Abstract

In this paper we describe a new formal methods tool called EVES. EVES consists of a set theoretic language, called Verdi, and an automated deduction system, called NEVER. We present an overview of Verdi, NEVER, and the underlying mathematics; and develop a small program, to demonstrate the basic functionality of EVES.

Keywords: Automated deduction, EVES, formal methods, logic of programs, NEVER, Verdi.

1 Introduction

In previous papers [Cra 87, CKM* 88] we presented our views on the putative benefits accruing from the application of verification systems, and we identified various problems with many of the existing verification systems. In particular, many verification systems have poor automated deduction support and lack demonstrable soundness. The primary purpose of the EVES project was to develop a verification system, integrating techniques from automated deduction, mathematics, language design and formal methods, so that the system is both sound and useful. The development of the prototype verification system m-EVES [CKM* 88] and various experimental applications of m-EVES allowed us to gain experience before finally committing to the particular design for EVES.

The development of EVES has followed two principal directions: (1) the design of a new specification and implementation language, called Verdi (which included the development of the formal semantics and proof system with demonstration of soundness), and (2) the implementation of a new theorem prover called NEVER.

*The development of EVES was sponsored by the Canadian Department of National Defence through DSS contract W2207-8-AF78 and various other contracts. This paper is a revised version of ORA Conference Paper CP-91-5402-43, March 1991.
In this paper, we will present an overview of various aspects of EVES, and present a flavour of EVES through a small example. Other reports (such as [Cra 90, Saa 89, Saa 90, SC 90, KP 90]) present detailed descriptions of Verdi, the underlying mathematics, and other applications of EVES (embedding a version of simple type theory in EVES, and a verified implementation of an interpreter for a small programming language).

For the purposes of this paper, a verification system consists of a language or languages that allow the formal specification and implementation of programs, a proof obligation generator, and some form of automated deduction support.

2 Verdi

Verdi [Cra 90] is a formal notation that may be used to write programs that are to be verified formally using the EVES verification system. Consequently, Verdi consists of syntactic forms for expressing specifications (what effect a program is to have), implementations (how a program is to cause an effect), and proofs (justification that a program meets its specification).

Verdi is based on a version of untyped set theory [Fra 68], which can be used to express rigorous mathematical concepts [Saa 89]. For example, Verdi can be used to prove theorems of set theory (e.g., Schroeder-Bernstein, Cantor), to prove the functional correctness of hardware designs (e.g., an n-bit adder), or to prove security-critical properties (e.g., versions of non-interference).

The syntax of Verdi is similar to the s-expressions of LISP.

While Verdi is, in general, an untyped language, typing information is necessary for syntactic forms that are to be executed. In proofs, types denote sets of values.

The Bool, Int and Char types are defined by the initial theory. The Bool type denotes the logical truth values. The Int type denotes the set of mathematical integers. The Char type denotes the ASCII character set. Other types are introduced through an enumeration declaration, an array declaration, or a record declaration. Verdi type sameness is a variant of structural sameness (since the structures of records and enumerations are not used in sameness checking). Hence, Verdi type sameness is a compromise between the extremes of name sameness and structural sameness.

The various function declarations introduce functions. Functions that are used in an executable context must have been introduced by a "typed function declaration." Such a declaration associates type information with the parameters and the result of the function. The result of evaluating an executable function application is predictable only if the application is legal. By "legal" it is meant that the parameters satisfy certain constraints. Non-executable functions may be defined recursively and, by using function recursion groups, through mutual recursion.

A procedure declaration introduces a procedure and is always executable. Procedures may be defined recursively and, by using procedure recursion groups, through mutual recursion.

An axiom declaration expresses a fact and may include heuristic information.

An expression is a Verdi sentence that can be evaluated to produce a value. The expressions are character literals, numerals, strings, identifiers (denoting variables), function applications, and quantifications.