Is My Software Consistent With the Real World?

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 from 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 faults 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 fault detection, and (3) provides efficient support to reduce user effort.

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

Check for self plaguerism

# Introduction

Real-world constraints are those inherited from properties of the real world, the laws of physics for example. Clearly, programs that interact with the real world, in particular cyber-physical systems, should observe these constraints. Software developers intend this to be the case, but software faults can occur that cause the software to violate the constraints.

Research efforts in this area have tended to focus on checking software’s consistency with specific types of real-world constraints, e.g. the consistent use of physical units and physical dimensions [CITE]. In prior research in which we sought a comprehensive approach, we introduced a new structure, the *interpreted formalism*, that combines: (a) the logic of the computation, i.e., the traditional notion of software with (b) the interpretation of the logic, i.e., details of how the software is “connected” to the real world. The interpreted formalism model provides a framework for analysis of the consistency of the system’s logic with the real-world entities with which the logic interacts. In the model, the consistency of physical units and of physical dimensions are special cases.

The interpretation component of a situated formalism is in machine readable form thereby allowing: (a) the precise definition of constraints derived from the real world, and (b) the use of several analysis techniques that enable automated checking of these constraints. We conducted a preliminary case study of interpreted formalisms on an open-source project, the Kelpie flight planner, that is approximately 13,000 lines of source code. This study illustrated the feasibility and potential benefits of the use of interpreted formalisms [CITE].

In this paper, we present a second case study designed to provide a more detailed assessment of the interpreted formalism concept. In this case study, the concept was applied to a system that provides a set of geographic services. The system, called OpenMap, is an open-source project with approximately 158,000 lines of Java source code. The authors have no connection to the OpenMap project beyond using it in this case study.

An interpretation was developed for the OpenMap software and an interpreted formalism created. Static analysis then revealed a substantial number of faults that violate real-world constraints. To the best of our knowledge, these faults were either unknown to the developers or were reported by users of the system after deployment. This second case study indicates that the interpreted formalism concept: (1) is feasible for large software systems, (2) is effective in fault detection, and (3) provides efficient support to reduce user effort.

The remainder of this paper is organized as follows. 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 almost all formal languages, including programming languages, are purely syntactic entities. Without an interpretation, they have no real-world meaning. Current software systems frequently document interpretations in an ad-hoc manner using meaningful identifiers, unstructured comments, and other documentation. This informal and unstructured approach leads to the possibility of: (a) the real-world semantics being defined incompletely, (b) connections between logic elements and the real-world entities with which they are associated being under specified, and (c) real-world constraints being violated by the logic.

A carefully defined interpretation documents the real-world meanings of logic elements in a precise manner. With an explicit, formal interpretation, important characteristics of real-world entities and the associated real-world constraints can be clearly defined, and the real-world constraints that the interpretation exposes can be checked automatically.

## Interpreted Formalism

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 the 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 mathematical 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 design is based upon the concept of *real-world types* [CITE]. An interpretation is: (a) a set of real-world types, and (b) a set of real-world type rules defined within the framework of a real-world type system. Real-world types specify characteristics of entities in the real world accessed by the software system, and real-world type rules specify the constraints that should be observed in the software system.

## Development of Interpreted Formalisms

In order to build an interpreted formalism for a software system of interest, three artifacts need to be developed: (1) a set of real-world types, (2) a corresponding set of real-world type rules, and (3) a set of bindings from real-world types to software entities. To facilitate development of interpreted formalisms, we have developed a *synthesis framework* that largely reduce the effort required in developing these artifacts [CITE]. Our first case study showed that the framework can reduce the effort required from users substantially.

# Fault Detection Based on Interpretation

Within a real-world type system, the real-world type rules document properties derived from real-world constraints. These type rules should be observed in software systems that manipulate real-world entities, and this requires:

* That program statements conform to static real-world constraints.
* That references from program elements to real-world entities are precise, consistent, and correct.
* That inevitable approximations in the values caused by hardware are accessible by users.
* That runtime values of program variables conform to real-world constraints.

Several analysis techniques were developed in order to establish these properties [CITE], specifically:

* Checking real-world constraint.
* Analysis of reasonable ranges of values for variables.
* Identification of locations within the source code that should be inspected for conformance to real-world constraints, i.e., targeted inspection.
* Generation of executable assertions to check constraints that are not statically checkable.

# Case Study

The goals of this case study were to assess the following in the context of a software system that is an order of magnitude larger than that used in our previous study:

* The feasibility of developing and applying interpreted formalisms.
* The effectiveness of the analysis techniques at detecting software faults.
* The effort level required to develop interpreted formalisms and apply them.
* Whether interpreted formalisms scale generally to the larger software subject system.

The case study was conducted using a toolset that: (a) implements all of the analyses described, (b) supports the creation and use of real-world type libraries, and (c) includes a framework to assist the user by partially synthesizing real-world type systems and the bindings from real-world types to elements of software. The toolset is described elsewhere [CITE, CITE].

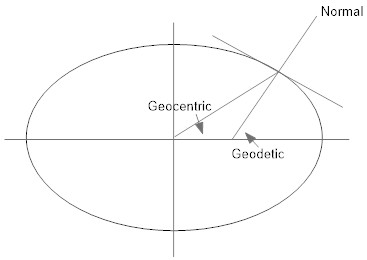
## Case Study Subject

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 1,193 source files.

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

**Units and dimensions**. The OpenMap software makes calculations involving distances, heights, speeds, angles, time and so on, and 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 faults. 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* is 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. This 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 the 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 demands high levels of accuracy.

A complete real-world type system was created for the OpenMap project. Real-world types were created for all of the 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. Details of the real-world type system are as follows:

* **Size**. 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. Clearly, bindings of real-world types to software entities cannot be reused, and so 1932 real-world type bindings were created for OpenMap.
* **Coverage**. Variables in 196 of the source files were bound to real-world types, and program elements in 232 source files accessed real-world types. The other source files do not interact with real-world entities, and so do not have real-world type bindings.

## Fault detection

After setting up the real-world type system, analyses were conducted on all 1193 source files using both real-world type (constraint) checking and reasonable range analysis. Table I summarizes the faults reported and the number of real faults.

Real-world constraint checking reported 53 faults from 18 source files of which 29 are false positives and 24 are real. Units checking revealed 49 faults of which 41 are false positives and eight are real. Reasonable range analysis reported 29 warnings from 18 source files of which 12 could lead to runtime faults.

1. Faults reported and real faults in OpenMap

|  |  |  |  |
| --- | --- | --- | --- |
| Analysis techniques | # of files with reported faults | # of faults reported | # of real faults |
| Units checking | 15 | 49 | 8 |
| Real-world constraint checking | 18 | 53 | 24 |
| Reasonable range analysis | 18 | 29 | 12 |

Table II summarizes the source files that contain real faults, the number of real faults, and the real-world semantics that caused the faults. Every real faulty statement in the software included one or more real-world semantic faults.

1. Real faults found by constraint checking

|  |  |  |
| --- | --- | --- |
| Program files | # of real faults | 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 |

Reasonable range analysis found 12 statements in 6 files. Table III summarizes them:

1. real faults found by reasonable range analysis

|  |  |  |
| --- | --- | --- |
| Program files | # of real faults | Possible runtime faults |
| 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 |

Real-world constraint checking reported 53 faults of which 24 are real faults. These non-real faults are categorized into two kinds: improper usage and false warning.

The definition of improper usage was introduced earlier [CITE] and refers to either: (a) a variable took on different real-world types but the same programming language type in different parts of the program, or (b) the elements of an array were not all of the same real-world type but were of the same programming language type. Both practices could easily lead to faults. Table IV 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 |

The real-world type system used in our first case study was partially created by the toolset’s synthesis system and was reused in this case study. That type system contains 35 real-world types and 97 real-world type rules. One additional real-world type was created for OpenMap.

Type bindings are system specific and so all bindings had to be created for the case study of OpenMap. Again, the toolset was able to synthesize many of these bindings. 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.

## Example Faults

In this section, we present examples of the faults that were identified by the analyses we describe. These examples illustrate the types of issues that arise through inconsistency of software with the real world.

**Example Fault 1.** Four real faults were found in the source file Road.java, all of which are misuse of units. The statement below contains two real faults:

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

prevPoint.getLongitude(),

thisPoint.getLatitude(),

thisPoint.getLongitude());

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

**Example Fault 2.** In source file TX7.java, one statement contains an inaccurate computation. This statement involves the 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 inaccurate.

**Example Fault 3.** In source file LOSGenerator.java three statements contain inaccurate computations. The inaccuracy is caused by the use of inconsistent *reference levels* of elevations. The three statements are similar to this statement:

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. 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, and so the computations are inaccurate.

Example faults 1, 2, and 3 were detected by real-world constraint checking. Example fault 4 was found by reasonable range analysis.

**Example Fault 4.** In the file CADRG.java, there is a possible *division of zero* in the statement:

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.

As noted above, real-world constraint checking and reasonable range analysis identified constructs in the source code that were not faults. These *non-real* faults were examined and found to fall into two broad categories: (1) *improper usage*, and (2) *false warnings*.

Most structures identified as improper usage derive from statements that are similar to the following:

lat = Math.toRadians(lat);

lon = Math.toRadians(lon);

On the left side of the assignments, variables lat and lon are latitude and longitude values in radians, but the two variables represent values in degrees on the right side. Essentially, lat and lon have different real-world types in the same statement.

Structures identified as false warnings frequently involve conversion between different real-world types. For example, two false warnings were reported in these statements:

double lambda = lon \* Degree;

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

In the first statement, variable lon, longitude in radians, is assigned to variable lambda which represents longitude measured in degrees. The second statement is similar. Detection of unit conversion in source code such as this has been studied by ?????? [CITE]

Improper usage and false warnings indicate fault-prone operations. Programmers should double check these statements to make certain that the entities referenced are being used correctly.

# Conclusion

We need a 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.
2. reference
3. reference
4. reference
5. reference
6. reference
7. reference
8. reference
9. reference
10. reference
11. reference
12. reference
13. reference
14. reference