Falcon: A Toolset 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 be consistent with constraints inherent in the real world. In previous work, we introduced the concept of real-world type systems and demonstrated its potential in checking the consistency of programs with the real-world. In order to support the use of real-world type systems, we have developed a support toolset for Java. The toolset, called Falcon, provides all the capabilities needed for developing and applying real-world type systems. The toolset has been used in two case studies of open-source software in which it supported: (a) the creation of the necessary real-world type systems, and (b) the analyses of the software. Using the toolset required reasonable effort and a substantial number of real software faults were detected.**

Keywords— Toolset, real-world types, software assurance

I. 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 [10].

In previous work [12], we introduced the concept of the *interpreted formalism*, a software version of interpreted logic. We also introduced a practical instantiation of interpreted formalism, *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, software faults were detected in the subject applications 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 *Falcon*, a toolset designed and developed to support real-world type systems for Java. The toolset provides support for (1) manipulating real-world type system, (2) invoking analysis techniques provided by real-world type systems, (3) facilitating the creation of real-world type systems.

The toolset has been validated and tested in two case studies in which real-world type systems were developed for open-source software projects [12], [14]. The results of the case studies showed that the toolset: (a) clearly supports user management of real-world type systems, (b) effectively synthesizes candidates of real-world type systems for faster development, and (c) successfully locates faults that violate real-world constraints.

The remainder of this paper is organized as follows. We present the objective and goals for Falcon in Section II. In Section III we describe the users' view of the toolset, and in Section IV we discuss the architecture of Falcon. In Section V we summarize the results obtained using the toolset in two case studies. In Section VI we review the related literature, and we present our conclusions in Section VII.

II. Objective

The objective of the toolset is to support efficient and effective application of real-world type systems to different software applications. To accomplish this objective, the toolset was designed to address the following goals:

- **Analysis support**. The toolset should support the analysis opportunities introduced by real-world type systems. These opportunities include real-world type checking, reasonable range analysis, targeted inspection, and synthesis of executable assertions for runtime checking.
- **Immutable code**. The toolset 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 toolset should support engineers in the development of real-world type systems. The effort required by engineers to develop real-world type systems

can be substantial for large software systems. The toolset should reduce such effort as much as possible. In addition, the toolset should provide guidance to engineers when developing real-world type systems.

- **Incremental adoption**. The toolset 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 with benefits increasing as more effort is expended rather than requiring wholesale change.
- **Reuse**. The toolset 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 toolset 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 toolset is to facilitate the manual creation of real-world type systems.

III. The User's View

In related work on units checking [6][7], toolset support has focused on: (a) source code annotations in the form of comments, (b) and command line invocation of analysis. Meeting the objectives listed above for real-world type systems requires a sophisticated interface that provides a number of facilities to the user.

A high-level view of the Falcon 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. The Java program display is purely for reference; development of the Java software is entirely outside of Falcon. Control of Falcon, including invocation of the various analyses and display of the results of analyses, are available through a set of graphic control panels.

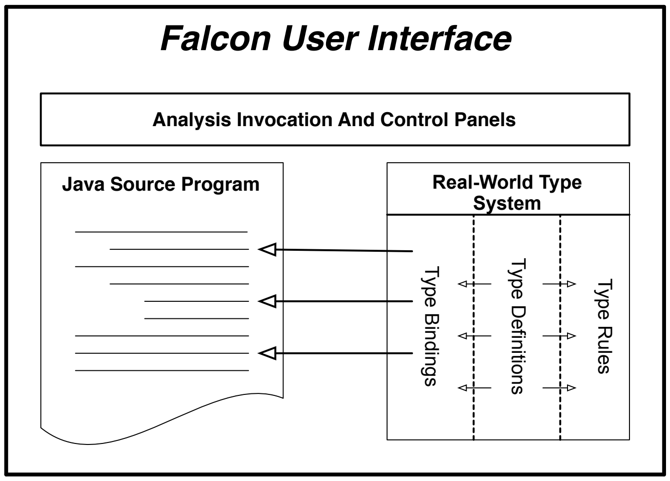

Fig. 1. The Falcon user interface.

The major features of the interface are as follows:

- **Java highlighting**. To assist the user in navigating the source code, the Java display highlights the source code with coloring in a typical fashion.
- **Type bindings**. Bindings of real-world types to program elements are established by highlighting the program element in the source code and then selecting the desired real-world type from a list. By default, bindings are not displayed in the source code although they can be easily displayed via a tooltip (see below). If desired Falcon will inject Java comments into the source code to indicate bindings as a convenience to the user.
- **Statistics**. Statistics about the real-world type system are displayed including the number of types, the number of type rules, and the number of bindings of real-world types to program elements.
- **Real-world types**. The list of type names is presented, and, for each, all of the attributes of the types, measurement units for example, are listed.
- **Program element types**. In the display of the Java source code, by hovering the mouse over a program element, the real-world type associated with the element is displayed as a tooltip. This mechanism allows the user to review and check the real-world typing for a complete expression by moving the mouse over the expression in an appropriate way.
- **Elements of a type**. All of the program elements of a particular type in the Java source code can be displayed.
- **Libraries**. A real-world type system can be developed and subsequently stored as a library, or an existing library can be read and made available for analysis of a given source program.
- **Type synthesis**. Partial support for synthesis of the real-world type system for a given source program is provided. The Falcon toolset includes a processor that processes the Java source code and parses all of the identifiers. The results are supplied to the user to consider as elements of real-world type definitions. With types defined, the toolset facilitates the establishment of type bindings by providing type propagation of initial bindings seeded by the user [13].
- **Analysis**. All forms of analysis are invoked and all resulting analyses displayed from the user interface. For example, real-world type checking is invoked from a menu item, and any real-world type errors (violation of a real-world type rule) that are detected are displayed in a separate window.

IV. The Toolset Architecture

Falcon is implemented as an Eclipse Rich Client Platform [5]. The overall architecture of the toolset is shown in Fig. 2. For completeness, the dashed rectangle at the top left of the figure shows the development of the Java subject system although this is not part of Falcon.

The major elements of the architecture are:

- **Custom Java parser**. Entirely separate from the compiler used to build the subject system, Falcon includes a Java parser that produces a symbol table and an abstract syntax tree for the subject software. This parser ensures that Falcon does not depend upon any of the toolsets being used in system development.

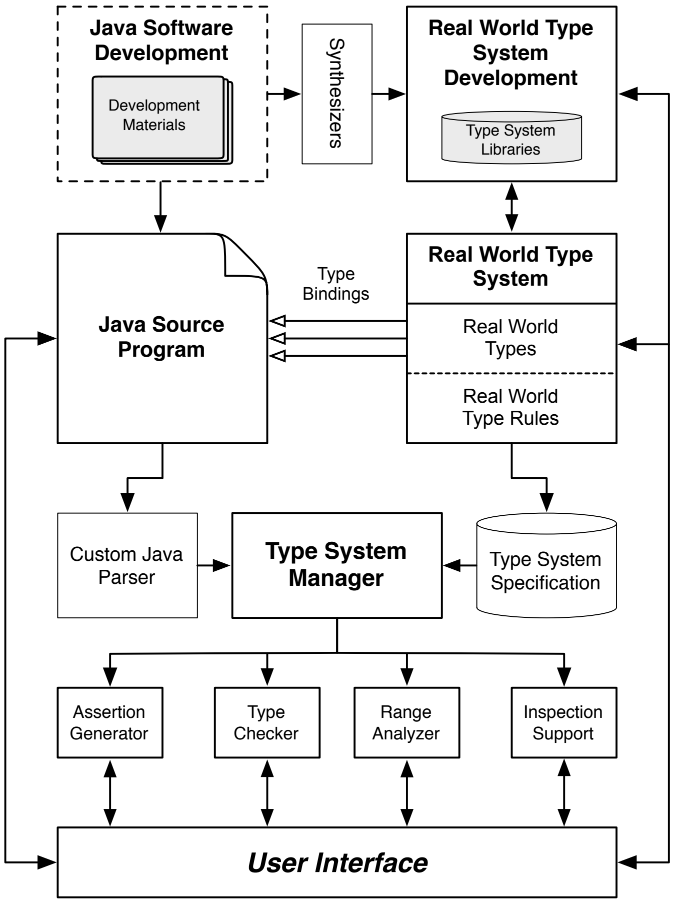

Fig. 2. Architecture of the Falcon toolset

- **Synthesizers**. The synthesizers are the support system provided to help the user develop the real-world type system for the subject software. The synthesizers process the development materials in various ways including: (a) parsing all of the identifiers in the subject software using grammars derived from typical naming conventions, (b) application of elementary natural language processing, (c) consultation of lexical/ontological databases, (d) real-world type inference, and (e) user review of synthesized type materials.

- **Type system manager**. The type system manager is a control mechanism that determines the materials needed for the various analysis mechanisms and supplies it as needed to the analysis components of the toolset.

- **Analysis engines**. Presently, Falcon provides four different forms of analysis. Each of the analyses is implemented by a different engine. The architecture is designed to support additional analyses as they become available.

- **Type system libraries**. Libraries of real-world types enable reuse across applications of type definitions. Some type definitions derive from real-world constraints and rarely change. Others derive from application-specific concepts and constraints (units and frames of reference for example), and are more likely to be suitable for reuse on an individual type basis. Support for reuse of individual types and type collections is implemented by a simple catalog and search mechanism.

All of the mechanisms provided by Falcon are implemented through a Java class hierarchy and a simple set of file types that ensure flexibility and support for future growth.

The current implementation of Falcon supports the use of real-world types for most Java program elements. In particular, the Java entities that can be bound to real-world types are: (a) local variables, (b) fields in classes, (c) method parameters, (d) method return values, and (e) class instances. In order to make the development of the prototype toolset tractable, the current version imposes some restrictions on the use of real-world types 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 associated 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 real-world type 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 a fault and so the two instances need to be distinguished.

- **Method return value**. Each function with a return value is associated with a real-world type. If a particular method does not have a real-world type, 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 real-world type for the return value will be the one with no faults. Also, if real-world types for return statements are inconsistent, a warning message is issued.

- **Arrays**. All elements of an array are assumed to have the same real-world type.

- **Constants**. Variables are associated with real-world types at the point of declaration, but constants are used as needed. Constants are dealt with simply by associating each one with a hidden variable and binding a real-world type to that variable.

- **Compound objects**. Class instances introduce the possibility of nesting of real-world typed entities because the class might have a real-world type and the fields within the class might have real-world types. 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();`

V. Validation

The performance of the Falcon toolset was assessed as part of two case studies on open-source software systems that implement various mapping/geographic services. The systems used in the case studies were: (1) the Kelpie Flight Planner (13,884 lines of code) [8], and (2) OpenMap (157,858 lines of code) [11].

In the Kelpie flight planner case study, seven faults were located by real-world type checking and 12 faults by reasonable range analysis [12]. In the OpenMap case study, 24 faults were located by real-world type checking and 12 faults by reasonable range analysis [14]. Our informal assessment of the toolset is that it provided comprehensive support for accessing and manipulating the necessary real-world type systems and for analyzing the subject software.

The toolset was also effective in assisting in the development of real-world type systems by synthesis. On average, the toolset: (1) synthesized candidate real-world types that were used to produce 60% of the real-world types needed, (2) produced candidate real-world type rules that covered all real-world type rules needed, and (3) synthesized 50% of the real-world type bindings required. In addition, the real-world types and type rules created for the Kelpie Flight Planner software were fully reused in the case study of the OpenMap software.

Again informally, from our experience with the toolset and with the underlying technology of real-world type systems, our initial assessment is that Falcon meets the six goals outlined in Section II, namely: *analysis support*, *immutable code*, *ease of use*, *incremental adoption*, *reuse*, and *type system management*.

VI. Related Work

The toolset introduced in this paper implements the idea of real-world type systems. Other researchers have developed a wide variety of enhancements to the basic types in programming languages in order to support additional checking capabilities.

Pluggable type systems [4] enhance 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 and dimensional analysis [6], [7]. They permit adding annotations or type qualifiers to the source programs to denote units and enforce unit consistency. These tools focus on units and dimensional analyses, both of which are special cases of the real-world types and the associated analyses.

The learning curves for using these tools are arguably high. Compared with these tools, Falcon emphasizes the separation of the development of source code from the development of real-world type systems. In addition, Falcon focuses on the ease of use by providing comprehensive user interfaces and synthesis mechanisms thereby targeting pragmatic applications for engineering purposes.

VII. Conclusion

In this paper, we have introduced the Falcon toolset; a toolset designed and implemented to support real-world type systems. The toolset enables analysis using constraints derived from properties of the real world. The toolset provides all the necessary capabilities to develop and apply real-world type systems to Java software and to analyze the result.

The toolset has been applied to modern software system as part of two case studies. In the case studies, the analyses made possible by the toolset detected a significant number of faults.

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

[6] 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.

[7] 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.

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

[9] Knight, J, J. Xiang, and K. Sullivan. "A rigorous definition of Cyber-physical systems". *Trustworthy Cyber-Physical Systems Engineering*. London: Chapman & Hall, 2016. 47-73.

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

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

[12] 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.

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

[14] Xiang, J., J. Knight and K. Sullivan, 2016. "Is My Software Consistent With the Real World?," Submitted to the 18th International IEEE Symposium on High Assurance Systems Engineering (HASE), Singapore, 2017.