Enabling External Physical Type Annotations for Physically Relevant C++ Code Segments

A Technical Report
presented to the faculty of the
School of Engineering and Applied Science
University of Virginia

by

Charlie Houghton

May 6, 2021

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Charlie Houghton

Technical advisor: Kevin J. Sullivan, Department of Computer Science

Enabling External Physical Type Annotations for Physically Relevant C++ Code Segments

Charlie Houghton
School of Engineering and Applied Science
University of Virginia
Charlottesville, Virginia, USA
choughton@virginia.edu

***Abstract*—Cyber-physical systems, such as autonomous vehicles and aerospace guidance, interface software with the physical world. Software errors can lead to expensive and catastrophic failures, so it is valuable to formally verify software correctness. Physical semantic errors occur when data representing physical phenomena are used in operations in ways that are not physically meaningful, e.g., by adding numbers that represent quantities that, in the physical world, cannot be added. How can cyber-physical systems engineers discover physical semantic errors in their programs? We developed a Visual Studio Code extension that embeds additional type information into physically relevant segments of C++ code and checks for physical semantic errors for time operations.**

***Index Terms*—cyber-physical systems, physical semantics, visual studio code**

I. Introduction

Cyber-physical systems (CPSs) interact with the physical world, but they are not necessarily constrained by the physical domain in which they operate. For example, suppose a CPS program erroneously adds 2 seconds and 3 years together and comes up with an answer of 5. In a CPS program devoid of physical meaning, this would be a computable and valid operation, however, when the intended physical meanings of the arguments, 2 and 3, are considered, this operation becomes physically non-sensical, as the real result is neither 5 seconds nor 5 years. Any operation where a CPS performs a machine-computable but physically incorrect operation is called a *physical semantic* error. "Major systems malfunctions have occurred due to the machine-permitted evaluation of expressions that have no well defined physical meanings," [1] one such example being NASA's 1999 Mars Climate Orbiter which was unsuccessful due to a failure to "convert from English to metric" before launch [2].

The aim of UVa's Peirce Project, led by professors Kevin Sullivan and Sebastian Elbaum, is to avert such errors by annotating CPS code with formal specifications of intended meanings of code elements. My project sought to build an interactive integrated software development environment (IDE) extension for the Peirce Project to provide an improved workflow for the software engineer who needs to annotate code and understand the results of annotation-enabled semantic analysis of CPS code.

Peirce detects physical semantic errors in C++ code and is used as the core back-end tool for this project. We built a Visual Studio Code (VSCode) extension that allows CPS engineers to annotate physically relevant code segments with appropriate physical types, and will indicate which code segments contain physical semantic errors, as identified by Peirce. Peirce itself is a command-line tool that compiles C++ code, generates an abstract syntax tree (AST), identifies which AST nodes are physically relevant, and asks an *oracle* to provide a physical interpretation for each node. Currently, the oracle is the engineer him or herself.

Before the completion of this project, CPS engineers could only interact with Peirce directly using a rudimentary command-line interaction. This approach has some drawbacks. An engineer first needs to identify which code segment they wish to annotate, which involves matching code coordinates (i.e. the starting and ending row and column indices) given by Peirce with their code. This process is slow and can grow frustrating for large code bases. An IDE extension that integrates Peirce with the code to-be-annotated streamlines the process of identifying physically relevant segments, providing additional type information, and alerting the engineer of an error. Second, analysis results from Peirce are provided only in the form of plain text error outputs. It would be better if the processes of annotating code and understanding Peirce feedback were gracefully integrated into the IDE used for ordinary software development.

The usability of Peirce impacts the likelihood that it will adopted by CPS engineers. Peirce will only help engineers find physical type errors if it easy to use. This project seeks to lower the barrier to entry, allowing engineers to more easily justify using Peirce for their projects.

II. Related Work

Physical semantic errors encompass more than just unit inconsistencies. Among other tools, Microsoft's programming language F# has a system called "Units of Measure" that detects unit inconsistencies [3]. To explain why detecting unit inconsistencies is insufficient, a notion of an *affine frame* is necessary. An affine frame is a pairing of an origin point and basis vectors, which can be used to create a space, relative to which other points and vectors can be defined. Tools like F# can catch mixed-unit operations, but Peirce can catch mixed-frame operations, even if the units attached to those frames were to be the same.

In December 2019, NASA launched Boeing's CST-100 Starliner spacecraft that planned to dock with the International Space Station but failed due to a misconfigured clock, resulting in the Starliner's autonomous instructions being offset by 11 hours [4]. What occurred was precisely a physical semantic error due to both CPSs performing operations in different affine frames, as their time *origin points* differed. This incident highlights why same-frame verification is a more complete model for judging CPS correctness than only same-unit verification.

As currently used by Peirce, affine frames only exist to define affine coordinate spaces. For this reason, we have combined the notion of affine frames and affine coordinate spaces, which we refer to as *coordinate spaces*. Coordinate spaces must be defined with respect to another coordinate space, but what happens when none are defined? A notion of a *standard frame* is required. The standard frame acts as backstop for the definition of all coordinate spaces. If a coordinate space is defined without a parent space (implicitly containing a frame), it is assumed to be defined with respect to the standard frame.

III. Background

This project is constrained by the Peirce system, so it is important to understand Peirce's operations and interaction model. Peirce operates in two phases. The first phase is *identification*, in which Peirce identifies physically relevant AST nodes and their corresponding code segments for a given C++ file. For clarity and consistency, all mentioned code segments are assumed to be physically relevant unless indicated otherwise for the remainder of this paper.

The second phase is *annotation*, in which the user creates new coordinate spaces and annotates each code segment with a coordinate space and value. Values correspond with literals in the C++ file. Consider the following example:

```
float time_in_seconds = 0;
```

After Peirce has identified the literal 0 as physically relevant, it could be annotated with a *seconds* coordinate space and with a value of 0.

Following annotation, Peirce will output known physical interpretations. Peirce can infer other physical interpretations based on interpretations previously given. If Peirce identifies, either directly or by inference, that there is a type mismatch, Peirce will alert the user. Consider the following example:

```
float time_in_seconds = 2;
float time_in_years = 3;
float total =
    time_in_seconds + time_in_years;
```

The following code would compile and the evaluation of the addition would yield 5, but it would have no physical meaning because of the implicit units mismatch. In other words, this error would go uncaught. However, if the literals are annotated with a seconds space and a years space (defined as a derived space relative to the seconds space), respectively, Peirce would identify a type error in the addition operation, as the values are now marked as being expressed in the coordinate systems of different coordinate spaces. This information can either be used to correct annotations or indicate to the engineer that a code revision is necessary.

IV. System Design

Recall the two phases of Peirce: identification and annotation. How can we use an IDE extension to identify and allow the annotation of code segments, and forward that provided information to Peirce? We split this task into two parts: the VSCode extension, and the API that interfaces with the extension and Peirce. With this design, all extension actions are asynchronous, allowing the user to continue working (e.g., programming) while Peirce operates in the background, only displaying information when finished. The API offloads Peirce runtime dependencies and processing time to a server. Furthermore, using an IDE extension allows for more seamless integration with existing development workflows, no longer requiring the user to navigate away from their IDE to use Peirce.

A. The Extension

The extension maintains a collection of coordinate spaces, code segments, and annotation information for each segment. Code segments and their corresponding annotations are stored in what we'll refer to as the *interpretation table*. All information managed by the extension is external and code independent. We will discuss this decision in a later section.

Before a user can begin to annotate code, the interpretation table must be populated with unannotated code segments, otherwise there is nothing to annotate. This corresponds to Peirce's identification phase, reached through the *populate* API endpoint. Once populated, a user will be notified of all identified code segments via code highlighting and various information widgets. A user can subsequently provide annotations, stepping through each code segment choosing either to annotate it or not. At any time, the user can choose to check whether their annotated interpretations have caused any physical semantic errors via the *check* API endpoint. This will not only inform the user of errors, but will also infer interpretations for other unannotated code segments.

B. The API

The API supports two endpoints, *populate* and *check*, introduced in the previous section. *Populate* runs Peirce on the provided file sent as part of the request and responds with the extracted code coordinates, which the extension uses to populate the interpretation table with unannotated code segments. *Check* runs Peirce in the same way but also does input generation. The input generation system takes the coordinate spaces and annotations and generates a sequence of inputs to the Peirce program such that, when given to Peirce, Peirce understands interpretations equivalent to those understood in the extension. The inferred and erroneous interpretations are sent back to the extension, allowing it to update the interpretation table to reflect that information accordingly.

C. External vs. In-Code Annotations

We chose to store annotations externally, via the interpretation table, rather than storing them within the code, perhaps embedded within C++ comments. External annotations have a number of advantages and drawbacks compared to the in-code alternative. A primary advantage was that the extension could be used without making any changes to existing code. The biggest drawback of external annotations is that, if the underlying code is altered after being annotated, attempting to reconcile existing annotations is difficult. This meant that annotations must be cleared before continuing to annotate. This places a lot of importance on getting the code correct first, then verifying physical correctness. This decision favors established codebases that are changed infrequently. This was a priority for the Peirce Project as a whole, and the decision to implement external annotations is an extension of that priority.

D. The Extension Interface

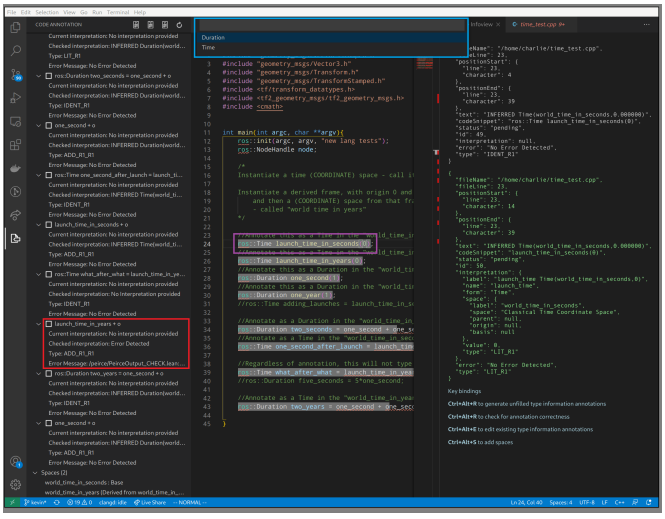

Fig. 1. A screenshot of the VSCode extension on a C++ file. The interpretation table has been populated and checked and physical type errors are displayed.

This screenshot of the VSCode extension shows all of the major tools and features including the information widgets on the left-side and right-side panels and the input system at the top. The line currently being annotated is boxed in purple. The annotation input system is highlighted in light blue. The panel on the right is the Peirce Infoview, displaying JSON objects representing the annotations for all code segments on the line being annotated. The second JSON object is highlighted in green, indicating that it is being actively edited. The interpretation table is on the left panel showing interpretation information for every code segment. An example of a physical semantic error in the interpretation table identified by Peirce is boxed in red, the interpretation table entry for "Checked Interpretation" reads "Error Detected."

V. Results

The extension integrated Peirce with an IDE, simplifying and centralizing its interaction model. A developer can more quickly and accurately annotate their CPS code for physical type errors, all without requiring extra dependencies or extra resources, while remaining in their development environment. We saw the extension saved the most amount of time when identifying code segments. Previously, users needed to manually match code coordinates with code segments. Now, identification is done with VSCode highlighting via the extension.

We found that the process of supplying annotations via the extension was not accelerated significantly compared to using Peirce directly. The information required for Peirce to operate is the same, whichever front-end is being used, so a large change in time cost was not anticipated. In some cases, the annotation process was less clear than when using the CLI. Decreasing annotation time remains an outstanding challenge for the Peirce Project.

VI. Conclusion

We designed and built a VSCode extension for Peirce to enable external physical type annotations for C++ code segments. The extension removes some otherwise necessary installation and resource requirements for Peirce, as Peirce has been offloaded to a server interacted with via an HTTP API. The extension is functionally equivalent to Peirce's CLI front-end, capturing physical type information and forwarding errors to the user. For some actions, the extension is measurably more efficient compared to the CLI, but for others, less. With future enhancements, we hope the extension will be the primary system engineers use to interact with Peirce.

VII. Future Work

A. Information Presentation and Highlight Errors

Currently, the VSCode extension uses a combination of information widgets and user input widgets. Annotation information and type errors, while presented to the user via a number of these widgets, are not quickly understood. Furthermore, because there are a large number of physically relevant code segments for a given CPS program and the interface does not scale well for large files. Further modifications to the information presentation parts of the extension are necessary before the extension can be widely adopted.

Due to an existing issue with Peirce, some code coordinates are incorrect, as can be seen in figure 1.

B. Supplying Physical Type Information and the Oracle

The process of supplying annotations has not been simplified or made significantly faster with this extension and this continues to pose a large time cost to developers that wish to use Peirce. Front-end changes can be implemented in the future to provide some kind of progress information as the user steps through supplying physical type information. Additionally, the *oracle* that is used to supply this information is currently the developer or engineer him or herself. Changes to the oracle have been proposed and any future oracle modifications would need to be supported by the extension.

C. Optimizations and Parallel Processing

API calls to Peirce's inference tool has approximately a 30 second time cost per call even using the small files and annotation sets explored in this initial prototyping project. It would be longer for more complex files and annotation sets. For an interactive extension, this delay is significant and problematic. While this inference delay problem is not within the scope of the extension, it does effect its usability. Peirce optimizations act as extension optimizations, so we anticipate improvements in usability as Peirce itself is improved over time.

Currently, the API only handles a single request at a time. This is because some Peirce communication is done through the file system with specific file names. Peirce was designed as a single-user tool, so some modifications will be required to allow multiple Peirce processes to run concurrently.

ACKNOWLEDGMENT

The author would like to thank Prof. Kevin Sullivan, Andrew Elsey, Ben Ascoli, and Charlie Conneen for their guidance and feedback for the extension and for their technical and conceptual assistance with the Peirce Project.

REFERENCES

[1] K. Sullivan. (2019) Explicating and exploiting the physical semantics of code. [Online]. Available: https://www.nsf.gov/awardsearch/showAward?AWD_ID=1909414

[2] R. Hotz. (1999) Mars probe lost due to simple math error. [Online]. Available: https://www.latimes.com/archives/la-xpm-1999-oct-01-mn-17288-story.html

[3] Microsoft. (2020) Units of measure. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/units-of-measure

[4] K. Chang. (2020) Boeing starliner lands in new mexico after clock error prompts early return. [Online]. Available: https://www.nytimes.com/2019/12/22/science/boeing-starliner-landing.html