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— **The actions taken by software should be consistent with constraints arising in the real world. For example, computations should not mix physical units unless intended. To enable checking of real-world consistency, in previous work we introduced: (a) a new structure, the interpreted formalism, that is the software analog of the notion of interpretation from classical logic, and (b) a practical implementation of the concept, real-world type systems. We reported preliminary results of the potential value of interpreted formalisms in improving software dependability. In this paper, we present details of a new case the results of which indicate that: (a) interpreted formalisms can be applied to large software systems, and (b) the fault-detection potential is substantial.**

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

I. 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 [4],[9]. In prior research [14][15] 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[11], that is approximately 13,000 lines of source code. This study illustrated the feasibility and potential benefits of the use of interpreted formalisms [14].

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 [12], 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.

II. Interpretation and Interpreted Formalism

A. 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, rigorous 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.

B. 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.

C. 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* [14]. 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 by the software system.

D. 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 the development of interpreted formalisms, we have developed a *synthesis framework* that largely reduce the effort required in developing these artifacts [15]. Our first case study showed that the framework can reduce the effort required from users substantially.

III. 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 [14], specifically:

- Checking real-world constraints.
- 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. We refer to this as targeted inspection.
- Generation of executable assertions to check constraints that are not statically checkable.

IV. 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 [15].

A. 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 semantics 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 within the support toolset by default.

Geographic and geocentric latitude. The real-world entity *latitude* is used widely 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.

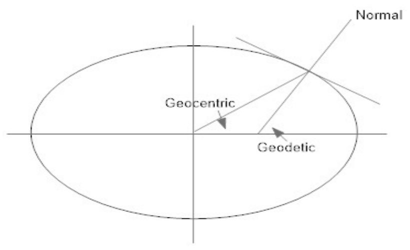

Fig. 1. Two different types of latitude

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 the Kelpie flight planner (previous case study) was reused in this case study. The 35 real-world types and 97 real-world type rules were reused. One additional real-world type was created for OpenMap. Clearly, bindings of real-world types to software entities cannot be reused, and so 1932 real-world type bindings were created for OpenMap. The toolset was able to synthesize 803 (41.6%) type bindings after 1129 (58.4%) were seeded by hand.
- **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.

B. *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 24 were real and 29 were false positives. Reasonable range analysis reported 28 warnings from 18 source files of which 12 could lead to runtime faults and 16 were false positives.

Table I 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.

TABLE I. 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 faults in 6 files. Table III summarizes the faults:

TABLE II. REAL FAULTS FOUND BY REASONABLE RANGE ANALYSIS

Program files	# of real faults	Possible runtime faults
CADRG.java	1	Division by 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 by zero
MercatorUVGCT.java	4	Infinite bound

The false positives indicated by real-world constraint checking are divided into two categories, *improper usage* and *false warning*, both of which are potentially useful. The definition of improper usage was introduced earlier [14] and refers to either: (a) a variable taking on more than one real-world type but the same programming language type in different parts of the program, or (b) the elements of an array having different real-world type but the same programming language type. Both practices could easily lead to faults. Table III summarizes the improper usage and false warnings found.

TABLE III. FALSE WARNINGS AND IMPROPER USAGE

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

C. *Example Faults*

In this section, we present examples of the faults identified by the analyses we describe illustrating 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());
```

The first fault is that `GreatCircle.sphericalDistance()` expects the units for the parameters to be `radians`, but the arguments in this statement are all measured in `degrees`. The second fault is that the return value of the function is an angle in `radians`, which is inconsistent with the variable `kilometers`.

Example Fault 2. In source file `TX7.java`, one statement uses Earth's *radius* incorrectly. 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. 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, the variable `wgs84_earthEquatorialRadiusMeters` represents Earth's equatorial radius with the Earth modeled as an ellipsoid.

Example Fault 3. In source file `LOSGenerator.java` three statements contain inaccurate computations caused by the use of inconsistent *reference levels* of elevation. 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 the 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.

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

```
dlon = lon2 - lon1;
…
deltaDegrees = dlon;
…
ret = pixPerDegree / (deltaPix / deltaDegrees);
```

The variable `deltaDegrees` represents the difference between two longitude values, which could be zero.

As noted above, we categorize false positives in the analysis as either improper usage or 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.

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 other research work [9]. Improper usage and false warnings indicate fault-prone operations and are worth checking to make sure that the entities referenced are being used correctly.

V. Related Work

The interpreted formalism is a new concept that models the relationship between the real world and the machine world. Other researchers have modeled the relationship [1],[5],[9],[13].

Units consistency has been explored in different languages [4],[9]. The interpreted formalism introduces general analysis opportunities to check real-world constraints comprehensively. Units and dimensional analyses are special cases of this comprehensive analysis.

The realization of the interpreted formalism builds on type theory. Pluggable type system and dependable type systems are type systems support checking additional type rules [2][3][7].

VI. Conclusion

This paper presents a case study of the interpreted formalism concept by applying to a large open-source project. This case study evaluated the performance of the interpreted formalism concept in feasibility, fault detection, and effort level. The results of this case study suggest that (1) the interpreted formalism is fit for large software systems, (2) error checking techniques are effective, and (3) the synthesis framework greatly reduces the effort required by users.

References

[1] Bhave, B., B. H. Krogh, D. Garlan and B. Schmerl. "View Consistency in Architectures for Cyber-Physical Systems." In Proceedings of the 2011 IEEE/ACM International Conference on Cyber-Physical Systems (ICCPS), Chicago, 2011, 151-160. IEEE Computer Society, 2011.

[2] Bove, A. and P. Dybjer. 2009. "Dependent types at work". In Language Engineering and Rigorous Software Development, edited by Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto. 57-99. Springer, 2009.

[3] Dietl, W., S. Dietzel, M. Ernst, K. Muşlu, and T. Schiller. 2011. "Building and using pluggable type-checkers." In Proceedings of the 33rd International Conference on Software Engineering (ICSE). Waikiki, Honolulu, 681-690. ACM Press, 2011.

[4] Hangal, S., and M. S. Lam. 2009. "Automatic dimension inference and checking for object-oriented programs." In Proceedings of the 31st International Conference on Software Engineering (ICSE). 155-165. IEEE Computer Society, 2009.

[5] Gunter, C. A., E. L. Gunter, M. Jackson, and P. Zave. 2000. "A Reference Model for Requirements and Specifications." IEEE Softw. 17, 3, 37-43. IEEE, 2000.

[6] International System of Units, National Institute of Standards Technology, Washington, DC.

[7] Markstrum, S., D. Marino, M. Esquivel, T. Millstein, C. Andreae, and J. Noble. 2010. "JavaCOP: Declarative pluggable types for java." In ACM Trans. Program. Lang. Syst. 1-37. ACM press, 2010.

[8] Mars Climate Orbiter Mishap Investigation Board Phase I Report, 1999. National Aeronautics and Space Administration, Washington DC, November 10, 1999.

[9] Jackson, M. 2000. "Problem Frames: Analyzing and Structuring Software Development Problems." Boston, Addison-Wesley Longman Publishing Co., Inc., 2000.

[10] Jiang, L. and Z. Su. 2006. "Osprey: a practical type system for validating dimensional unit correctness of C programs." In Proceedings of the 28th international conference on Software engineering (ICSE). Shanghai, 262-271. ACM Press, 2006.

[11] Kelpie flight planner for FlightGear. http://sourceforge.net/projects/fgflightplanner/

[12] OpenMap. http://openmap-java.org/

[13] Parnas, D. L. and L. Madey. 1995. "Functional documents for computer systems." In Sci. Comput. Program. 41-61. Amsterdam: Elsevier North-Holland, Inc., 1995.

[14] Xiang, J., J. Knight, and K. Sullivan. 2015. "Real-world Types and Their Application". In Proceedings of the 34th International Conference on Computer Safety, Reliability and Security (SAFECOMP). Delft, 2015, 471-484. Springer, 2015.

[15] Xiang, J., J. Knight and K. Sullivan, 2016. "Synthesis of Logic Interpretations," In Proceedings of the 17th International Symposium on High Assurance Systems Engineering (HASE), Orlando, FL, 2016, pp. 114-121.