Real-World Types and Their Application

Jian Xiang, John Knight and Kevin Sullivan

University of Virginia, PO Box 400740, Charlottesville, VA 22904, USA

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

Abstract. Software systems sense and affect real world objects and processes in order to realize important real-world systems. For these systems to function correctly, such software should obey constraints inherited from the real world. Typically, neither important characteristics of real-world entities nor the relationships between such entities and their machine-world representations are specified explicitly in code, and important opportunities for detecting errors due to mismatches are lost. To address this problem we introduce real-world types to document in software both relevant characteristics of real-world entities and the relationships between real-world entities and machine-level representations. These constructs support specification and automated static detection of such mismatches in programs written in ordinary languages. We present a prototype implementation of our approach for Java and case studies in which previously unrecognized real-world type errors in several real systems were detected.

Keywords: real-world types, dependability, static analysis

1 Introduction

Software systems, especially cyber-physical systems, interact with the real world in order to sense and affect it. Such software should obey rules inherited from the real world that it senses, models, and affects, as well as from the machine world in which the software executes. For example, software that computes physical quantities should respect units and physical dimensions, and many analysis systems have been built to perform units and dimensions checking on software.

Frequently, however, important characteristics of real-world entities are undocumented or are documented incompletely, informally, and implicitly in code. As a result, constraints inherited from the real world are stated and enforced either in ad-hoc ways or not at all. Crucial relationships between machine-world representations and real-world entities remain under-specified, and programs treat machine-world representations as if they were the real-world entities themselves. As a result, faults are introduced into systems due to unrecognized discrepancies, and executions end up violating constraints inherited from the real world. The results are software and system failures and adverse downstream consequences.

The failure of software to represent real-world entities with sufficient fidelity has been a causal factor in various accidents and incidents. In 1999, the Mars Climate

Orbiter was lost because different parts of the software system used different units of measurement [1]. More recently, in 2013 a delay in berthing Orbital's Cygnus spacecraft with the International Space Station (ISS) occurred because Cygnus and the ISS used different forms of GPS time data; one based on the original 1980 ephemeris, and the other based on an ephemeris designed in 1999 [2]. In both cases, different real-world entities were represented using the same machine-world types, allowing machine-world operations to be applied in ways that made no real-world sense.

In this paper, we introduce the concept of *real-world types* and *type checking*. For entities of a given real-world type, the type definition documents the real-world specification, the machine representation, and the relation between the two. The specification of a real-world type defines relevant, observable properties of real-world entities of that type. The machine representation defines how a real-world entity is represented in the machine and thus becomes accessible by software. The relationship defines the connection between real-world entities and associated machine elements.

The goal of real-world types is to enable reasoning about and automated checking of the logic of a program based on the real-world entities with which the program has to interact. To exploit real-world types, we introduce the concept of connecting them with entities in programs such as variables, constants and functions so as to extend the programmer designated type. The explicit, formal documentation of real-world types supports analysis and enforcement of real-world constraints on programs in a systematic way, thereby enabling new classes of software fault prevention and detection, without requiring programmers to adopt new machine-level programming languages. Among other things, real-world types can be used in effect to annotate uses of machine-level values in ways that enable prevention of the kind of real-world type mismatches that led to such failures as that of the Mars Climate Orbiter.

If real-world types are to be used in the development of realistic software systems, an approach to integrating them into widely used languages and development methods is needed. We have developed a prototype demonstrating the integration of real-world types with the Java language. The implementation connects real-world type definitions to Java entities without modifying the Java source program and provides type checking of Java code against real-world type rules defined by system experts. We present case studies in which real-world types were applied to a set of pertinent projects including a project with 14,000 LOC. The real-world type checking revealed both unreported faults and faults that had previously been reported as bugs.

In the next section, we discuss the role of real-world entities in software, and in section 3 we present the real-world type system concept. In section 4 we discuss our prototype implementation for Java. Our evaluation of real-world types is presented in section 5. Related work is summarized in section 6 and we conclude in section 7.

2 From the Real World to the Machine

Frequently, software systems are developed with an emphasis on the machine, focusing on the machine context rather than the real-world entities that the machine has to manipulate. We refer to this as the *machine-world* viewpoint. Variables, values, or

instances, i.e., machine-world entities, are introduced to represent real-world entities, thereby also introducing uncertainties and assumptions about the real-world entities in those representations.

That real-world entities are what the computer ultimately is *intended* to manipulate is often forgotten. The problem is that machine-world entities are not the same as real-world entities. The relationships between them are usually incomplete because of: (a) the absence of much of the real-world semantic information in the machine world, (b) the approximations inherent in finite-precision representations in the machine world and in the sensing of values from the real world, (c) noisy and mis-calibrated sensors, and (d) sensor and related failures. However, by focusing on the real-world context and documenting comprehensively the relationship between the real-world entities and the associated machine-world entities, i.e., taking a *real-world* viewpoint, the incompleteness and other limitations in the relationships become explicit and analyzable. Software design and analysis should be fully informed by the real-world context if it is to be rigorous and complete. The contrast between the real-world and the machine-world viewpoints is illustrated in Figure 1.

This real-world viewpoint has several benefits: (a) the precise semantics of the real world including all relevant attributes and invariants are clear; (b) the rules and relationships between real-world entities are explicit and can be used to check the software for violations of the rules and relationships; (c) the differences between real-world entities and their machine-world realizations can be documented and analyzed; (d) specific analysis techniques derived from the real-world context can be developed.

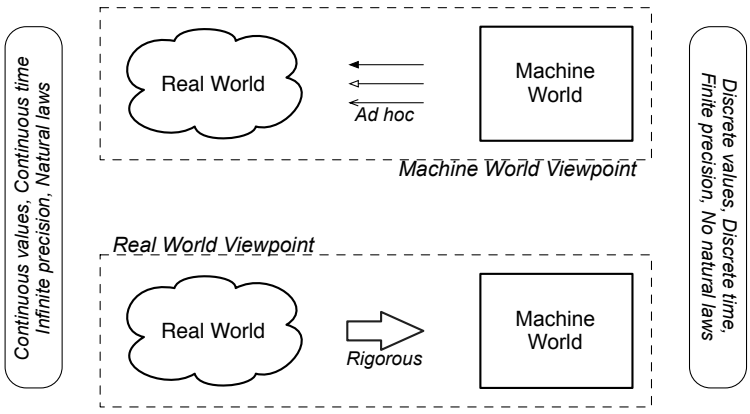

Fig. 1. From a machine-world viewpoint to a real-world viewpoint

Real-world types are designed to document real-world entities, and to document and leverage the relationship between real-world entities and programs. They provide a pragmatic and robust infrastructure for error analyses. Many characteristics of real-world entities define constraints that are unlikely to change yet should be observed in any applications that access the pertinent real-world entities. Therefore, a library of generic real-world types and rules could be established for use in various different applications.

3 The Real World Type System

3.1 Overall Structure

The overall structure of the real-world type system and its interfaces with: (a) the real world, (b) the computation of the associated system, and (c) overall system design are shown in Figure 2.

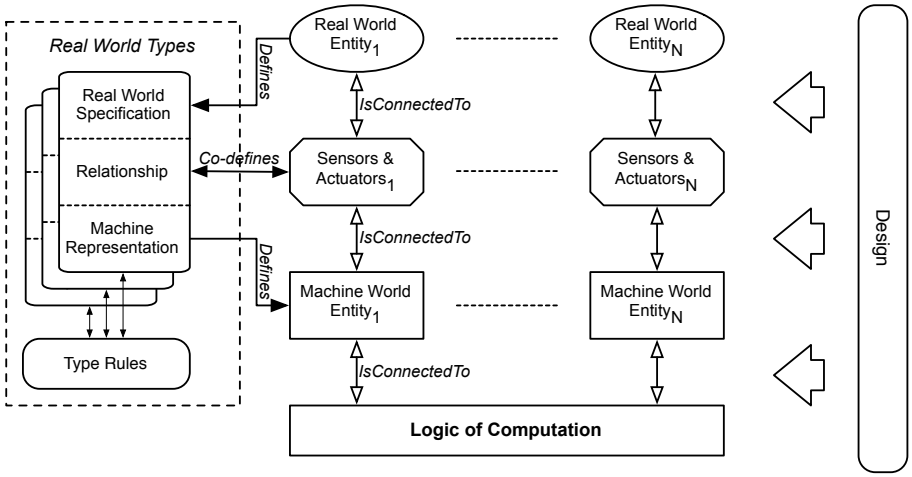

Fig. 2. The real-world type system structure

By *logic of computation*, we mean the sequence of logical actions defined by software in a computer. By *design*, we mean the overall system design activity that determines many characteristics of the computing system. Clearly, design affects which real-world entities will be involved in the computation, the required precision of computations, details of timing, and so on.

In the overall structure, system designers define the specifications of the real world types based on real-world entities of interest. The representations of real-world types are determined by design and are used to define the machine-world entities that will represent real-world entities in the computation. The relationships between the specifications and representations are determined by design and analysis, including details of the sensing and actuation needed to interface the computation to the real world. Sensing and actuation can contribute to the definition of the relationships because of limitations inherent in the associated equipment such as sensing precision and rate of update. Finally, the type rules associated with the real-world types are determined by the semantics of the operations that are required for the entities being manipulated.

3.2 The Definition of Real-World Types

A real-world type has a name and a definition that consist of three parts:

- The specification of the type derived from the associated real world entities.
- The machine representation of instances of the type.
- The relationship between the specification and the representation.

We present details of these three parts in the remainder of this section.

Real-world Specification

The first part of the specification is an explication of the type intended for humans. An explication is a detailed explanation of the meaning of something, and explications are required for real-world types so as to ensure that a single source of meaning is provided for all the entities with which the computing system interacts. The explication could be in natural language, one or more formal languages, or a combination. The explication is the means by which interpretation is given to the real-world type.

As an example, consider the notion of altitude in an avionics system. Altitude could mean height above local ground level or height above mean sea level, and could be determined by radar, barometric pressure, or GPS. Exactly how each semantic concept is used for this type needs to be explained in an explication for altitude.

The second part of the specification is a set of real-world semantics. A real-world semantic concept can be any real-world property of interest. Every semantic concept is defined through: (a) an explication of the concept, (b) the set of values that the concept can take, and (c) reference sources such as an ontology or a dictionary.

Returning to the example of an aircraft's altitude, a semantic concept is the measurement reference level. This concept could be either sea level or local ground level. Documentation for the specification of such a concept is shown in Table 1.

Table 1. Example real-world semantic concept

Concept:	Reference level
Explication:	Reference datum from which altitude value is measured
Possible values:	Mean sea level; local ground level
References:	Basic Geo Vocabulary; DAML's location ontology

A complete set of semantics for altitude would include reference level, frame of reference (surface location, Earth center, etc.), units of measurement, etc.

Units and physical dimensions are examples of real-world semantics, and their introduction into programming languages along with analysis techniques to perform type correctness checks have been explored previously [3, 4, 5]. In our theory of real-world types, units and dimensions are just special case semantics and are predefined because of their widespread use and importance in real-world properties.

Units can be enumerated as needed by an application. The dimensions semantic concept consists of the seven basic dimensions of physics (mass, length, time, electric current, temperature, luminosity, and amount of substance) [6]. The existence of this semantic concept allows the standard dimensional analysis of physics to be applied.

For simplicity, in our own use of dimensional analysis we added *angle* to the set for a vector length of eight. Thus, a semantic value of dimensions is an eight-element vector of integers defining the real-world dimensions of the associated variable. Some example dimensions are:

- Speed: (0,1,-1,0,0,0,0,0)
- Acceleration: (0,1,-2,0,0,0,0,0)
- Energy: (1,2,-2,0,0,0,0,0)

Machine Representation

The machine representation of a type is characterized by a set of semantics that describe the properties derived from the machine context. Machine-world semantics in the representation use a similar format to that used for the real-world semantics in the specification.

Some examples of machine-world semantics and the associated values that they can take are:

- encoding: integer, floating point, double
- mutability: mutable, non-mutable

The mutability semantic concept indicates whether objects of the type are constant, thereby allowing for detection of unintended assignments.

Relationship

The relationship that connects the specification to the machine representation of a real-world type is defined as a logical expression. The expression typically has the form of a conjunction of predicates where each predicate indicates one dimension of imprecision. In the altitude example, the mapping might take the form:

```
value: error < delta and delay < tau
```

The first predicate documents the maximum difference between the actual altitude and the value supplied by the sensing system, and the second documents the maximum delay between sensing the altitude and the associated value being available in the machine world.

3.3 Real-World Type Rules

Real-world type safety is determined by rules derived from real-world entities. All necessary checking of the predefined semantics (units and physical dimensions) are included by default. Such analysis is limited though useful, and the general notion of real-world types presented here provides a broad basis for defining type safety rules. Any relevant limitation, expectation or invariant that arises in the real world can be encoded as type rules provided it can be expressed using real-world type semantics.

Developers define type rules based on the semantics of types and the desired effect on semantics of operations by programs. In arithmetic expressions, for example, units must match, the dimensionality rules of physics must be observed, arithmetic operations can only be applied to types for which they are defined, and the results of arithmetic operations must have the correct real-world type.

Example type rules that detect operations which are probably erroneous include:

- The units of an angle and a latitude must match if they are added. The result is of type latitude measured in the same units.
- A velocity, dimensions (0,1,-1,0,0,0,0,0), cannot be added to a distance, dimensions (0,1,0,0,0,0,0,0).
- A latitude or a longitude cannot be added to a latitude or a longitude.
- An x coordinate in one frame of reference cannot be used in any arithmetic operation with a coordinate from a different frame of reference.
- A variable of type *magnetic* heading cannot be used in an expression expecting a variable of type *true* heading, even if both are represented as integers and are commensurable.
- A variable of type *geodetic* latitude cannot be used in an expression expecting a variable of type *geocentric* latitude, even if both are represented as floating point and are commensurable.

As an example of type-rule definition, consider the semantics of the result of subtracting two operands of type `vertical_cartesian_axis`, e.g., for calculating the altitude difference between two points in the same Cartesian coordinate system. The definition is illustrated in Fig. 3.

$$\frac{e1:\left\{\begin{array}{c}unit{:}feet\\ Dimension{:}length\\ coordinate\_sys{:}cartesian\\ technology{:}radar\\ offset\_origin{:}mean\ sea\ level\\ geometry\_plane{:}\ vertical\\ axis{:}z\end{array}\right\} \quad e2:\left\{\begin{array}{c}unit{:}feet\\ Dimension{:}length\\ coordinate\_sys{:}cartesian\\ technology{:}radar\\ offset\_origin{:}mean\ sea\ level\\ geometry\_plane{:}\ vertical\\ axis{:}z\end{array}\right\}}{e1-e2:\left\{\begin{array}{c}unit{:}feet\\ Dimension{:}length\\ axis{:}z\\ geometry\_plane{:}\ vertical\\ coordinate\_sys{:}cartesian\end{array}\right\}}$$

Fig. 3. Example type rule definition. The notation e:T denotes a type judgment (e is of type T), and the overall construct defines an inference rule defining the type of the result of applying a specific operator, here subtraction, to operands, e1 and e2, of the specified types

3.4 Real-World Types and Program Structures.

The notion of type in programming languages includes structures such as arrays and records. An array of integers and a record with multiple fields are types with various associated usage and equivalence rules.

Real-world types do not have any structure beyond that discussed so far. The reason is that including all of the structures that arise in programming languages is nei-

ther possible nor necessary. Defining real-world types as presented above and using them as building blocks in language structures provides the necessary structure. For a record, for example, a real-world type would be defined for the record as a whole and separate real-world types could be defined for each field. The type for a given field is then the *union* of the type information for the record and the type information for that field. Nested records accumulate type in the union as each level is progressed.

This approach raises an issue with structured types, because different instances of a user-defined type might have different real-world types. A record structure might be instantiated more than once to hold information with different real-world semantics but identical structures. This issue is dealt with in our present theory of real-world types by associating different type information with different instances of a structure but requiring that the components of a structure have the same type information in all instances. This restriction might be relaxed in future to allow more flexibility.

3.5 Real-World Type Example

An example of a real-world type is a point in three-dimensional space, `3Dloc`. Measurements designed to locate a point are only relevant if the associated coordinate system is defined completely. If multiple coordinate systems are in use in a program they must be distinguished. Thus, the real-world type information associated with an instance of the class needs to document the different aspects of the coordinate system.

The type definition for the coordinate system of `3Dloc` could be:

```
geographic_cartesian_coord_sys:
  Specification
    explication             : <text>
    real_world_semantics
      coordinate_sys_type   : cartesian
      target_space          : Earth
      origin                : center of mass of Earth
      dimensionality        : 3
      earth_model           : spheroid
      x_axis_orientn        : positive toward 0 degrees longitude
      y_axis_orientn        : positive toward 90 degrees east longitude
      z_axis_orientn        : positive northward
  Representation
    machine_semantics
      representation        : record structure – (float, float, float)
  Relationship              : <null>
```

Note that this type definition is created just to *distinguish* coordinate systems. Separately, we need the types of the three fields that will be used for a point in the coordinate system. For this, one, two, or three different type definitions might be needed. For this example, we assume that the `x` and `y` variables can share a type definition and a second definition is used for `z`. For `x` and `y`, we define the following type:

```
horizontal_cartesian_axis:
  Specification             : <text>
    real_world_semantics
      linear_units          : mile
```

```
      dimensions           : (1, 0, 0, 0, 0, 0, 0, 0)
      technology           : GPS
      geometry_plane       : horizontal
  Representation
    machine_semantics
      representation       : float
      mutable              : no
  Relationship
      value_error < delta1 and delay < tau1
```

In this example, altitude is part of a complete reference frame with origin at the center of mass of the Earth but with a presumed offset to mean sea level:

```
vertical_cartesian_axis:
  Specification
    explication            : <text>
    real_world_semantics
      linear_units         : feet
      dimensions           : (1, 0, 0, 0, 0, 0, 0, 0)
      technology           : radar
      geometry_plane       : vertical
      offset_origin        : mean sea level
  Representation
    machine_semantics
      representation       : float
      mutable              : no
  Relationship
      value_error< delta2 and delay < tau2
```

Such a type might be used to hold data in any space of interest. For example, the type could be used to hold location information for aircraft, climbers, balloons, etc.

4 An Implementation for Java

We have developed a prototype implementation of real-world types for Java. A choice made for the overall approach was that the system should operate without requiring changes to the subject Java program. This choice provides three major advantages:

1. Real-world type information does not obscure the basic structure of the Java program.
2. Real-world type information can be added to existing Java programs without having to modify (and possibly break) the original source text.
3. Real-world type information can be added to a Java program without impeding development of the Java program itself.

Motivated by this design choice, the prototype operates separately from the compiler via its own user interface.

4.1 Operation of the Prototype

The subject Java program is parsed, and real-world type definitions and type rules are either loaded from a file or entered via the user interface. Using the type-checking system requires that relevant Java program variables be annotated with appropriate real-world types. Entities just in the machine world, such as loop counters, are not annotated. Annotations are established by clicking on the type and then on the variable. Various displays are available to allow the real-world types and their use to be viewed, e.g., all of the Java variables of a given real-world type can be displayed, all of the type rules associated with a given real-world type can be displayed, etc.

Annotations can be limited to subsets of variables that are of interest. Entities with no annotation are assumed to be type correct in all circumstances. This permits incremental introduction of annotations as use of real-world types increases.

Real-world type checking involves examination of the Java program looking for violations of the type rules. The system provides for subsets of the type rules to be checked separately if desired. For example, separate checking of just the rules for units might be useful. This mechanism is also useful in debugging type rules.

Running the type checker yields diagnostics if type rules are violated. Diagnostics are presented with details of the type rule and the associated text is highlighted.

4.2 Type Conversion

An important issue in the type system is type *conversion*. For example, a conversion from feet to inches requires multiplying a variable storing a value in feet by 12. Explicit type conversion is dealt with simply by including type rules associated with whatever special operator or function is used. Implicit type conversion is more difficult. Conversions between real-world types can be syntactically simple. The difficulty is locating such conversions automatically without generating false negatives.

We deal with implicit type conversion by requiring that the programmer mark implicit type conversions as such. Thus, diagnostics generated for type conversions of which the type system was unaware require that the programmer suppress the diagnostic by indicating that there is an expected type conversion thereby indicating that the diagnostic has been investigated and the code found to be as desired.

5 Evaluation

We conducted a two-part study in which we developed real-world types for several open-source projects with which we have no association. In the first part, a complete set of real-world types were defined for a project called the *Kelpie Flight Planner* [7]. Various elements of the software were given real-world types, a set of type rules defined, and type checking was performed.

In the second part, we reused real-world types and type rules created in part one on a set of projects that access the same real-world entities. For these projects, type checking has only been applied to pertinent packages and files to detect errors.

5.1 Kelpie Flight Planner

The Kelpie Flight Planner is an open-source Java project based on Flightgear [8]. The program uses the airport and navaid databases of Flightgear to determine routes between airports based on user inputs. The program is 13,884 lines long, is organized as 10 packages, and is contained in 126 source files.

A critical element of the data used by the Kelpie Flight Planner in modeling aircraft movement, the *velocity surface,* is a two-element vector consisting of the horizontal velocity (motion across the Earth's surface) and the vertical velocity (climb or sink rate) of the aircraft. The details of the velocity surface are shown in Figure 4.

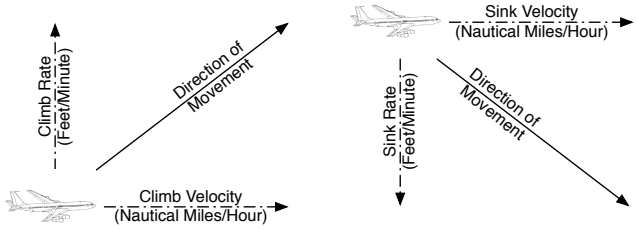

Fig. 4. The velocity surface

Various models of the Earth's geometry have been created, including a sphere and an ellipsoid. For the ellipsoid, different models have been developed for special purposes; for example, the International Geomagnetic Reference Field IGRF [9] and the World Magnetic Model WMM [10]. In order to undertake useful calculations, programs like the Kelpie Flight Planner have to operate with a model of the Earth's geometry, and details of the model need to be included in the real-world type system in order to allow appropriate checking.

For the Kelpie Flight Planner, 35 different real-world types were defined along with 97 type rules. The total number of type links for the project was 255. For illustration, we summarize three of six faults that were identified by the real-world type analysis. None of these faults had been reported in the project error log.

The source code containing the first fault is:

```
alt -= legTime * plan.getAircraft().getSinkSpeed()/60;
```

The expression references the wrong data. `getSinkSpeed()` returns a quantity measured horizontally and `alt` is measured vertically.

The source code containing the second fault is:

```
alt += legTime * plan.getAircraft().getClimbRate()/60;
```

`plan.getAircraft().getClimbRate()` returns the climb rate in feet/minute, the variable `legTime` is time in hours, and `alt` is altitude in feet. The conversion factor is 60, but the conversion requires multiplication by 60, not division.

The source code containing the third fault is:

```
alt -= legTime * plan.getAircraft().getSinkSpeed()/60;
```

The expression references the wrong data. As in the first fault, `getSinkSpeed()` returns a quantity measured horizontally and `alt` is measured vertically. Correcting this fault yields code with the same units issue as arose in the second fault requiring an additional fix.

5.2 Other Java Applications

We reused the real-world types and type rules created for the Kelpie Flight Planner project to check packages and source files in other applications with pertinent functions. We chose applications for which a log was available for defects reported from the field after release. We include examples here for illustration.

OpenMap is a Java Beans based toolkit for building applications and applets that access geographic information [11]. The code for the first fault we detected is:

```
lon2 = ll2.getY();
```

The variable `lon2` is a longitude, but the method `getY()` returns a latitude.

The code for the second fault is:

```
double[] llpoints = GreatCircle.greatCircle(startLLP.getY(),
  startLLP.getX(), endLLP.getY(), endLLP.getX(), numPoints, true );
```

The arguments to `greatCircle()` should be in radians, but in this call the arguments are measured in degrees.

Geoconvertor is a Java API that converts latitude and longitude to points in the Universal Transverse Mercator coordinate system [12]. The faulty code detected is:

```
if (datum.name().equals("WGS84")) {
     e1=eccentricityOne(WGS84.MAJOR_AXIS, SAD69.MINOR_AXIS);
     e2=eccentricityTwo(WGS84.MAJOR_AXIS, SAD69.MINOR_AXIS)};}
```

The constructors `eccentricityOne()` and `eccentricityTwo()` each expect two arguments of the same Earth model, SAD69 for the first and WGS84 for the second. The code has the argument Earth models confused.

6 Related Work

Research efforts on requirements and specification have modeled the connections between the real world and the machine world [13, 14, 15, 16]. Parnas and Madey introduced the 4-variable model that describes the relationship between real-world entities and machine world primarily as mathematically relations [16]. Miller and Tribble extended the original 4-variable model that isolated the virtual versions of the monitored and controlled values in subsystems [15].

Zave and Jackson characterized phenomena of interest to the system and separate world phenomena from machine phenomena [14]. The reference model of Gunter et al. gives a detailed explanation of different classes of phenomena and relationship between environment and system [13].

Other research efforts extend type systems to support additional checking capabilities [17, 18, 19]. These frameworks refine the built-in type system so as to allow additional types to be defined. By contrast, real-world types urge programmers to think from the perspective of the real world.

Ait-Ameur et al. [1] propose bringing more system information into the development of software. They emphasize the separation of implicit and explicit semantics.

Dimensional analysis and units checking have been explored in many programming languages [21, 22]. Previous research focused on extending programming languages to allow checking dimensions of equations. Some researchers focused on validating unit correctness [3, 4, 5].

Researchers have attempted to improve program understanding by linking structured semantic information in real-world contexts to formalisms [23]. Ratiu et al. have developed techniques to improve the understanding of formalism elements by making explicit mappings between ontology classes and the program elements [23].

7 Conclusion

We have presented a general notion of type based upon the real-world properties of entities with which a computer system operates and the mapping from those entities onto their representations in the computer system. Real-world types allow the definition and enforcement of type rules derived from the real world, as well the machine world. This enables a new range of analyses including automatic detection of real-world type violations, and inspections targeted at error prone software practices.

Future work in this area includes development of libraries of real-world types, additional empirical studies, the mining of system documentation for real-world type information, and the further clarification, for software engineering applications, of the representational correspondences between symbolic references and their referents.

Acknowledgements. This work was supported in part by Dependable Computing LLC, in part by the National Science Foundation grant number 1400294, and in part by the U.S. Department of Defense under Contract H98230-08-D- 0171. Any opinions, findings and conclusions or recommendations are those of the authors and do not necessarily reflect the views of the United States Department of Defense.

References

1. Mars Climate Orbiter Mishap Investigation Board Phase I Report. National Aeronautics and Space Administration, Washington DC (1999)
2. Bergin, C. Harding, P.: Cygnus Delays ISS Berthing Following GPS Discrepancy, http://www.nasaspaceflight.com/2013/09/cygnus-cots-graduation-iss-berthing/
3. Antoniu, T., Steckler, P. A., Krishnamurthi, S., Neuwirth, E., Felleisen, M.: Validating the Unit Correctness of Spreadsheet Programs. In: 26th International Conference on Software Engineering, pp. 439-448. IEEE Press, New York (2004)

4. Grein, C., Kazakov, D. A., Wilson, F.: A Survey Of Physical Unit Handling Techniques In Ada. In: Rosen, J-P., Strohmeier, A. (Eds.) Ada-Europe '03, LNCS, vol. 2655, pp. 258-270. Springer, Heidelberg (2003)
5. Kennedy, A.: Dimension Types. In: 5th European Symposium on Programming, pp. 348-362. ACM Press, New York (1994)
6. International System of Units. National Inst. of Standards Technology, Washington, DC.
7. Kelpie Flight Planner for Flightgear. http://sourceforge.net/projects/fgflightplanner/
8. FlightGear.http://www.flightgear.org/
9. International Association of Geomagnetism and Aeronomy. International Geomagnetic Reference Field: The Eleventh Generation. Geophysical Journal International, 183, 3, 1216-1230 (2010)
10. World Magnetic Model. http://www.ngdc.noaa.gov/geomag/WMM/DoDWMM.shtml
11. OpenMap. https://code.google.com/p/openmap/
12. Geoconvertor. https://code.google.com/p/geoconvertor/
13. Gunter, C. A., Gunter, E. L., Jackson, M., Zave, P.: A Reference Model for Requirements and Specifications. IEEE Software. 17, 3, 37-43 (2000)
14. Jackson, M., Zave, P.: Deriving Specifications From Requirements: An Example. In: 17th International Conference On Software Engineering, pp. 15-24. ACM, New York (1995)
15. Miller, S.P., Tribble, A. C.: Extending The Four-Variable Model To Bridge The System-Software Gap. In: 20th Digital Avionics System Conference, pp. 1-5. IEEE Press, New York (2001)
16. Parnas, D. L., Madey, L.: Functional Documents For Computer Systems. Science of Computer. Programming. 25, 1, 41-61 (1995)
17. Papi, M., Ali, M., Correr JR., T. L., Perkins, J. H., Ernst, M. D.: Practical Pluggable Types For Java. In: SIGSOFT International Symposium on Software Testing and Analysis, pp. 201–212. ACM Press, New York (2008)
18. Markstrum, S., Marino, D., Esquivel, M., Millstein, T., Andreae, C., Noble, J.: JavaCOP: Declarative Pluggable Types For Java. ACM Transactions on Programming Languages and Systems. 32, 2, pp. 4:1-4:37 (2010)
19. Dietl, W., Dietzel, S., Ernst, M.D., Muşlu, K. and Schiller, T.W.: Building And Using Pluggable Type-Checkers. In: 33rd International Conference on Software Engineering, pp. 681-690. ACM, New York (2011)
20. Ait-Ameur, Y., Gibson, J.P., Méry, D.: On Implicit And Explicit Semantics: Integration Issues In Proof-Based Development Of Systems. In: Margaria, T., Steffen, B. (eds.) ISoLA, Part II. LNCS, vol. 8803, pp. 604–618. Springer, Heidelberg (2014)
21. Chen, F., Rosu, G., Venkatesan, R. P.: Rule-Based Analysis Of Dimensional Safety. In: Nieuwenhuis, R. (Ed.), RTA 2003. LNCS 2706, pp. 197-207. Springer, Heidelberg (2003)
22. Jiang, L., Su, Z. Osprey: A Practical Type System For Validating Dimensional Unit Correctness Of C Programs. In: 28th International Conference On Software Engineering, pp. 262-271. ACM, New York. (2006)
23. Ratiu, D., Deissenboeck, F.: From Reality to Programs and (Not Quite) Back Again. In: 15th IEEE International Conference on Program Comprehension, pp. 91-102. IEEE Press, New York (2007)