A Tool for Analyzing The Real-World Consistency of Software

Jian Xiang, John Knight, Kevin Sullivan

Department of Computer Science

University of Virginia

Charlottesville, VA USA

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

*Abstract*— Software systems that interact with the real world should observe constraints inherent in the real world. The concept of real-world type system has shown great potential in checking programs against real-world constraints. In order to support developing and using real-world type systems, we developed a tool for Java. The tool provides all the capabilities needed for applying a real-world type system. The tool was validated by applying to two open-source software in which it successfully detected a substantial amount of real faults and facilitated the users in the development of real-world type systems.

Keywords— Tool, real-world types, software assurance

# Introduction

Computer systems interact with real-world entities under the control of software to produce desired real-world behaviors. The logic in such systems should observe constraints from the real world, e.g. laws of physics. The failure of the software to obey real-world constraints can lead to serious consequences, especially in safety-critical systems. In previous work [10], we introduced the concept of *real-world type systems* and *real-world type checking*, to systematically define and check real-world constraints. In case studies of the application of real-world type checking, faults were detected that had 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 creation and application of real-world type systems. In this paper, we present details of *Osprey*, 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 the creation of real-world type systems.

The tool has been validated and tested in two case studies in which real-world type systems were developed for open-source software projects [CITE, CITE]. 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 faults 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 primary objective of the tool is to support efficient and effective application of real-world type systems to different software applications. To accomplish this objective, the tool was designed to address the following goals:

* **Analysis support**. The tool should support the analysis opportunities introduced by real-world type systems. These opportunities include real-world type checking, reasonable range analysis, targeted inspection, and executable assertions for runtime checking.
* **Immutable code**. The tool should operate without requiring any changes to the subject program. Satisfying this goal enables easier adoption of this technology. Specifically, meeting the goal would provide 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.
* **Ease of use**. The tool should facilitate engineers to develop real-world type systems. The effort required by engineers to develop real-world type systems could be substantial for large software systems. The tool should reduce such effort as much as possible. In addition, the tool should provide guidance to engineers when developing real-world type systems.
* **Incremental adoption**. The tool should allow incremental adoption when applied to a large software system. Adoption of the technology and successful results are more likely if the technology can be applied incrementally rather than requiring wholesale change with benefits increasing as more effort is expended.
* **Reuse**. The tool should allow the reuse of real-world type systems. Real-world types and type rules define the characteristics of real-world entities, and those characteristics are unlikely to change. Thus, real-world types and type rules are ideal candidates for the creation of libraries supporting reuse and the associated reduction in development effort.
* **Type system management**. The tool should support the 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 facilitate manual creation of real-world type systems.

# Design

The basic design of the Osprey user interface is shown in Fig. 1. The subject Java program is presented to the user in one window and the real-world type system in use is presented in a second window. Control of Osprey, including invocation of the various analyses and display of the results of analyses, are available through a set of graphic control panels.



1. The Osprey user interface.

Development of the Java software is entirely outside of Osprey.

To the user, the tool is organized as a group of GUI based operations. Users manipulate real-world type systems and trigger analyses through user interfaces. Upon requests from the users, the tool conducts these analyses and then sends the users feedbacks. The users’ view of the tool is shown in Fig. 1.

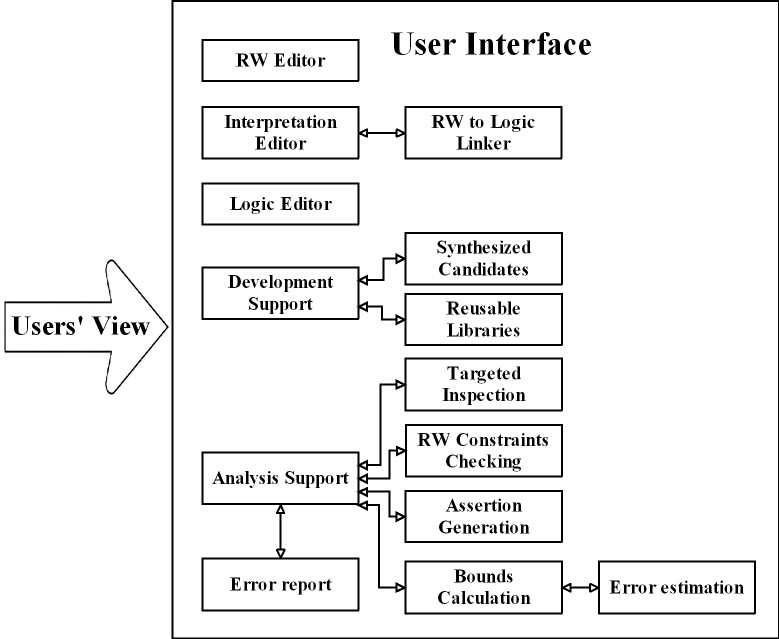
Real-world type systems are accessed via the user interface enabling:

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

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. These real-world type bindings can be displayed as JavaDoc comments inside the Java programs.

* Program analysis with real-world type checking and reasonable range analysis.

The two analysis techniques check violations of real-world constraints in the Java programs. The two techniques are triggered in the same way, by clicking in the Java program editor. After the analysis is finished, program statements with possible violations of real-world constraints are highlighted with color. They are considered as possible faults. The diagnoses of these possible faults are displayed in a table format view. Users can click on the diagnoses to trace the sources of the possible faults.



1. Users’ view of the tool

* Generations of assertions for runtime checking.

The assertion generation can be triggered by clicking on a specific real-world type (click the type name) or an element in the program (click on the text), and then produces an assertion around the program element or all elements corresponding to the real-world type.

* Targeted inspection of the programs.

Users can trigger an *inspection mode* with a clicking on the Java problem. The inspection mode provides a display that allows all Java entities to be traced to their real-world types. Users can inspect each program element carefully. The inspection mode also synthesizes 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.

* Synthesis of real-world types and real-world type rules.

Synthesis of candidate real-world types and type rules are triggered by clicks on the Java programs. The synthesis provides standalone text files for candidate real-world types and real-world type rules. The text files are then reviewed by the users to select the candidates that apply and then construct complete real-world types and type rules.

* Synthesis of real-world type bindings.

Synthesis of real-world type bindings is triggered by clicks on the Java programs which already have some bindings seeded inside. The synthesis then propagates these existing bindings to other program statements where the inference rules apply. The binding synthesis can be triggered on a collection of source files with one click.

* Reuse interpreted formalism.

Existing real-world types and type rules can be easily reused in different real-world type systems. Contents in a real-world type system are organized as files in a folder. Files for real-world types are type rules can be copied from one real-world type system to other real-world type systems.

* Reference to the details of the real-world type system.

All definitional aspects of the real-world types, real-world type rules, 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.

## Architecture of the tool

To accomplish the goals stated in section II and provide all the support to the users, we implement the tool in an architecture shown in Fig. 2. The tool is implemented as an Eclipse Rich Client Platform [5]. The subject Java source program is shown on the left toward the top of the figure, and the interpretation is shown on the right. The development of the Java program and the real-world type system, is shown at the top of the figure.

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 real-world types and type rules produces a database that specifies all of the details of the types and type rules. The abstract syntax tree and the specification for the real-world type system are processed by an analyzer shown in the center of the figure that supports four types of analysis:

* Real-world type checking.

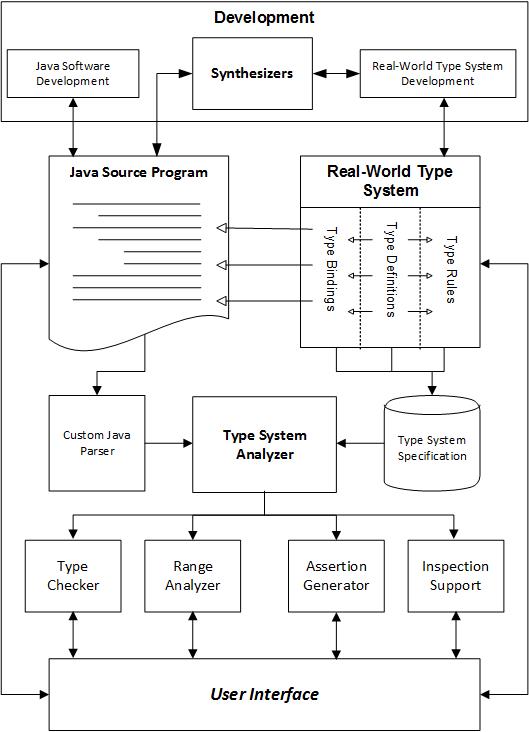
A *type* *checker* was implemented for this analysis. It loads the real-world types and type rules, examines each node, especially infix expressions, in the abstract syntax tree, and then checks for violations of real-world type rules. Diagnostics are displayed for users to review.

* Reasonable range analysis.

A *range* *analyzer* was implemented to conduct reasonable range analysis. It reads the reasonable range values specified in real-world types and then conducts interval analysis on the Java program. Warning messages are issued when calculated intervals of program elements exceed their reasonable ranges.

* Assertion generation.

An *assertion* *generator* was implemented to synthesize assertions as Java fragments that can be inserted into the subject program. These assertions can be used to implement runtime checking of real-world invariants that cannot be checked statically.



1. Architecture of the tool

* Targeted inspection.

The *inspection* *support* was implemented to assist human inspections. It has a display called *inspection mode* that reads and displays real-world types for every program element selected by users. It also synthesizes a checklist of locations in the subject program which inconsistent use of real-world types is referred to.

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

* Synthesis of real-world type systems.

Three synthesizers were implemented for producing candidate real-world types, real-world type rules, and real-world type bindings. The details about the synthesizers were introduced in our prior work [11]. In summary, the synthesizer for real-world types leverages natural language processing techniques to process the identifiers in the program to produce a list of candidate real-world types. The synthesizer for type rules extracts operations that bound with real-world types to produce candidate real-world type rules. The synthesizer for type binding produces bindings according to several inference rules.

* Reuse interpreted formalism.

The real-world types and type rules are saved as files following a predefined syntax. They can be directly reloaded into different real-world type systems. The rules for units checking are set as default rules for all 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 fault 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 faults. 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

With this tool support, the goal of *analysis support*, *immutable code*, *ease of use*, *incremental adoption*, *reuse*, and *type system management* stated in section II have all be fulfilled.

In order to investigate the performance of the tool, we have used it to conduct case studies on two open-source geographic software: (1) a moderate-sized software, the Kelpie flight planner [8], with 13,884 lines of code, and (2) a large-sized software, the OpenMap[9], with 157,858 lines of code. The detailed results of the first case study can be found in[]. The details of the second case study can be found in another paper submitted to [].

This tool provides comprehensive support for developing real-world type systems and analyzing the subject software. In summary, the tool found a substantial amount of real faults in the two case studies. In the Kelpie flight planner case study, the tool located 7 faults by conducting real-world type checking and 12 faults by reasonable range analysis. In the OpenMap case study, the tool located 24 faults by conducting real-world type checking and 12 faults by reasonable range analysis.

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

# Related Work

The tool introduced in this paper implements the idea of real-world type systems. Real-world types are real-world analog of types used in programming languages. Other researchers have made effort on extending the basic types to support additional checking capabilities.

Pluggable type systems [4] enhance the built-in type systems in applicable formal languages and provide support for additional fault checking capabilities. The Checker framework [2] is a toolset that implements the idea of pluggable type system for Java. Dependent type systems, such as Coq [3] and Agda [1] provide formal languages to write mathematical definitions, executable algorithms, and theorems, and then support development of proofs of these theorems.

Some research tools support units checking capabilities[][]. They permit adding annotations or type qualifiers to the source programs to denote units and enforce units consistency.

# Conclusion

In this paper, we introduce the tool designed and implemented to support real-world type systems, thereby conducting fault checking against real-world constraints. The tool provides all the necessary functions to apply the real-world type system. It has been used on modern software system for fault detections. In the case studies, the tool has found a substantial amount of faults. It can be practically used in different open-source software projects.

##### References

1. Agda. http://wiki.portal.chalmers.se/agda/pmwiki.php.
2. Checker framework. http://types.cs.washington.edu/checker-framework/
3. Coq. <https://coq.inria.fr/>
4. 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.
5. Eclipse Plug-in Development.

<http://www.vogella.com/tutorials/EclipsePlugin/article.html>

1. Hills, M, F. Chen, and G. RoşU. 2012. A Rewriting Logic Approach to Static Checking of Units of Measurement in C. Electron. Notes Theor. Comput. Sci. 290 (December 2012), 51-67.
2. 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.
3. Kelpie flight planner for FlightGear.

http://sourceforge.net/projects/fgflightplanner/

1. OpenMap. http://openmap-java.org/
2. 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.
3. 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.