Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation

Kurt M. Olender and Leon J. Osterweil

K. M. Olender was with the Department of Computer Science, University of Colorado, Boulder, CO 80309. He is now with the Department of Computer Science, Colorado State University, Fort Collins, CO 80523.

L. J. Osterweil was with the Department of Computer Science, University of Colorado, Boulder, CO 80309. He is now with the Department of Information and Computer Science, University of California, Irvine, CA 92717.

Abstract This paper presents a flexible and general mechanism for specifying problems relating to the sequencing of events and mechanically translating them into dataflow analysis algorithms capable of solving those problems. Dataflow analysis has been used for quite some time in compiler code optimization. It has recently gained increasing attention as a way of statically checking for the presence or absence of errors and as a way of guiding the test case selection process. Most static analyzers, however, have been custom-built to search for fixed, and often quite limited, classes of dataflow conditions. We show that the range of sequences for which it is interesting and worthwhile to search is actually quite broad and diverse. We create a formalism for specifying this diversity of conditions. We then show that these conditions can be modeled essentially as dataflow analysis problems for which effective solutions are known and further show how these solutions can be exploited to serve as the basis for mechanical creation of analyzers for these conditions.

Index Terms—Dataflow analysis, finite state machines, sequencing, specification, static analysis.

I. Introduction

The importance of assuring high quality in software has been recognized for decades. In that time progress has been made toward developing software quality tools and techniques, but it is still very difficult to develop high quality software.
and to accurately determine the quality of developed software. We believe that the most fundamental techniques required in assuring software quality are the detection of the presence of errors (the goal of most testing activities) and the determination of the absence of errors (the goal of validation and verification). Errors can be present in any software object—ranging from specifications to design objects, code, and test cases. Further, errors can be of various kinds—functional errors, errors of omission, security errors, and so forth.

We believe, however, that all errors can be viewed as inconsistencies between a specification of intent and a specification of an approach to satisfying that intent with some solution vehicle. A design error can be seen to be a specification of a solution approach which, if implemented, would produce a system whose behavior is inconsistent with the intent embodied in the requirements specification that the design was intended to address. Similarly, code errors are inconsistencies between what the design stipulates must be done and what the code actually effects. Inadequate performance, robustness, or security can similarly be seen as execution-time characteristics that are inconsistent with the requirements or design specifications of these characteristics. Unfortunately, all too often these characteristics are not made explicit or rigorous and inconsistencies are identified only in the minds of users, customers, or managers. Often these unstated specifications differ among these individuals leading to differences of opinion as to whether errors or inadequacies exist at all.

The notion of error is a relative one. All errors are inconsistencies between specifications and solutions. An erroneous solution is only erroneous with respect to the specification that it purports to satisfy. For example, testing—the process of attempting to detect the presence of errors—cannot be a definitive process unless and until there exists a suitably definitive specification of intent against which to test. Similarly, validation and verification must be based upon the existence of both the software object to be evaluated and a software object which captures the intent which the object must satisfy. Moreover, the more rigorously these objects are expressed, the more rigorous and definitive that testing, validation, or verification process can be.

Historically, testing has most often been directed at code, because code is written in a programming language with syntax and semantics that are rigorously defined at least in terms of the observed actions of the machines on which compiled and loaded versions are executed. Even in the case of testing code, however, the results are usually not as rigorous and definitive as desired due to the lack of correspondingly well defined specifications of intent. The functional intent of programs has most often been captured in terms of first order predicate calculus assertions [9], [10], [18]. In some cases whole new assertion languages have been developed to facilitate capturing functional intent [24]. To the extent that specifications of functional intent can be made complete and consistent with the intent residing in the minds of all humans concerned with the software development project, these assertions then form a firm and undeniably useful basis for functional testing, validation, and verification activities. Much has been written about the