The VERUS™ Design Verification System

Dan Putnam
Gould Inc., Software Division
Urbana, Illinois 61801

VERUS is a design verification system developed by Gould Software Division. VERUS is a general purpose system, but its primary intended application is to specify and prove security properties of systems modeled as state machines. Since VERUS is a general purpose system, the state machine methodology is not enforced by the VERUS parser and theorem prover. Instead, a state machine specification checker has been provided to ensure that specifications are correctly formulated as state machine models. Below we give an example showing how a state machine model may be expressed in VERUS.

The VERUS language is a dialect of the predicate calculus adapted to a software engineering environment. It includes parser directives (macros, file inclusion, and conditional compilation), types, definitions, and proof directives.

For simplicity, the language is small. For example, generic type constructions, such as set theory, are not part of the language. Instead, such constructions may be defined using macros that express the defining relationships in terms of the basic VERUS language.

VERUS features a prefix notation for logical operators, which, combined with proper indentation makes the precedence and scope of operators visually obvious. Readability is also enhanced by the use of the DEFINE construct, which allows simple identifiers to be used in place of complicated logical expressions.

A state machine specification expressed in VERUS consists of the following components:

1. Types that represent the domains of entities or values that constitute the system (e.g., processes, security levels).

2. Functions that represent relationships that may exist among the system entities and that collectively constitute the state of the system at any given time.

3. An initial condition predicate that describes the beginning state of the system in terms of the state functions.

4. Event predicates that define the ways in which the state of the system may change.

5. Correctness requirements that specify allowable system behavior.

6. Theorems that state that the system always meets the correctness requirements.

A state machine specification may be progressively refined until the lowest-level event predicates become detailed descriptions of the programming language procedures that implement them. Additional theorems are required to show that each refinement is correct.

The following text shows how VERUS might be used to express a state machine specification of an example system involving processes and buffers with security levels. First we declare names for three values of time to allow statements about the system at some initial time as well as before and after state changes.

```
$$ Comments follow dollar signs.

TYPE Time;
CONST INIT, OLD, NEW : Time;          $$ start, before and after

TYPE Process;
TYPE Buffer;
TYPE Level;                           $$ security level values
TYPE ProcessState = (Active, Free);

DECLARE ProcState( Time, Process ) : ProcessState;
DECLARE ProcLevel( Process ) : Level;       $$ not time-dependent
DECLARE BufLevel( Time, Buffer ) : Level;
DECLARE BelongsTo( Time, Buffer, Process ) : BOOLEAN;
```

Using this vocabulary of types and relationships, we can state initial-condition, event and correctness-requirement predicates. In this very simple example, the initial-condition predicate might state that when the system starts, every process is in the free state and no buffer belongs to any process:

```
VAR t : Time;
VAR proc : Process;
VAR buf : Buffer;

DEFINE InitialCond(t) : BOOLEAN
BY
   AND
   {
    FOR proc ProcState(t,proc) = Free;
    FOR buf FOR proc NOT BelongsTo(t, buf, proc);
   };
```

As one of the correctness requirements, we can specify that no buffer may belong to a free process and that if a buffer does belong to a process, then their levels must be equal:

```
DEFINE StateReq(t) : BOOLEAN
BY
   AND
   {
    FOR buf FOR proc
      IF ProcState(t,proc) = Free
      THEN NOT BelongsTo(t,buf,proc);

    FOR buf FOR proc
      IF BelongsTo(t,buf,proc)
      THEN BufLevel(t,buf) = ProcLevel(proc);
   };
```

An intial-condition theorem, which asserts that the system begins in a correct state can now be easily stated:

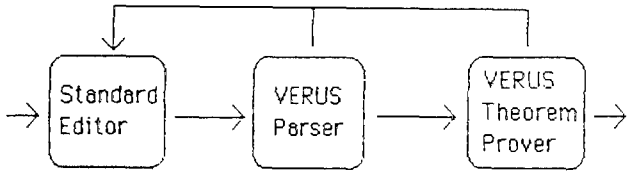
```
PROVE
   IF InitialCond(INIT)
   THEN StateReq(INIT);
```

The specification must also include event predicates, which describe state changes. Each event predicate requires a theorem stating that an occurence of that event preserves the system's correctness. The event theorems together with the initial-condition theorem constitute an inductive proof that the system is correct.

A key characteristic of the VERUS system is that theorems are stated and proved as an integral part of the specification. Each theorem is stated in a proof outline, which includes supporting statements (proof steps) and proof directives to guide the theorem prover. The proof outline strategy allows the VERUS software to be simple and easy to use.

VERUS software consists primarily of a parser and a noninteractive theorem prover. A specification checker that enforces the state machine specification methodology has also been provided. The parser performs syntax checks on a specification and creates an output file of tables and tree structures. The theorem prover reads this file and uses the embedded proof outlines to attempt to produce complete, formal proofs.

Since the VERUS theorem prover is guided by its input file, the prover is not complicated by the inclusion of interactive functions. The user composes proof outlines with a standard text editor and interacts with the prover indirectly via the parser. The parser is very fast, and separate compilation can be used to aviod unnecessary reparsing. Thus, the VERUS proof process is a cycle of editing, parsing, and proving, as illustrated below:



VERUS Proof Cycle

The theorem prover is largely automatic to compenstate for the lack of direct user interaction. Many propositions can be proved using proof outlines that simply list the relevant facts from the specification as proof steps.

Proof outlines have other advantages. When a specification requires small changes, proof outlines can easily be modified in the text editor and rerun. When, as is often the case, a specification has many theorems with similar proofs, new proof outlines can easily be created from existing outlines by making slight modifications. Finally, a proof outline can be viewed as a human-level proof that clearly exhibits the global structure of the underlying formal proof.

VERUS is a trademark of Gould Inc.