Electronic Communications of the EASST
Volume XXX (2014)

Proceedings of the 14th International Workshop on Automated Verification of Critical Systems (AVoCS 2014)

No-Test Classes in C through Restricted Types

Dave Donaghy and Tom Crick

2 pages

Guest Editors: Marieke Huisman, Jaco van de Pol
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

No-Test Classes in C through Restricted Types

Dave Donaghy[1] **and Tom Crick**[2]

[1] dave.donaghy@hp.com
HP Bristol, UK

[2] tcrick@cardiffmet.ac.uk
Department of Computing
Cardiff Metropolitan University, UK

Abstract: Object-oriented programming (OOP) languages allow for the creation of rich new types through, for example, the `class` mechanism found in C++ and Python (among others).

These techniques, while certainly rich in the functionality they provide, additionally require users to develop and test new types; while resulting software can be elegant and easy to understand (and indeed these were some of the aspirations behind the OOP paradigm), there is a cost associated to the addition of the new code required to implement such new types. Such a cost will typically be at least linear in the number of new types introduced.

One potential alternative to the creation of new types through *extension* is the creation of new types through *restriction*; in appropriate circumstances, such types can provide the same elegance and ease of understanding, but without a corresponding linear development and maintenance cost.

Keywords: Compilers, Plug-ins, Verification, Restricted Types

1 Introduction

Object-oriented programming (OOP) languages allow for the creation of rich new types through, for example, the `class` mechanism found in C++ and Python. However, it might be possible to obtain some of the gains of such techniques without the associated overheads in cost.

2 Development Cost of New Types

In an object-oriented development environment, it can reasonably be said that *all* software is encapsulated as methods on various types; indeed, Java, for example, requires that all executable code be written as type methods, allowing for the notion that `static` methods are still a kind of type method.

At the very least, then, the development of new types has *some* cost (and in particular, some financial or resource cost) associated to it. While we do not intend to directly measure this cost, a fair starting assumption might be that is linear in the number of new types introduced.

3 Restricted Types

One potential alternative to the creation of new types through *extension* is the creation of new types through *restriction*; in appropriate circumstances, such types can provide the same elegance and ease of understanding, but without a corresponding linear development and maintenance cost.

As an example, consider an integer counter, intended to represent the number of occurrences of a certain event: the operations one might like to have on such an entity can be described as follows:

1. Create a new counter, with a value of zero.
2. Increment the counter by one.
3. Compare the value of the counter against a given integer.

Note that we might want to describe such operations *explicitly*, with the assumption that all other operations (for example, multiplying the counter by 8, or setting bits 2, 3 and 7), are disallowed.

One could clearly create such an object simply (and elegantly) in C++ or Java using a class construct, but the point here is that creation of such a new type would involve new, deployable, testable software with a non-trivial associated cost; a counter such as this is, mathematically and naturally speaking, a special kind of integer, and therefore we already have all the required software (built into the hardware and run-time environment) that we need. In particular, what we *really* need is a constraint: we must promise not to use disallowed "non-counter" operations on counters.

4 Open Questions

We can ask the following questions to frame future work in this area:

1. What existing common (or indeed uncommon) types naturally present themselves as *restrictions* of existing types, either built-in/primitive types or other existing types?
2. What amount of software is involved in the definition of those types?
3. (Harder) What financial cost has historically been involved in the creation and maintenance of those types?
4. What proportion of that cost might be saved by new techniques for developing restricted types?