A Practical Approach for Units Checking on Large Systems

Jian Xiang, John Knight, Kevin Sullivan

Department of Computer Science

University of Virginia

Charlottesville, VA USA

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

*Abstract*— Units checking

Keywords— Units checking, logic interpretation, real-world types, software assurance

# Introduction

Dimensional and unit analysis have been extensively explored by researchers in many programming languages [15, 30, 82]. Most of previous research focused on two kinds of approaches: (1) creating extensions for programming languages to support dimensional and units analysis and (2) adding annotations into source programs to denote units and associate units with program variables. Both approaches reply on developers’ manually creation of associations between program elements and their corresponding units. The amount of effort required for developers highly depends on the nature of the program of interest. The effort required from users could be substantial.

The effort required frequently defers developers from using units checking techniques in software systems, especially large systems. Previous research work on units checking often lack evidence and results on large experimental subjects. Most experiments were conducted on programs with about or less than 5,000 lines of code [spreadsheet, CHEN]. The largest subject programs are around 10,000 lines of code [SU, STANFORD]. Assessments of runtime performance have been done on larger programs which are only synthesized programs but not realistic ones.

In this paper, we introduce a practical approach for units checking on large software systems. The approach combines static units checking with inference mechanisms. Units annotations which are created manually by developers are inferred to other program elements. The approach was applied to two open-source geographic software systems, one with 13,000 lines of code and the other with 157,000 lines of code. Our approach found several units errors in both case studies. The effort required from the developers for annotating units were reduced by around 50%.

The remainder of this paper is organized as below: section II introduces some background about this research. Section III presents the details of our approach. Section IV describes the results of the case studies conducted and compare them with other research effort. Section V and VI present the related work and conclusion.

# Background

## Units Checking Fundamental

## Our Prior Work

### Real-world type checking

### Synthesis framework

# A Practical Approach

## Interpretation + Synthesis

Some researcher work attempted to alleviate the burden required from developers by introducing inference mechanisms.

## Application Procedure

In this case study, the process of adding type bindings was carefully organized. It was composed of a sequence of two binding operations: *binding* *seeding* and *binding* *propagation*.

Binding seeding was done by the user. Users read source files and seeded type bindings to program elements manually. At the beginning of this case study, bindings were seeded to a few JavaBean files that directly interact with real-world entities. For example, LatLonPoint.java was the first source file into which type bindings were seeded. The file contains global variables representing latitude, longitude, and altitude. The file also has a collection of utility functions, e.g., a function that calculates distance between two coordinates; a function that computes the heading between two coordinates. During later parts of the case studies, bindings were seeded to different source files that access real-world entities.

Binding propagation was done by the type binding synthesizer. After type bindings were seeded to source files, the binding synthesizer propagated these bindings to other source files in OpenMap. From user’s perspective, binding propagation can be done on one or several files in one operation. For example, binding propagation was frequently applied to a package of source files. Type bindings in all files inside the package were propagated sequentially.

# Case Studies

In order to get insights about the performance of our approach. We conducted two case studies by applying the approach to two open-source geographic software systems. This section first presents the results we collect and then compare with some of previous research work on units checking.

## Case Study Subjects

TABLE I. shows the basic information of the two software systems:

1. Basic information of two subject software

|  |  |  |  |
| --- | --- | --- | --- |
| Software | # of packages | # of source files | Lines of code |
| Kelpie flight planner | 10 | 126 | 13,884 |
| OpenMap | 92 | 1193 | 157,858 |

## Error Found

Our approach found real unit errors in both case studies, TABLE II. summarizes the errors:

1. Errors found in two software

|  |  |  |
| --- | --- | --- |
| Software | # of errors | # of errors reported |
| Kelpie flight planner | 4 | 10 |
| OpenMap | 11 | 48 |

Units checking found 4 unit errors in the Kelpie flight planner. Two errors are caused by incorrect conversion between units of hour and units of minute. Another error is caused by misuse of variables in units of feet per minute and variables in units of nautical mile per hour. The fourth error is caused by misuse of the units radians and degree. Here is a sample error found in Kelpie flight planner.

The erroneous statement is:

double roughLonSep = range /

( 60 \* Math.cos(point.getLatitude()));

The function point.getLatitude() returns a latitude value with unit of degree when unit of radians is needed.

Units checking found 11 units errors in the OpenMap. Most of these errors are caused by misuse of the units degree and radians. As error samples, four units errors have been reported in source file Route.java. Two of them exist in the statement below:

float timeLimitBase =

GreatCircle.sphericalDistance(

toLat, toLon, fromLat, fromLon)

/ worstConvoySpeed;

The function GreatCircle.sphericalDistance() expects parameters in units of radians, while the arguments toLat, toLon, fromLat, and fromLon all are in units of degree. Another error in this statement is caused by the division. This function sphericalDistance() returns a value in units of radians, and worseConvoySpeed is in units of kilometer per hour. Such division is an inconsistent use of units. Two other units related errors were found in this file.

All errors found in OpenMap have been reported to the authors(s) of OpenMap through their GitHub site and the author confirmed them.

## False Warnings

As indicated by the table, the units checking has a number of false warnings, especially in the OpenMap case study. Most false warnings come from statements that are similar to the statements below:

lat = Math.toRadians(lat);

lon = Math.toRadians(lon);

Variables lat and lon on the left side represent values of latitude and longitude values in units of radians, but the two variables represent values in units of degree on the right side. The variables take different units in the same statements. As stated above, we choose to mark such statements as warnings to request developers’ attentions for units conversion.

## Effort

In both case studies, real-world type bindings are both created manually by developers and automatically by synthesizers. The synthesis framework has effectively reduced the amount of bindings required from developers. XXX summarizes the results pertinent to effort:

1. Effort in two case studies

|  |  |  |  |
| --- | --- | --- | --- |
| Software | Type bindings in total | Type bindings seeded manually | Type bindings synthesized |
| Kelpie flight planner | 255 | 122 | 133 |
| OpenMap | 1932 | 1129 | 803 |

1. Comparison with selected research work on units checking

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Research Work | Methodology | | | Experimental Subjects | | Errors Detection | | Human Effort | |
| Programming Language | Need Annotation | Inference Support | Size of Real Applications | Size of Artificial Programs | # of Real Errors Found | Most Errors Found in Single File | # of Annotation in Total | # of Annotation by human |
| Our Approach | Java | Yes | Yes | 171,000 LOC | 0 | 15 | 4 | 2187 | 1251 |
| Hills | C | Yes | No | 400 LOC | 111,000 LOC | 8 | 3 |  |  |
| Antoniu | Spreadsheet | Yes | No | 13,000 Cells | 0 | 3 | 1 |  |  |
| Hangal | Java | No | Yes | 19,000 LOC | 0 | 3 | 1 | NA | NA |
| Jiang | C | Yes | Yes | 9,400 LOC | 745,000 LOC | 5 | 2 | 795 | 795 |

The Kelpie flight planner case required 255 bindings in total, and 133 of these were generated automatically, i.e., 52%. The OpenMap case required 1932 bindings in total, and 803 of these were generated, i.e., 41.5%. The fractions suggests that the combination of some human effort and inference can yield reasonable performance in binding units with program elements.

An interesting fact is that the binding synthesizer demonstrated better efficacy at early stages. For the first 507 type bindings, 199 bindings were seeded and 298 bindings were synthesized; for the first 1024 type bindings, 488 type bindings were seeded and 536 bindings were synthesized.

## Comparsion with other approaches

With the results presented, we now can compare them with some prior research work. In this comparison, we select 4 relative new publications on units checking which have explicit experimental results. The 4 research work come from Jiang[], Hills[],Antoniu[], and Hangal []. The comparison is presented in XXX.

The

# 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.