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 real world applications with about or less than 5,000 lines of code [spreadsheet, CHEN]. The largest subject programs are around 15,000 lines of code [SU, STANFORD]. Assessments of runtime performance have been done on larger programs which are only synthesized programs but not real applications.

In this paper, we introduce a practical approach for units checking on large software systems. The approach combines type based units checking with inference mechanisms. Units annotations 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

## Our Prior Work

### Real-world type checking

### Synthesis framework

## Units Checking Fundamental

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 []. 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 consists of the seven basic dimensions of physics (mass, length, time, electric current, temperature, luminosity, and amount of substance) [36]. The existence of this semantic 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)

# 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 which we don’t have association with.

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 |

The Kelpie flight planner[] is an open-source Java project based on FlightGear []. The Planner project uses the airport and navaid databases of FlightGear to determine routes between airports based on user inputs. Results are presented using a sophisticated graphical interface.

OpenMap[] is a JavaBean-based toolkit for building applications and applets needing geographic information. Using OpenMap components, users can access data from legacy applications. The core components of OpenMap are a set of Swing components that understand geographic coordinates. These components allow users to show map data and manipulate that data.

Both the Kelpie flight planner and the OpenMap make calculations involving distances, velocities, speeds, latitude, longitude, time and so on, and it does so using a variety of units. Clearly, the two software are of the type for which dimensional and units analysis has the potential to discover errors.

## 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 |

Four unit errors were found 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 degrees. 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.

Below shows another sample error. One units error was found in the file QuadTreeNode.java. The error is located in the statements:

double distanceSqr = dx \* dx + dy \* dy;

if (distanceSqr < bestDistance.value) {

bestDistance.value = distanceSqr;

closest = qtl.object;

}

Variable distanceSqr and variable bestDistance.value are different in units, one in units of square of degrees and the other in units of degrees. The two variables are not commensurable.

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. Comparison with selected research work on units checking

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Research Work | Methodology | | | Experimental Subjects  (in total) | | User Effort | | | Errors Detection |
| Language | Need annotation | Units inference | Size of real applications | Size of artificial programs | # of annotation in Total | # of annotation by user | Ratio:  user annotation /  program size | # of Real Errors Found |
| Our Approach | Java | Yes | Yes | 171,000 LOC | 0 | 2187 | 1251 | 0.7% | 15 |
| Hills | C | Yes | No | 400 LOC | 111,000 LOC | 65 | 65 | 16% | 8 |
| Antoniu | Excel | Yes | No | 13,000 Cells | 0 | 13,000 | 13,000 | 100% | 3 |
| Hangal | Java | No | Yes | 19,000 LOC | 0 | NA | NA | 0 | 3 |
| Jiang | C | Yes | Yes | 9,400 LOC | 745,000 LOC | 795 | 795 | 8.5% | 5 |

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 |

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 Hills[],Antoniu[], Hangal[], and Jiang[]. We give a brief introduction to each research work here.

* The work from Hills et.al [] uses annotations in comments to denote units for program variables, then it statically analyzes the subject programs based on an abstract rewriting logic semantics of C programs.
* The work from Antoniu et.al [] associates units specified in Excel spreadsheet with cells in the spreadsheet, then it checks units consistency by solving the units related constraints generated from the spreadsheet.
* The work from Hangal et.al [] is an automatic units checking approach which relies on software update. It infers and traces units relationships between program elements in every release of the software, and detects inconsistency between the releases or code updates.
* The work from Jiang et.al [] adds units annotations to C programs, and then generates units related constraints. Errors are reported when the constraints cannot be solved.

The comparison is presented in TABLE III. For each approach, we show basic information, subjects for experiments, error detected, and effort made by the users. The details of the comparison are list below:

* Experimental subjects.

Experiments in this research involve a much larger software systems with 170,000 lines of code. Prior research work conducted experiments on programs with less than 20,000 lines of code. Some of them conducted experiments on large artificial programs to assess the scalability, rather than the units checking capability. In these large artificial programs, only a few units were annotated to the subject programs.

* User effort.

In this paper, the two case study subjects require a total of 2187 annotations (i.e. real-world type bindings). The number of annotations made by users is 1251. Since the total size of the two subjects is 171,000 LOC, the annotation ratio is about 1.3% and the ratio for user annotation is 0.7% (1251/171,000). In the other research work which require user annotations, the number of annotations involved is less than the number required in our case studies. Their ratios of user annotation to the size of programs are 8.5%, 16%, and 100%. The work from Hangal doesn’t require annotations but its applicability is very limited since it relies on software updates.

* Error detection.

Our approach detected 15 errors in the two case studies. All other approaches have detected a reasonable amount of errors.

The comparison presented in this section indicates our case studies involved more user effort and larger software systems. The ratio of user effort to the size of software is lower. The performance of units error detection is arguably the same.

# Related Work

Dimensional analysis and unit checking have been explored in many programming languages [15, 30, 82]. Previous research focused on extending programming languages to allow checking these constraints on dimensions of equations are not broken. Extensions to support dimensional and unit analysis have been developed for several programming languages. For the most part, previous research focused on checking dimensions of equations and validating unit correctness [5, 19, 30, 37, 44, 68]. Nevertheless, these efforts are limited to basic rules derived from dimensions or combinations of entities with different units.

# Conclusion

##### References

1. Antoniu, T., P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. 2004. “Validating the Unit Correctness of Spreadsheet Programs”. In Proceedings of the 26th International Conference on Software Engineering (ICSE), Edinburgh, Scotland, 2004, 439-448. IEEE Computer Society, 2004.
2. 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.
3. Hangal, S., and M. S. Lam. 2009. “Automatic dimension inference and checking for object-oriented programs.” In Proceedings of the 31st International Conference on Software Engineering (ICSE). 155-165. IEEE Computer Society, 2009.
4. 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.
5. Kelpie flight planner for FlightGear.

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

1. OpenMap. http://openmap-java.org/
2. Petty, G. 2001. “Automated computation and consistency checking of physical dimensions and units in scientific programs.” In Softw. Pract. Exper. 1067-1076. New York: John Wiley & Sons, Inc., 2001.
3. 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.
4. 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.
5. Kennedy, A. 1999. “Dimension types.” In Proceedings of the 5th European Symposium on Programming (ESOP): Lecture Notes in Computer Science volume 788. Springer, 1999.