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 relevant constraints arising in the real world. For example, computations should not mix values that are understood as being expressed in incompatible physical units. To enable checking of such consistency our previous work introduced: (a) a new structure, the interpreted formalism, that is a software analog of the notion of an informal interpretation from classical logic, and (b) a practical implementation of the concept in the form of real-world type systems. We reported preliminary results of the value of interpreted formalisms in improving software dependability. In this paper, we present details of a new case study, the results of which indicate that: (a) interpreted formalisms can be applied to large 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 are an example. Clearly, programs that interact with the real world, in particular cyber-physical systems, must respect such constraints. Software developers intend this to be the case but often lack mechanisms to document and check such consistency. Software faults that cause software to violate such constraints can thus go unnoticed.

Research in this area has tended to focus on checking software's consistency with very specific types of real-world constraints, e.g. the consistent use of physical units and physical dimensions [4],[9]. In prior research [14][1] 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., an explicit representation of the correspondence of software elements with the real world. The interpreted formalism model provides a framework for analyzing the consistency of software logic with the real-world entities with which the logic interacts. Consistency of physical units and of physical dimensions are special cases.

The interpretation component of an interpreted formalism is machine readable, 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 study.

We developed an interpretation for OpenMap and then used it to analyze the code. This work 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 case study indicates that the interpreted formalism concept: (1) is feasible for large systems, (2) is effective in fault detection, and (3) provides efficient support to reduce user effort.

II. Interpretation and Interpreted Formalism

A. Explicit Interpretation

Elements in expressions written in formal languages, including programming languages, are purely syntactic. Without interpretations, they have no real-world meaning. In current programming practice, interpretations are generally documented only in an informal, incomplete, non-computable manner, e.g., relying on identifiers, comments, and other documentation. Such an approach leads to the possibility of: (a) real-world semantics being defined and understood incompletely, (b) connections between software and real-world entities being underspecified, and (c) real-world constraints being violated by software 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 is a two-tuple comprising software logic and an associated interpretation. The software is defined in a manner appropriate for a system of interest. The choice of programming language, coding standards, compiler, and so on, are unaffected by the interpreted formalism structure. The key difference is the addition of an explicit interpretation.

In our view, in the development of a particular system the task is no longer just to develop software logic, but instead is to develop an interpreted formalism, i.e., both the logic and an explicit interpretation. Without the explicit interpretation, whatever would be developed as "software" runs the risk of failing to satisfy the desired correspondence with the real world correctly, which is the entire purpose of the software system.

C. *Realization: Real-World Type System*

The concept of an interpretation is well established in logic. It can take two forms: a mapping from logical terms to elements of a formal domain (e.g., a mapping of the variables of a Boolean formula to the domain {*true, false*}), or a mapping from logical to the *empirical* real world (e.g., as in the interpretation of the proposition, *some swan is black*, as being about the real world). Our work is based on the empirical notion of interpretations.

The problem we face is that defining the content and structure of an effective and complete interpretation for practical software systems 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, (b) a set of type rules, and (c) an association of logical elements in code with real-world types. 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, four artifacts need to be developed: (1) the traditional software logic, (2) a set of real-world types, (3) a corresponding set of real-world type rules, and (4) a set of bindings of software entities to real-world types. To facilitate the development of interpreted formalisms, we have developed a *synthesis framework* that largely reduces the effort required in developing these artifacts [1]. Our first case study showed that the framework can substantially reduce the effort required from users.

III. Fault Detection Based on Interpretation

Within a real-world type system, 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. This requires that:

- program statements satisfy real-world constraints;
- references from program elements to real-world entities are precise, consistent, and correct;
- approximations in logical values caused by hardware and discrete sampling are accounted for in an interpretation;
- and that runtime values satisfy real-world constraints.

We developed several analysis techniques in order to establish these properties [14], including:

- checking real-world constraints;
- analysis of reasonable ranges of values for variables;
- identification of locations within the source code that should be targeted for inspection for conformance to real-world constraints;
- 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 practicality of interpreted formalisms;
- the effectiveness of analysis for detecting faults;
- the effort required to develop interpreted formalisms;
- whether interpreted formalisms scale to larger systems.

We conducted a case study 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 [1].

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 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, including the following.

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, as illustrated in Fig. 1. This difference is crucial when the shape of Earth is modeled as an ellipsoid.

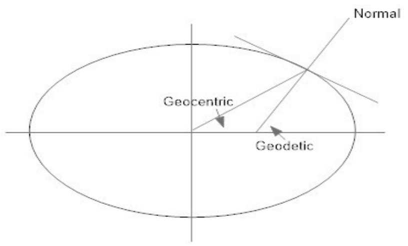

Fig. 1. Two different types of latitude

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

We created a complete real-world type system for OpenMap. We created real-world types for all real-world entities accessed by the software. Variables and methods that access real-world entities were bound to their real-world types. We defined type rules to document relevant relationships between real-world entities. Details of the real-world type system are as follows.

- **Size**. We reused the real-world type system created for the Kelpie flight planner (previous case study), including 35 real-world types and 97 rules. We created one new real-world type for OpenMap. Bindings of real-world types to software entities cannot be reused, so we created 1,932 real-world type bindings for OpenMap. The toolset was able to synthesize 803 type bindings (41.6%) after 1,129 binding (58.4%) were seeded by hand.
- **Coverage**. Variables in 196 source files were bound to real-world types. Program elements in 232 source files accessed real-world types. The other source files did not interact with real-world entities, and needed no bindings.

B. Fault Detection

For simplicity, we use the term fault in this paper although the analysis we describe can detect *potential* faults rather than actual faults. Although very unlikely, violating real-world constraints might be the programmer's intent. In practice, the detected faults are of two types: (1) violations of real-world constraints (type A), and (2) unexpected uses of real-world types (type B). The latter are consistent but potentially confusing uses such as arise with implicit type conversion.

After setting up the real-world type system, we analyzed all 1,193 source files using both real-world constraint and reasonable range analysis checking. Table I summarizes the faults reported and the number of type A faults.

Real-world constraint checking reported 53 faults from 18 source files of which 24 were type A and 29 type B. Reasonable range analysis reported 28 warnings from 18 source files of which 12 could lead to runtime faults and 16 were type B.

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

TABLE I. REAL FAULTS FOUND BY CONSTRAINT CHECKING

Program files	# type A 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	# type A 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 type B faults indicated by real-world constraint checking are divided into two categories, *improper usage* and *false warnings*, 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 types but the same programming language types. 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 type A faults were found in the source file `Road.java`, all of which are misuse of units. The statement below, for example, contains two type A 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 by 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 type B faults 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 results from a test of the practicality and utility 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.