We note that our Vec class imposes about the same amount of abstraction as ROS. It does not use completely *bare* floating point three-tuples, but neither does it representing the information needed for physical type checking (such as space to which a vector belongs or the coordinate frame in terms of which its underlying coordinates are represented).

Simple Application Our test cases are built on this library. A typical test case is simply a main routine in which we instantiate several vector objects—e.g., **v1** and **v2**–and use them in expressions to initialize the values of other new vector objects,

such as **v3** and **v4**. When running Peirce on such a piece of code, it asks additional information from the Oracle for each and every relevant term, including vector literal expressions, constructor expressions, identifiers, operator (add) expressions, and binding (assignment) commands. In the current version, it interactively asks the programmer to indicate the particular *space* that the associated vector is assumed to inhabit.

Here is one of our test cases. We elide the inclusion of the Vec class library above.

```
// A simple case
  Vec v1(1.0,1.0,1.0);
  Vec v2(2.0,2.0,2.0);


  Vec v3 = v1.vec_add(v1);
  Vec v4 = v1.vec_add(v2);
```

The code snippet above in C++ declared 2 vector instances, namely *v1* and *v2* by calling the constructor and provide it with the initial values. It is clear in the AST dump, easily obtained from Clang, that, at the top level, line *Vec v1(1.0,1.0,1.0);* is rendered as an AST node of type, *DeclStmt,* in Clang. The Vector variable name *v1* is a node of type *VarDecl*, and a constructor node *CXXConstructorExpr* has as its children the elements of a vector literal expression, $(1.0, 1.0, 1.0)$. The *CXXConstructExpr* has 3 children. Essentially they are *FloatingLiteral* objects, which correspond to the numerical values passed to the Vec constructor. How we mapped this structure to the definition in the domain is that the *DeclStmt* is considered to be a *Vec_Def* with two essential components, *VecIdent* and *VecExpr*. *VarDecl* from the Clang representation gets mapped to the *VecIdent*. The *CXXConstructExpr* in Clang gets mapped to the *Vector* class in the domain.

Similar logic applies to the line $Vecv3 = v1.vec\_add(v1)$, in this case, the *CXXConstructExpr* is a bit more complicated than the previous one. *ImplicitCastExpr* is an auxiliary Object from the Clang hierarchy design and it has no corresponding code elements, and therefore it can be ignored. Inside this *CXXConstructExpr*, it is a

CXXMemberCallExpr, this object corresponds to the expression $v1.vec\_add(v1)$. It has two children, namely *MemberExpr* and *DeclRefExpr*, the *MemberExpr* is the *vec_add* operation and the *DeclRefExpr* is the right operand **v1** and the child of *MemberExpr*, *DeclRefExpr* is the left operand of *vec_add*.

The *CXXConstructExpr* can get complicated when there are nested terms in the *VecExpr*. However, at the top level, it is always consistent that a *Vec_Def* is defined by a *VarIdent* assigned by a *VarExpr*. In Clang terms, *DeclStmt* is defined by a *VarDecl* assigned by a *CXXConstructExpr*.

Based on the AST dump file, we first define the matching pattern to match the *DeclStmt*s from the entire AST dump file. Once those nodes are captured, we use those nodes as the top level nodes to do further matching on inner nodes that have corresponding code elements. From the examples above, there are different kinds of *CXXConstructExpr* expressions, some of them contains *FloatingLiteralExpr* and others contains *CXXMemberCallExpr* expressions. Therefore it is meaningful to dispatch base on different constructions.

```
`-CompoundStmt 0x55719bce64e0 <col:32, line:47:1>
  |-DeclStmt 0x55719bce3d68 <line:39:3, col:22>
  | `-VarDecl 0x55719bce3a08 <col:3, col:21> col:7 used v1 'class Vec' callinit
  |   `-CXXConstructExpr 0x55719bce3d20 <col:7, col:21> 'class Vec'
        'void (float, float, float)'
  |     |-ImplicitCastExpr 0x55719bce3cd8 <col:10> 'float' <FloatingCast>
  |     | `-FloatingLiteral 0x55719bce3a68 <col:10> 'double' 1.000000e+00
  |     |-ImplicitCastExpr 0x55719bce3cf0 <col:14> 'float' <FloatingCast>
  |     | `-FloatingLiteral 0x55719bce3a88 <col:14> 'double' 1.000000e+00
  |     `-ImplicitCastExpr 0x55719bce3d08 <col:18> 'float' <FloatingCast>
  |       `-FloatingLiteral 0x55719bce3aa8 <col:18> 'double' 1.000000e+00
  |-DeclStmt 0x55719bce5920 <line:40:3, col:22>
  | `-VarDecl 0x55719bce3d90 <col:3, col:21> col:7 used v2 'class Vec' callinit
  |   `-CXXConstructExpr 0x55719bce58d8 <col:7, col:21> 'class Vec'
        'void (float, float, float)'
  |     |-ImplicitCastExpr 0x55719bce5890 <col:10> 'float' <FloatingCast>
  |     | `-FloatingLiteral 0x55719bce3df0 <col:10> 'double' 2.000000e+00
```

```
|     |-ImplicitCastExpr 0x55719bce58a8 <col:14> 'float' <FloatingCast>
|     | `-FloatingLiteral 0x55719bce3e10 <col:14> 'double' 2.000000e+00
|     `-ImplicitCastExpr 0x55719bce58c0 <col:18> 'float' <FloatingCast>
|       `-FloatingLiteral 0x55719bce3e30 <col:18> 'double' 2.000000e+00
```

Complicated situations: Handling the recursion The simple definition above does not account for all the possible ways of defining a Vector. There are cases where the *VecExpr* are nested. Based on the BNF grammar, it is absolutely valid to define a *VecExpr* in the following manner.

```
Vec v5 = (v1.vec_add(v1)).vec_add(v1);
Vec v6 = (v1.vec_add(v1)).vec_add(v1.vec_add(v2));
```

It requires careful design to match on the right node at a different level to recursively construct the interpretation for certain code elements. Based on the AST View of the Clang Parsing result, we designed the following algorithm to recursively match on the code elements that require to establish the interpretation.

Consider rephrase the following pseudo-code using algorithm representation.

```
Expression&  handleCXXMemberCallExpr(CXXMemberCallExpr& root)
{
        // handle root.left cases
        // left case 1: DeclRefExpr
        if(root.left.type == DeclRefExpr)
        {
                VecVarExpr& left_expr =  handleDeclRefExpr(root.left);
                return left_expr;
        }

        // left case 2: CXXMembercallExpr
        else
```

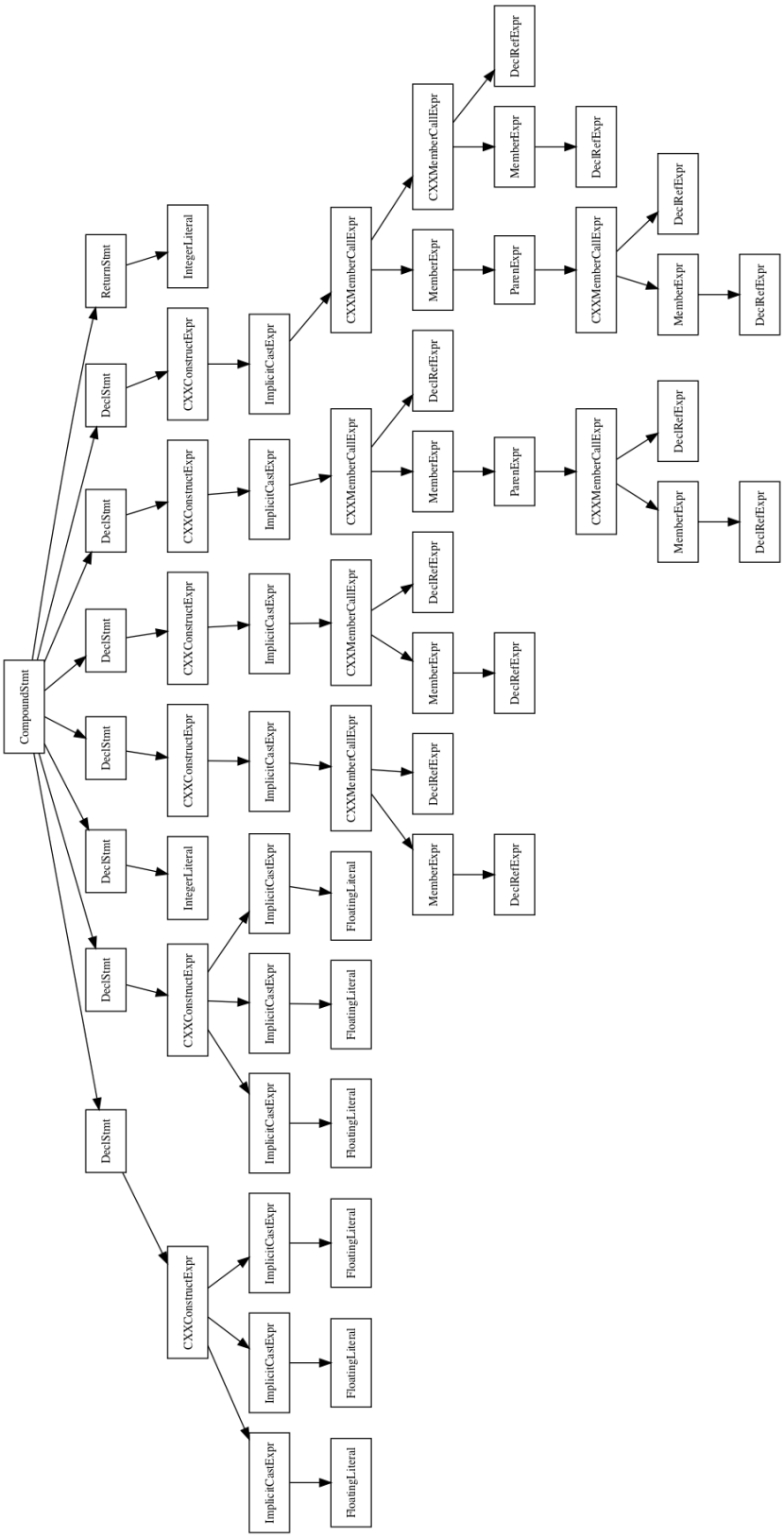

Figure 5-1: AST View in graphics

```
        {
                Expression& left_expr = handleCXXMemberCallExpr(root.left);
        }

        // handle root.right cases
        // right case 1: MemberExpr
        if(root.right.type == MemberExpr)
        {
                Expression& right_expr = handleMemberExpr(root.right);
        }

        // glue right  together
        VecAddExpr& vae = addVecAddExpr(space, root, left_expr, right_expr);
        return vae;
}

VecVarExpr& handleDeclRefExpr(DeclRefExpr& leaf)
{
        return lookUpInDomain(leaf);
}

DeclRefExpr& handleDeclRefExpr(CXXMemberCallExpr& root)
{
    if(root.type == DeclRefExpr)
    return root;
}

VecAddExpr& handleMemberExpr(MemberExpr& root)
{
```

```
        // MemberExpr only has one child
        // case 1
        if(root.right.type == DeclRefExpr)
        {
                VecVarExpr&  child_expr = handleDeclRefExpr(child);
                return child_expr;
        }

        // case 2
        else
        {
                // ignore the cast
                const Expr* implicit = root->IgnoreImplicit();
                const CXXMemberCallExpr * child =
                static_cast<const CXXMemberCallExpr *>(implicit);
                return handleCXXMemberCallExpr(child)
        }
}
```

Figure 5-1 made the hierarchy of the AST explicit. The transformation at different phases that we described in Chapter 4 happens by sequence once a code element is found in the AST based on the matching patterns that we specified to the entity in the domain of discourse. To put it together, we use Fig5-2 to demonstrate how Peirce process the concrete example including both flat and nested definitions using expressions.

Currently, we host this project on Github and we configure the virtual environment to set up the infrastructure to run this project. Please contact the authors if you have any thoughts or questions.

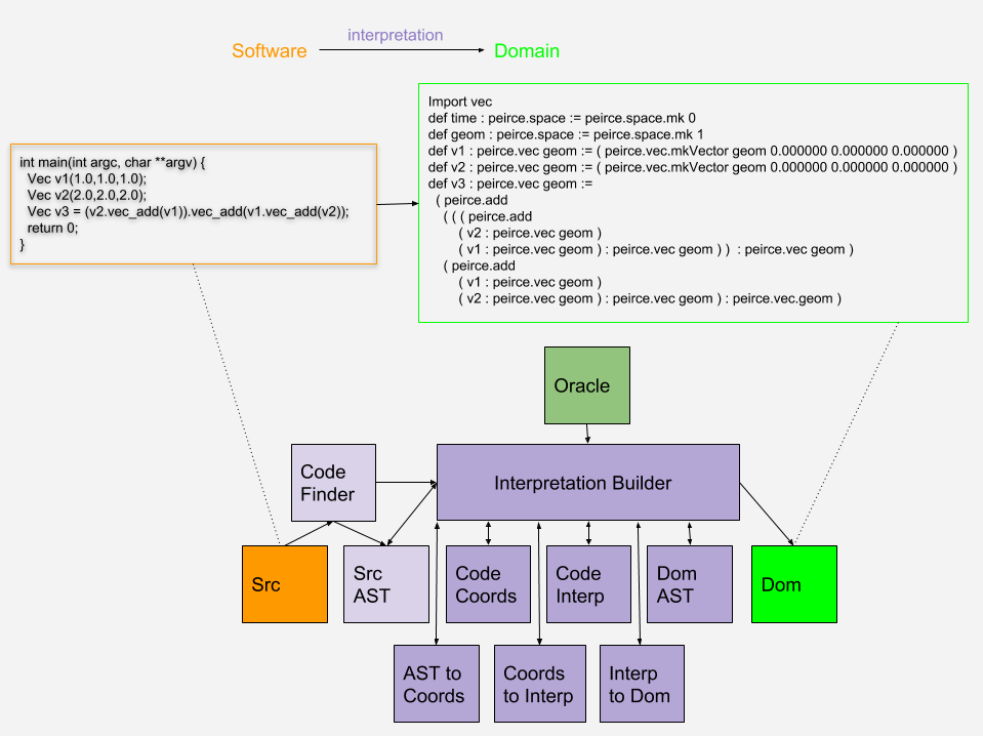

Figure 5-2: Peirce transforms the pure symbolic code to domain AST

Chapter 6

Discussion and Future Work

6.1 Discussion

About our work In this thesis, we demonstrated the inadequacy of prevailing approaches to representing physical objects in terms of programming-language-level types, objects, and operations. While much information about physical types can in principle be representing in the types definable in such languages, in practice, software developers tend to favor representations that are over-abstracted from the mathematical physics of the real-world. We have shown that it is feasible, with the assistance of an *oracle*, to construct interpretations of that now carry complete information about the physics that is being represented for a checker to decide if consistency is preserved across a system. Code that in its raw state is over-abstracted and therefore augmented with an explicit physical semantics, is now checkable for what we call abstract, or physical, type errors.

6.1.1 A Physical Semantics of Code

The concept of a *physical semantics of code* that we introduced in this work draws from the corresponding concept of the semantics of predicate logic, rooted in the notions of a *denotation* function and a *valuation* function.

A denotation associates logical terms, analogous to code elements, with meanings

in some other domain of discourse, which in our work would be a domain of mathematical physics. Our system builds a denotation in the domain of mathematical physics for selected elements in C++ code and then assigns a consistency value to these elements by checking the corresponding domain elements for type inconsistencies. The elements of interest in C++ can, at least so far, be recognized based on the code, but constructing a denotation requires extra information, which we obtain from an *oracle*.

An oracle is any source of the extra information required to construct a complete physical denotation. In our current realization, we ask a human analyst for the required information for each and every relevant code element, including expressions, sub-expressions, identifiers. Clearly, this approach is not ideal for the human analyst. Rather, it is a demonstration of the principle. We are confident that the approach can be greatly improved with modest additional investment.

The analogy with the denotational semantics of predicate logic is worth further discussion. In predicate logic, there exists the notion of a model. A model in predicate logic consists of a set of objects in the domain of discourse and an interpretation, *I*.

Firstly, we will discuss the interpretation. In predicate logic, an interpretation associates constant values and predicates with meanings in a domain of discourse. It associates each constant to a unique entity in the domain and it associates each predicate to a unique, fixed set of entities in the domain. Our interpretation construct does this and more. In particular, it associated vector literal expressions in code with corresponding objects of type *vector* in our Lean representation of vector spaces.

The second component of denotational semantics for expression in predicate logic is called a *variable assignment function*. In the same way that an interpretation maps constants to their meanings, the variable assignment function maps *variables* in the logic to corresponding meanings. In the lexicon of this thesis, our *interpretation* implements both concepts in a uniform manner, mapping both vector-valued constants and variables, to their corresponding meanings in physical vector spaces.

The third component of denotational semantics for predicate logic is a *denotation function*. It is just a compound function that when applied to constants or predicates

reduces to the interpretation function and when applied to variables reduces to the variable assignment function. Defining it makes it easier and cleaner to define the last and crucial element of semantics, the *valuation function*. What we have called an *interpretation*, which handles the mapping of constants, variables, and larger expressions to physical meanings, is thus what logicians would call a denotation.

Finally, a *valuation function* in predicate logic associates truth values to logical formulas in such a way that the value *true* is assigned if the expression corresponds to the actual state of affairs in the domain of discourse, and *false* otherwise. For example, a logical expression that asserts that Mary is friends with Tom would be assigned the value true if the human being corresponding to the symbol Mary really is real-world friends with the human being corresponding to the symbol, Tom. Our physical type checker performs an analogous function by signaling physical type errors (like false values) when logical expressions in the code do not correspond to meaningful situations in the physical domain represented by corresponding expressions in Lean.

There are elements in the semantics of predicate logic that we do not have to consider. Because C++ code has no notion of quantifiers, for example, there is no need to deal with the complications they introduce in the definition of a semantics for predicate logic. Conversely, there are aspects of C++ for which we would like to have semantics that have no analogs in the semantics of predicate logic, most notably the computational semantics of C++. Our work to date provides what might be called a *structural physical semantics* of code, but not a *behavioral physical semantics*. We make no attempt at this time to explain what vectors are *computed* by given expressions, for example, and we have no way to check such meanings for richer constraints on the physics of the domain, such as limits on permissible magnitudes of computed vectors.

6.1.2 Design Trade-offs

In terms of implementation details, we represent a denotation, a physical semantics, for elements of C++ code by a composition of bijective mappings: from code element to code AST node, then to code coordinates objects, onto *interp* object, to

domain AST nodes, and finally to concrete syntax in the logic of the Lean prover. With this denotation in hand, our physical type checker then assigns consistency values by detecting and flagging inconsistencies in the C++ code given a constructed denotation/interpretation.

For architectural and engineering purposes, we designed it to be *two-way*, which means that all the states of those entities from Clang AST to domain AST and back are fully connected. One reason that we adopted such a design is for the purpose of tracing back to source code when physical type errors occur. Once such an error is detected in Lean code, via the connection from domain AST to Clang AST, we can trace back to the source code to point out precisely where the physical type error is. Another use for this *back-mapping* is to eventually enable the annotation and transformation of the actual C++ source code based on the results of the analysis done on the constructive logic representation of the full physical semantics. Among other things, this would allow for the introduction of run-time assertion checks in the source code to detect invariant-violation errors in code that cannot readily be detected statically. Many potential clients could be built out of this mechanism, including but not limited to capabilities for programming understanding, test case generation, etc.

Secondly, we will discuss the entities in the domain. Those entities are usually constants as all the terms and expression has to be evaluated down to a value, which is the entity in the domain. We constructed entities parameterized with spaces in the domain model to provide a way to be flexible to cater to the complexity in the code base, which is done with the assistant of the Oracle model.

Overall, the interpretation module in our work function as such a mapping from the code elements of interest to the corresponding augmented entities in the domain to create such isomorphism between two worlds. We model the behaviors of the entities with respect to the mathematical and physical laws and operations, which are the predicated constants in the conventional sense and the augmented domain AST objects are the constants in the domain of discourse.

6.2 Future Work

6.2.1 Design to be refined

By following principled design concepts, we believe Peirce could evolve into a powerful tool that could assist the analysis of large code bases existed in the current cyber-physical or robotic systems or assist in developing software for future applications. Yet, there are many aspects to be refined before it can provide actual service on genuine codes that runs on robotic systems. For the near future work, we believe the following aspects are worthy directions to head into.

On the theoretic model In order to enforce physical abstractions to the extent that no failures are caused by the kinds of inconsistencies described in this thesis, dynamic methods need to be incorporated to further check the code at run time. As a concrete example, our Lean-based vector space abstraction makes the *space* in which a vector *lives* part of its static type, but in another version of the library that will replace the current minimal version, the *affine frame* in terms of which underlying coordinates are expressed is not. The reason is that run-time change of basis operations can be used to enable the application of operations to vectors expressed in different frames. We do not want to statically prohibit such operations. Rather, we hope to annotate code with assertions and coercion operators to ensure that no operations are ever applied to objects in the same spaces but with incompatible coordinates. Clearly, another important aspect that needs to be explored is the extension of available physical abstractions, i.e., domains of discourse, to which code elements can be mapped.

On refining the Interpretation Our interpretation mapping table is the core component that links the code entities to data values formalized in Lean as derived from the physical domain of discourse. It requires certain performance optimization technique to construct the interpretation table efficiently. Currently, we index code objects by the pointer addresses of their AST nodes in memory. This will become a

concern when the code base that Peirce analyses extends beyond a single compilation unit. First, we need durable representations of code elements. Second, our approach will have to scale to code bases that can't all be parsed simultaneously into main memory.

On the abstract algebraic structure Cyber-physical systems work in many different physical domains. There are many physical quantities to be represented, in general, with corresponding algebraic structures. Most terrestrial robotics system will operate in Euclidean geometric spaces, but they might also have to be checked for software consistency with respect to such quantities as temperature, mass, time, electrical current (to rotors), luminous intensities, and so forth. Satellite or other space systems might encounter relativistic effects and thus require non-Euclidean physical semantics. Our proof of concept, system, Peirce only tackled the vector space component of affine and Euclidean geometry, and only incompletely. There are many more structures to be modeled. A complete vector space abstraction required the addition of a scalar multiplication operation, for example. To extend to affine spaces, we need to add representations of points and operations involving points and vectors. To extend it to the domain of Euclidean spaces, in which distances and angles are defined, the standard inner product needs to be defined. Moreover, complex transformations, such as rigid body motions (translations, rotations) will need to be modeled and denotation-building-mechanisms will have to be designed and implemented.

On the clients In the diagram4-1, Peirce was designed not only to provide type checking for the code base but also to eventually be extended to support a broad array of other uses. These include aiding software developers in understanding the physical meanings of their code; tailored test cases generation constrained by physics; optimization of simulations in light of physical constraints that can be inferred from the denotations of semantically impoverished code; and more.

6.3 Conclusion

This work begins to provide a stronger theoretical foundation for work on physical units checking and real-world types carried out previously by Elbaum, Sullivan, Knight, Xiang, and many others. It suggests a new direction for fundamental computer science research, on the semantics of programming in cases where the code itself is insufficient to ascertain its own meanings. Rather, additional information from outside is needed to construct a proper denotation for the purely symbolic terms of the programming *logic*. This work provides additional evidence for the view initially proposed by Knight, Xiang, and Sullivan that, in addition to the traditional code, software for cyber-physical (or perhaps any kind of) systems should now be accompanied by a separate, explicit, computable interpretations. We hope this work will open new lines of inquiry into the (physical) semantics of software.

Appendix A

The API design across system

Main	Clang	Coords	AST	Interp	domain
		VecExpr	union	mkVecExpr	VecExpr
		VecLitExpr	VecLitExpr	(uses mkVector)	VecLitExpr
HandlerForCXXConstructLitExpr	CXXConstructExpr	VecIdent	VecIdent	mkVecIdent	VecIdent
HandlerForCXXMemberCallExprRight_DeclRefExpr	DeclRefExpr	VecVarExpr	VecVarExpr	mkVecVarExpr	VecVarExpr
HandlerForCXXAddMemberCall, handleMemberCallExpr	CXXMemberCallExpr	VecVecAddExpr	VecVecAddExpr	mkVecVecAddExpr	VecVecAddExpr
HandlerForCXXConstructAddExpr(recursive)	CXXConstructExpr	Vector	Vector	mkVector	Vector
VectorDeclStmtHandler, handleCXXDeclStmt (rec)	CXXConstructExpr	Vector_Def	Vector_Def	mkVector_Def	Vector_Def
CXXMemberCallExprArg0Matcher					
handle_arg0_of_add_call (recurse)					
CXXMemberCallExprMemberExprMatcher(paren)					
handle_member_expr_of_add_call					
CXXConstructExprMatcher (|lit | add)					

Figure A-1: System modular design layout

Appendix B

Sample Concrete Code

```
class Vec {

public:
  Vec(float i= 0.0, float j= 0.0, float k = 0.0):_x (i),_y (j),_z (k){};
        void set(float x, float y, float z)
        {
                _x = x;
                _y = y;
                _z = z;
        }

  float get_x() const{return _x;}
  float get_y() const{return _y;}
  float get_z() const{return _z;}

  Vec& vec_add(Vec& v)
    {
      set(v._x + _x, v._y + _y, v._z + _z);

      return *this;
```

```
    }


  ~Vec(){};

 private:
        float _x;
        float _y;
        float _z;

};

#include <iostream>

using namespace std;

int main(int argc, char **argv){

  Vec v1(1.0,1.0,1.0); // as in frame 1; def v1 := mkVec(...)
  Vec v2(2.0,2.0,2.0); // as in frame 2;

  Vec v3 = v1.vec_add(v1); // as in the same frame as v1 -- frame 1
  Vec v4 = v1.vec_add(v2); // should be rejected;

  return 0;
}
```

Bibliography

[1] Mishap Investigation Board and PageKicker Robot Eddington. *Mars Climate Orbiter: Phase I Report*. Nimble Books LLC, Ann Arbor, CA, USA, USA, 2013.

[2] T. D. DeRose. Theory and practice of geometric modeling. chapter A Coordinate-free Approach to Geometric Programming, pages 291–305. Springer-Verlag New York, Inc., New York, NY, USA, 1989.

[3] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[4] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 681–690, New York, NY, USA, 2011. ACM.

[5] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. Phriky-units: A lightweight, annotation-free physical unit inconsistency detection tool. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 352–355, New York, NY, USA, 2017. ACM.

[6] Kevin Sullivan Jian Xiang, John Knight. "is my software consistent with the real world?". *High Assurance Systems Engineering (HASE)*, 2017.

[7] `https://leanprover.github.io`.

[8] Coq – the proof assistant official website. https://coq.inria.fr, 2018 (Accessed 10/13/2018).

[9] J. Xiang, J. Knight, and K. Sullivan. Synthesis of logic interpretations. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, pages 114–121, Jan 2016.

[10] http://sourceforge.net/projects/fgflightplanner.

[11] J. Ore, S. Elbaum, and C. Detweiler. Dimensional inconsistencies in code and ros messages: A study of 5.9m lines of code. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 712–718, Sep. 2017.

[12] http://wiki.ros.org/common_msgs, Accessed December 12, 2018.

[13] http://wiki.ros.org/geometry_msgs, Accessed December 10, 2018.

[14] Cygnus delays iss berthing following gps discrepancy. https://www.nasaspaceflight.com/2013/09/cygnus-cots-graduation-iss-berthing/.

[15] http://clang.llvm.org/docs/IntroductionToTheClangAST.html, 2010 (Accessed December 7, 2019).

[16] https://clang.llvm.org/docs/LibASTMatchersTutorial.html, 2019 (Accessed January 03, 2019).