A Case Study on Interpretation-based Error Detection

Jian Xiang, John Knight, Kevin Sullivan

Department of Computer Science

University of Virginia

Charlottesville, VA USA

{Jian,Knight,Sullivan}@cs.virginia.edu

*Abstract*—Programs should conform to constraints inherited in the real world, e.g., laws of physics. Violations of these constraints could lead to serious consequences. The concepts of interpretation and interpreted formalism were introduced to check errors that violate real-world constraints systematically. The potential benefits of using interpreted formalism are substantial and have been illustrated in our prior work. However, the interpreted formalism has not been applied to large software systems, thereby the benefits on large systems have yet to be demonstrated. In this paper, we present a case study in which we applied the interpreted formalism to an open-source geographic software with 150K lines of code. The results of the case study suggest that the interpreted formalism (1) is fit for large software systems, (2) is very effective in error detection, and (3) provides efficient support to reduce user effort.

Keywords— Case study, logic interpretation, real-world types, software reliability

# Introduction

Real-world constraints are those inherited from real-world properties (e.g., the laws of physics), and these constraints should be observed in programs. Current research efforts focus on specific kinds of real-world constraints, e.g. units consistency, yet lack systematic approaches to check constraints derive from various sources.

In our prior research work, a new concept of interpretation and a new structure of interpreted formalism were introduced to deal with the issue [][]. An interpreted formalism is a structure that combines software logic with an explicit interpretation. The interpreted formalism permits an approach that systematically allows the definition of constraints derived from the real world. The interpreted formalism supports several analysis techniques that enable automated checking of these constraints. The benefits of using the interpreted formalism is substantial. A case study on a moderate-sized software system of 13,000 lines of code has illustrated the feasibility and benefits of the interpreted formalism [].

However, one case study of such size might not be sufficient to illustrate the pragmatic and efficacy of the interpreted formalism. In this paper, we present a larger case study that we conducted to investigate the utility of the interpreted formalism. In this case study, the interpreted formalism was applied to a geographic software, OpenMap, which contains 150,000 lines of code. The error detection techniques provided by the interpreted formalism were conducted on the OpenMap. As a result, more than dozens of real errors that violate real-world constraints were found; around 50% of the annotation burden required from users were synthesized. The results of this case study suggest that the interpreted formalism (1) is feasible for large software systems, (2) is very effective in error detection, and (3) provides efficient support to reduce user effort.

The remainder of this paper is organized as below: section II reviews the concepts of interpretation and the structure of the interpreted formalism. Section III

# Interpretation and Interpreted Formalism

## Explicit Interpretation

Program elements in programming languages are purely syntactic entities such that without a proper interpretation, they don’t have any real-world meaning. An interpretation adds the real-world meanings to elements in programs. Current software systems frequently document interpretations in an ad-hoc and implicit manner using meaningful identifiers and unstructured comments. Such interpretation leads to problems such as (1) real-world semantics are incomplete, (2) The connections between elements in logic and their real-world entities are under-specified, and (3) Real-world constraints are violated.

To deal with inadequate and implicit interpretations, in our prior work[], we advocated that software systems should include explicit interpretations. An explicit interpretation documents the real-world meanings of logic elements in a precise, clear and explicit manner. With an explicit interpretation, important characteristics of real-world entities, such as units and dimensions, and associated real-world constraints, such as not mixing units, can be clearly defined. Real-world constraints that interpretation exposes can be automated checked.

## Interpreted Formalism

The interpreted formalism is a novel structure that used to incorporate an explicit interpretation into the engineering artifacts. An interpreted formalism combines logic with an explicit interpretation. The logic in an interpreted formalism is defined in whatever manner is appropriate for the system of interest, i.e., the choice of programming language, programming standards, compiler, and so on, are unaffected by the interpreted formalism structure. The key difference, of course, is the addition of the explicit interpretation.

In the development of a particular software system, the task is no longer to develop software. The task is, in fact, to develop an interpreted formalism for the system of interest. Without the explicit interpretation, whatever would be developed as “software” runs the risk of failing to define the desired interaction with the real world correctly, where the implementation of that interaction is the entire purpose of the software system.

## Realization: Real-World Type System

The concept of logic interpretation is well established, but defining the content and structure of an effective and complete interpretation for practical use is a significant challenge. In our preliminary design, the interpreted formalism concept is realized by the concept of real-world types we introduced in our prior work []. The design of the interpretation is a set of real-world types and a set of real-world type rules defined within the framework of a real-world type system. For an interpretation for a software system, real-world types specify characteristics of entities in the real world accessed by the software system; real-world type rules specify the constraints that should be observed in the software system.

## Development of Interpreted Formalism

The concept of interpreted formalism is realized in the form of real-world type system. In order to build an interpreted formalism for a software system of interest, three artifacts need to be developed: (1) real-world types, (2) real-world type rules, and (3) real-world type bindings. We introduced a synthesis framework that largely reduce the effort required in developing these artifacts []. Overall, the effort required from users can be reduced around 50%.

# Error Detection Based on Interpretation

The high-level goal of the interpreted formalism concept is to provide a mechanism to improve software quality. Real-world type systems bring the opportunity to realize this goal by establishing new properties in software programs. 、

The properties associated with real-world type systems are properties derived from real-world contexts. These properties should be established in software systems that manipulate real-world entities. The properties includes:

* Program statements conform to static real-world constraints.
* References from program elements to real-world entities are precise and consistent.
* Approximations caused by hardware are accessible by users.
* Runtime values of program variables conform to real-world constraints.

Several analysis techniques were developed in order to establish these properties []. They are:

* Real-world constraint checking.
* Reasonable range analysis.
* Targeted inspection.
* Runtime assertion generator.

# Case Study

In our prior work, the feasibility and efficacy of real-world type system has been illustrated by application on a moderate-sized software, Kelpie flight planner []. The real-world type system has found several real errors that violate real-world constraints in the software.

In order to better understand the utility of the real-world type system, another bigger case study was conducted on an open-source project named OpenMap []. This case study evaluates the performance of the real-world type system in large-sized software. This section presents the details of this case study.

## Case Study Subject

### Introduction of OpenMap

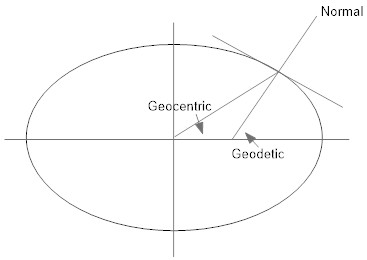
OpenMap is a JavaBean-based toolkit for building applications and applets needing geographic information. Using OpenMap components, users can access data from legacy applications. The core components of OpenMap are a set of Swing components that understand geographic coordinates. These components allow users to show map data and manipulate that data. The software system is 157,858 lines long, is organized as 92 packages, and is contained in 1193 source files.

### Important real-world semantics

Some real-world semantics are important in understanding the errors found in OpenMap. These semantic include:

**Units and dimensions**. The OpenMap software makes calculations involving distances, heights, speeds, angles, time and so on, and it does so using a variety of units. Clearly, the software is of the type for which real-world constraint checking has the potential to discover units related errors. The dimensions and units are all real-world concepts that are defined in the real-world type system by default.

**Geographic and geocentric latitude**. The real-world entity latitude has been widely used in the OpenMap software. The software uses two types of latitude: geographic (geodetic) latitude and geocentric latitude. The two types of latitude are different, and the difference is shown in Fig. 1. Such difference is crucial when the shape of Earth is modeled as an ellipsoid.



1. Two different types of latitude

**Reference level of elevation**. In OpenMap, the computation of the distance between two objects on Earth’s surface frequently involves objects’ elevations. The elevations have different reference levels. Two important reference levels are local ground and mean sea level. The difference between the two reference levels should be carefully handled when the computation expects high accuracy.

## Case Study Procedure

### Purposes

This case study aims at assessing several properties of the real-world type system when applied to large-sized software systems. These properties are listed below and they are assessed in an attempt to answer several research questions:

* **Feasibility**. Is it feasible to apply the technology?
* **Error detection**. How effective are the analysis techniques?
* **Effort level**. How much effort is required to apply the technology?
* **Scalability**. Is the real-world type system scalable?

### Procedure

In order to assess these properties, complete real-world type systems were created for the OpenMap project. Real-world types were created for all real-world entities accessed by the software applications, and variables and methods that access real-world entities were bound to their real-world types. A set of type rules were defined so that relevant relationships between real-world entities could be established. After setting up the real-world type system, analyses were conducted on the OpenMap. Real-world constraint checking was used to detect violations of real-world constraints. Reasonable range analysis was conducted to detect error-prone computations.

## Results

Each property is assessed by collecting a set of data and analyzing the data.

### Feasibility

To assess feasibility, this case study collected the following items:

* Size of real-world type system.

The real-world type system created for Kelpie flight planner was reused in this case study. The 35 real-world types and 97 real-world type rules were reused. Real-world type bindings cannot be reused. In order to fully interpret OpenMap, 1932 real-world type bindings were created.

* Coverage of real-world type system

Variables in 196 source files have been bound with real-world types. Real-world types have been accessed by program elements inside 232 source files. The other source files are not interacting with real-world entities. They do not have real-world type bindings.

* Numbers relevant to error checking

With real-world type system deployed, real-world constraint checking and reasonable range analysis were conducted to detect errors. All 1193 source files were checked for errors. Real errors were found in different files.

### Error detection

The analyses performed on OpenMap showed promising results. Both real-world constraint checking and reasonable range analysis were used for error detection. Real-world constraint checking found real errors that have not been reported before. Reasonable range analysis found statements that could lead to runtime errors.

Data relevant to error checking are shown below:

* Number of reported errors and real errors

TABLE I. summarizes the results of error reported and the number of real errors.

Overall, units checking 49 errors and 8 of them are real errors. Real-world constraint checking reported 53 errors from 18 source files; 24 of them are real errors. Reasonable range analysis reported 29 warnings from 18 source files; 12 of them could lead to runtime errors.

As stated in section III, units checking is a special form of real-world constraint checking, so the errors found by real-world constraint checking include the errors found by units checking.

1. Errors reported and real errors in OpenMap

|  |  |  |  |
| --- | --- | --- | --- |
| Analysis techniques | # of files with reported errors | # of errors reported | # of real errors |
| Units checking | 15 | 49 | 11 |
| Real-world constraint checking | 18 | 53 | 24 |
| Reasonable range analysis | 18 | 29 | 12 |

Every real error was caused by a (or more) real-world semantic. TABLE II summarizes the source files that contains real errors, number of real errors, and the real-world semantics that cause the errors.

1. real errors found by real-world constraint checking

|  |  |  |
| --- | --- | --- |
| Program files | # of real errors | Real-world semantic involved |
| RoadFinder.java | 1 | Latitude and longitude |
| Route.java | 4 | Units |
| Road.java | 4 | Units |
| Gonomic.java | 1 | Latitude and longitude |
| OMDistance.java | 2 | Units |
| TX7.java | 1 | Earth radius |
| LOSGenerator.java  (openmap/tools/terrain/) | 3 | Reference level |
| LOSGenerator.java  (openmap/layer/terrain/) | 3 | Reference level |
| GeoTestLayer.java | 1 | Geodetic and geocentric latitude |
| GeoCrossDemoLayer.java | 3 | Geodetic and geocentric latitude |
| QuadTreeNode.java | 1 | Units |

The errors reported above were detected by real-world constraint checking. In addition, reasonable range analysis was used to detection possible erroneous computations. The analysis found 12 statements in 6 files. TABLE III summarizes them:

1. real errors found by reasonable range analysis

|  |  |  |
| --- | --- | --- |
| Program files | # of real errors | Possible runtime errors |
| CADRG.java | 1 | Division of zero |
| Road.java | 2 | Out of reasonable range |
| Route.java | 2 | Out of reasonable range |
| OMDistance.java | 1 | Out of reasonable range |
| OMRasterObject.java | 2 | Division of zero |
| MercatorUVGCT.java | 4 | Infinite bound |

* Number of false warnings

In the case study, analysis techniques reported more errors than the number of real errors. Real-world constraint checking reported 53 errors, while 24 of them are real errors. These non-real errors are categorized into two kinds: improper usage and false warning.

The definition of improper usage was introduced in []. The improper usage refers to either (a) a variable took on different real-world entities (but the same programming datatype) in different parts of the program, or (b) the elements of an array were not all of the same real-world entities (but were of the same programming type). We consider improper usage as an inappropriate way to access real-world entities.

TABLE III summarizes the improper usage and false warnings found by all analysis techniques.

1. False warnings and impropert usage

|  |  |  |
| --- | --- | --- |
| Analysis techniques | # of improper usage | # of false warning |
| Units checking | 24 | 17 |
| Real-world constraint checking | 25 | 4 |
| Reasonable range analysis | 4 | 12 |

### Effort level

The data collected in this case study pertinent to effort level are list below:

* Reusing existing real-world type systems

The real-world type system we developed for our prior case study were reused in this case study. The existing real-world type system for the prior case study contains 35 real-world types and 97 real-world type rules. Only one real-world type was newly created for OpenMap. Therefore, real-world type system for OpenMap software contains 36 real-world types in total and 97 real-world type rules.

* User’s effort

Reusing existing real-world type system saves users from creating all the real-world types and type rules. Most effort required from users are on creating real-world type bindings. Among the real-world type bindings created for OpenMap, a part of them were created by user, and another part of them were synthesized.

In summary, the total number of real-world type bindings for OpenMap is 1932. 1129 (58.4) type bindings were seeded by users in different source files, and 803 (41.6%) type bindings were synthesized. The binding synthesizer demonstrated better efficacy at early stages of developing type bindings. For the first 507 type bindings, 199 (41.2%) bindings were created by users and 298 (58.8%) bindings were synthesized. For the first 1024 type bindings, 488 type bindings were created by the user (47.7%) and 536 (52.3%) bindings were synthesized.

## Analysis

### Error samples

The errors found by real-world constraint checking involve various kinds of real-world semantics. We list a few sample errors here.

**Error Sample #1.** Four real errors were found in the source file Road.java. They are all misuse of units. The statement below contains two real errors:

kilometers += GreatCircle.sphericalDistance( prevPoint.getLatitude(),

prevPoint.getLongitude(),

thisPoint.getLatitude(),

thisPoint.getLongitude());

For the first error, GreatCircle.sphericalDistance() expects the unit of measurement to be radians for the parameters, but the arguments in this statement are all of units degrees. For the second real error, the return value of the function is of units radians, which is not commensurable with variable kilometers.

**Error Sample #2.** One statement in source file TX7.java contains an inaccurate computation. This statement involves incorrect use of Earth’s *radius*. The statement is:

distance = GreatCircle.

sphericalDistance(lt1, ln1, lt2, ln2) \*

Planet.wgs84\_earthEquatorialRadiusMeters;

This statement computes the distance between two points on the Earth’s surface. According to basic geometry, angular distance (or angle) multiplied by radius yields distance on a great circle of a sphere. The function GreatCircle.sphericalDistance() computes the angular distance between the two points on Earth surface, with the assumption that Earth is a sphere. However, variable wgs84\_earthEquatorialRadiusMeters represents Earth’s equatorial radius with Earth modeled as an ellipsoid. The computation of distance in this statement is not accurate.

**Error Sample #3.** Three statements in file LOSGenerator.java contain inaccurate computations. The inaccuracy is caused by inconsistent *reference level* of elevations. The three statements are similar to the statement below:

double cutoff = startTotalHeight +

Planet.wgs84\_earthEquatorialRadiusMeters;

All three statements intend to compute the distance between an object and Earth’s center by adding Earth’s radius to the object’s height above Earth’s surface ground. The radius here represented by wgs84\_earthEquatorialRadiusMeters is the distance between Earth’s center and *Earth’s surface ground*; but variables endTotalHeight and startTotalHeight represent objects’ heights measured above *mean sea level*. The two reference levels are different. The computations are inaccurate.

The sample errors stated above are errors detected by real-world constraint checking, we present a sample error found by reasonable range analysis below.

**Error Sample #1**. In the file CADRG.java, we found a possible *division of zero* in the statement below.

…

dlon = lon2 - lon1;

…

deltaDegrees = dlon;

…

ret = pixPerDegree / (deltaPix / deltaDegrees);

The variable deltaDegrees represents the difference between two longitude values. Such difference could be zero, which would lead to a division of zero at runtime.

### False warnings

As stated above, real-world constraint checking and reasonable range analysis reported more errors than real errors. These non-real errors are categorized as improper usages and false warnings.

Most improper usages come from statements that are similar to the following one:

lat = Math.toRadians(lat);

lon = Math.toRadians(lon);

Variables lat and lon on the left side represent values of latitude and longitude values in units of radians, but the two variables represent values in units of degree on the right side. The variables take different real-world entities in the same statements. The statements are flagged as improper usage.

Other non-real errors are false warnings. These false warnings frequently involve conversion between different real-world types. For example, two false warnings were reported in statements below:

double lambda = lon \* Degree;

double phi = Math.abs(lat \* Degree);

In the first statement, variable lon which represents longitude of radians is converted to variable lambda which represents longitude measured in degrees. The second statement is similar.

The statements reported as either improper usages or false warnings indicate some error-prone operations. Programmers should double check these statements to make certain that the entities referenced are being used correctly.

## Observations

# Related Work

# Conclusion

##### References

1. S. Abebe, and P. Tonella. “Towards the extraction of domain concepts from the identifiers,” In Proceedings of the 18th Working Conference on Reverse Engineering (WCRE), 2011, pp. 77–86.