Design and Validation of a Tool for Checking Real-World Validty

Jian Xiang, John Knight, Kevin Sullivan

Department of Computer Science

University of Virginia

Charlottesville, VA USA

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

*Abstract*— Logic interpretations

Keywords— Prototype, logic interpretation, real-world types, software assurance

# Introduction

Software systems interact with real-world entities under the control of software logic to produce desired real-world behaviors. Software logic in such software systems should observe constraints in the real world, e.g. laws of physics. The failure of software to obey real-world constraints can lead to serious consequences, especially in safety-critical systems. In previous work [SAFECOMP], we introduced the concept of *real-world type* and *real-world type checking*, to systematically define and check real-world constraints. For a software system of interest, a *real-world type system* is developed and used for error detection. The preliminary results of applying the new technology are promising. Real-world type checking detected real errors in open-source projects which have not been previously reported.

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. This necessity demands a tool that supports the approach of real-world types. In this paper, we present a tool designed and developed to support real-world type systems for Java. The tool provides the support for (1) manipulating real-world type system, (2) conducting analysis techniques provided by real-world type systems, (3) facilitating development of real-world type systems.

The tool has been validated and tested in the case studies of real-world type systems. The results of the case studies showed that the tool (1) clearly supports user management of real-world type systems, (2) effectively synthesizes candidates of real-world type systems for faster development, and (3) successfully locates real errors that violate real-world constraints.

The remainder of this paper is organized as below: section II introduces objective and goals of the tool. Section III presents the design of the tool. Section IV describes the validation of the design. Section V and VI present the related work and conclusion.

# Objective

The main objective of the tool is to support easy and effective application of real-world type systems to different software systems. To accomplish this objective, the tool is designed with a list of goals:

* Goal #1: the tool should operate without requiring changes to the subject Java program.

Satisfying this goal would makes it easier for engineers to adopt to this technology. It provides three advantages: (1) the real-world type information would not obscure the basic structure of the program, (2) the real-world type system can be added to existing programs without modifying the original programs, and (3) real-world type systems can be added to programs asynchronously, thereby not impeding the development of the programs and permitting real-world types to be added to legacy software.

* Goal #2: the tool should allow incremental adoption when applied to a large software system.

It’s easier for the engineers to apply this technology if they can try the technology in an incremental manner. Engineers should be able to start by trying a few features with minimum effort. Then, they should be able to proceed with more effort and receive more benefits. In such a way, engineers can experiment with the technology to determine its efficacy for their software.

* Goal #3: the tool should facilitate engineers to develop real-world type systems.

The effort required from engineers to developing real-world type systems could be substantial for large software systems. The tool should be able to reduce such effort by a reasonable percentage. Also, the tool should be able to guide the engineers in developing real-world type systems.

* Goal #4: the tool should allow reusing real-world type systems.

Real-world types and type rules define the characteristics of real-world entities, and those characteristics are unlikely to change. Therefore, real-world types and type rules are ideal candidates for reuse. Reusing these real-world types and type rules can greatly reduce the effort required from engineers.

* Goal #5: the tool should support management of real-world type systems.

Essentially, real-world type systems are created by engineers. Therefore, one of the basic operations required for the tool is to manually creation of real-world type systems.

* Goal #6: the tool should implement analyzers to support analysis opportunities introduced by real-world type systems.

As introduced in our prior work, real-world type systems introduce analysis opportunities and the results of the analyses are very promising. In order to benefit from these analyses, the tool should implement functions to conduct all sorts of analyses.

# Design

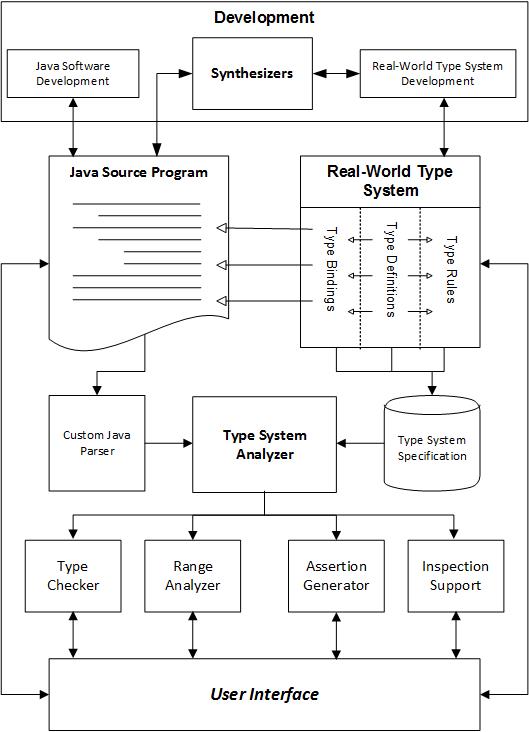
The design of the tool is motivated these goals stated above. The tool operates separately from the compiler via its own user interface.

## Design of the tool

The design of the tool is illustrated in Fig. 1. The subject Java source program is shown on the left toward the top of the figure, and the real-world type system is shown on the right. The development of the source program and the real-world type system, is shown at the top of the figure. The analyzers supported are shown at the bottom of the figure.

The tool supports:

* The definition of a set of real-world types for a Java program of interest.
* The definition of set real-world type rules by system experts based on real-world and application invariants.
* Creation of bindings between the real-world type definitions and entities in the Java source program.
* Static type checking of the Java program based on the set of real-world type rules.
* Synthesis of assertions as Java fragments that can be inserted into the subject program to implement runtime checking of type rules that cannot be checked statically.
* Synthesis of a checklist of locations in the subject program to which human inspection is required to check type rules that cannot be checked statically or dynamically.
* Synthesis of candidate real-world type systems from the Java source program. Synthesizers can produce candidate real-world types, real-world type rules, and real-world type bindings.



1. Design of the tool

## Use of the prototype

Real-world type systems are accessed via the user interface enabling:

* The establishment and display of bindings between items in the Java program and real-world type definitions in the interpretation.

Selecting an entity in the Java program that is to have a real-world type (clicking on the text) and selecting the particular real-world type to be used (clicking on the type name) establishes a binding. This binding corresponds to concept of real-world type binding introduced in section 4.3.1. In this prototype, the bindings are often referred to as annotations. These annotations can be displayed as comments in JavaDoc of Java programs.

* Reference to the details of the interpretation.

All definitional aspects of the real-world types and all bindings to Java entities can be displayed. The set of bindings can be displayed in various ways, e.g., all bindings, binding of a given Java entity, all Java entities bound to a particular real-world type, etc.

To support analysis of the system, a custom parser produces a representation of the subject Java program as an abstract syntax tree, and the implementation of the interpretation produces a database that documents all of the details of the interpretation. The abstract syntax tree and the details of the interpretation are processed by an analyzer shown in the center of the figure that supports four types of analysis:

* Real-world type checking.

A *type* *checker* was implemented for this analysis. It loads the real-world type rules, examines the parsed subject Java programs, and then statically checks for violations of real-world type rules. Diagnostics are displayed for user to confirm.

* Reasonable range analysis.

A *range* *analyzer* was implemented to conduct interval analysis on the source programs. Warning messages are issued when calculated intervals of program elements exceed their reasonable ranges.

* Assertion generation.

The *assertion* *generator* synthesizes assertions as Java fragments that can be inserted into the subject program to implement runtime checking of real-world invariants that cannot be checked statically.

* Targeted inspection.

The *inspection* *support* provides a display allowing all interpreted Java entities to be traced to their interpretations. Definitional aspects of the real-world specifications and all interpretations to Java entities can be displayed. It also synthesizes a checklist of locations in the subject program at which human inspection is required to check real-world constraints or invariants that cannot be checked statically or dynamically.

As indicated by the top part, Java programs are separately developed and parsed without being affected by the development of interpreted formalisms. In this way, the two artifacts can be developed in parallel without impeding each other. Engineers can manually create interpreted formalisms through user interface. In addition, the prototype implements two features that facilitate developing interpreted formalisms:

* Synthesis of interpreted formalisms.

The concept of synthesis has been introduced in Chapter 6. The prototype has mechanisms that implement the synthesis framework. Specifically, three mechanisms were developed: (1) synthesis of candidate real-world types, (2) synthesis of real-world type rules, and (3) synthesis of real-world type bindings.

* Reuse interpreted formalism.

Existing real-world type systems can be reused instantly for developing new real-world type systems. In this prototype, real-world types and type rules are stored as text files and can be readily reused in other real-world type systems.

## Typed Program Elements

Software entities that have real-world meanings should be interpreted with their real-world specification. In the context of a real-world type system, these software entities are bound with real-world types. The Java prototype covers most of these software entities. In the prototype, the Java entities being bound with real-world types are: (a) local variables, (b) fields in classes, (c) method parameters, (d) method return value, and (e) class instances. In order to make the development of the prototype tractable, the current version imposes some restrictions on the use of interpretations in Java, specifically:

* Fields. Fields in classes are assumed to be monomorphic, i.e., a field in a class is assumed to have the same corresponding real-world entity in all class instances. Fields are interpreted with real-world specifications inside the class declaration body.
* Class instances. Different instances of a class might have different real-world meanings and so the interpretation is of the instance, not the class. For example, suppose a class Point has three fields x, y, z. Further, suppose that pt1 and pt2 are both instances of Point but are from different coordinate systems. Writing a statement that involves both pt1.x and pt2.x such as pt1.x + pt2.x might be an error and so the two instances need to be distinguished.
* Method return value. Each function with a return value is interpreted with a real-world specification. If a particular method is not interpreted with a real-world specification, the analysis treats the method as polymorphic. For a polymorphic method, at each invocation site, all the expressions in the method declaration body are examined to determine the real-world type of the return statement. That ultimately will be the real-world type of the method invocation. If the method contains multiple return statements, the interpretation for the return value will be the one with no errors. Also, if interpretations for return statements are inconsistent, a warning message is issued.
* Arrays. Since individual array elements cannot be interpreted separately, all objects inside an array are treated as having the same interpretation.
* Constants. Variables are interpreted when declared, but constants are used as needed. Constants are dealt with simply by associating each one with a hidden variable and associating an interpretation with the variable.
* Compound objects. Class instances introduce the possibility of nesting of interpreted real-world entities because the class might have an interpretation and the fields within the class might have interpretations. In that case, the real-world specification of a qualified name is the union of the specifications of all the elements in the path to a specific item of interest in an expression. This same rule applies to method invocation where fields are retrieved such as cs2.get\_x();

# Validation

This section checks that if the tool satisfy the goals listed in section II. Every goal is reviewed and the tool is evaluated on if this goal is fulfilled.

* Goal #1: the tool should operate without requiring changes to the subject Java program.

Within this tool, real-world type systems are operated via a separate user interface. Java programs are not impacted by real-world type systems. Real-world types, real-world type rules, and real-world type bindings are saved as independent files in a separate folder. When analyses are triggered, the tool parses the Java programs and loads real-world type systems for subsequent processing.

* Goal #2: the tool should allow incremental adoption when applied to a large software system.

The tool permits incremental development of real-world type systems. Real-world types and type rules can be created one-by-one. Real-world type bindings can be created incrementally for each program element. Analysis techniques can be performed on the program statements where real-world type systems have been applied. The more effort engineer spend on developing real-world type system, the more potential benefits they shall receive.

* Goal #3: the tool should facilitate engineers to develop real-world type systems.

The tool implements three synthesizers to facilitate developing real-world type systems. A real-world type synthesizer extracts concepts from the subject Java program and generates a list of candidate real-world types. A real-world type binding synthesizer infers the type bindings for various program elements based on bindings created by engineers. A real-world type rules extracts candidate real-world type rules from program statements with type bindings. The synthesizers greatly reduce the effort required from engineers.

* Goal #4: the tool should allow reusing real-world type systems.

The real-world types and real-world type rules are implemented in an easily reusable formats. They are saved as independent files that can be reloaded into different real-world type systems.

* The tool should support management of real-world type systems.

This is the basic functionality provided by the tool. Engineers can import, create, modify, and delete real-world types, type rules, and type bindings. The contents in real-world types and type rules are displayed in the user interface. The real-world type bindings can be displayed as comments in Java source programs.

* The tool should implement analyzers to support analysis opportunities introduced by real-world type systems.

The tool implemented four analysis techniques. Real-world type checker and range analyzer found errors that violate real-world constraints in the programs statically. The errors found are displayed with diagnostics and traces in the user interface. The inspection support allows engineers to trace all real-world types. The assertion generator can produce assertions in the source programs.

# Experiences

We have used the tool to conduct case studies on two open-source geographic software: the Kelpie flight planner[] and the OpenMap[]. The detailed results can be found in [][]. This tool provides comprehensive support for developing real-world type systems and analyzing the subject software.

The tool found substantial amount of real errors in the two case studies. In the Kelpie flight planner case study, the tool located 7 errors by conducting real-world type checking and 12 errors by reasonable range analysis. In the OpenMap case study, the tool located 24 errors by conducting real-world type checking and 12 errors by reasonable range analysis.

The tool was very effective in facilitating engineers in developing real-world type systems. On average, it can (1) synthesize 60% candidate real-world types, (2) produce candidate real-world type rules for engineers to review, and (3) synthesize 50% of the real-world type bindings. Also, real-world types and type rules created in the Planner software were highly reused in the case study for OpenMap.

# Related Work

# Conclusion

##### References

1. L. Jiang, and Z. Su. “Osprey: a practical type system for validating dimensional unit correctness of C programs,” in Proceedings of the 28th international conference on Software engineering (ICSE '06). ACM, New York, NY, USA, 2006, pp. 262-271.