The Enhanced HDM System for Specification and Verification

Michael Melliar-Smith and John Rushby

Computer Science Laboratory
SRI International

1. Introduction

The Hierarchical Development Methodology (HDM) and its associated tools and specification language (known as SPECIAL), which SRI developed during the 70's, have been quite widely used for the specification and verification of secure systems and found useful for that purpose.

SRI is currently developing a new specification and verification environment that is substantially different from the old HDM, but builds on the experience gained with the previous system. Notable features of the new system include

- Specifications in first order predicate calculus with second order capability
- Strong type checking with overloading
- Parameterized modules with semantic constraints
- User interface based on multi-window screen-editor
- Theorem proving by reduction to propositional calculus, with decision procedures for common theories
- Hoare sentences and code proof
- Multilevel security (MLS) checking by information flow analysis

2. Man-Machine Interaction

In contrast to systems that seek to largely automate the theorem proving process, our verification system is based on an interaction between the human and the computer that utilizes what we believe to be the strengths of each. Rather than build an intelligent (or "expert") system, we have attempted to construct a system to provide support for an intelligent and expert user. We believe that our users have a comprehensive understanding of the reasons why their theorems are true, and of how to construct their proofs; the computer is excellent at reasoning in decidable domains and at recording what has been done. Over the next year or so, we hope to demonstrate the effectiveness of this approach in practice.

Since the user is an integral component of our system, the interface presented to him must be particularly convenient. The user's interaction with our system is conducted entirely in terms of his own specifications, and the system does not expose the user to formulae generated internally (such as "verification conditions"). On the mechanical level, our system interface is based on a full screen editor: the user composes and edits his specification on the screen, and invokes various system functions by means of extended editing commands. Proofs, for example, are accomplished by pointing at the appropriate proof statement and pressing the "prove" key. The system similarly returns error and status information by placing the cursor at the specification text concerned and displaying a message in a system window. The functions available to the user include parsing, prettyprinting, typechecking, and proving. Various status and information commands are also available.

The initial implementation uses a customized EMACS as its editor interface, and thereby provides a familiar environment with support for many terminal types. A more sophisticated display using an IBM PC with bit-map graphics and mouse is currently under development. This will provide a multi-window environment with structure editing and will be appropriate for remote access to conventional mainframes. This interface, and indeed the entire system, will also be supported on a Symbolics 3600.

3. Specification Language

The specification language is based on multi-sorted first-order predicate calculus and is intended to support an axiomatic rather than a constructive specification style. Such property-oriented (and possibly partial) specifications facilitate the description of just those properties relevant to the level of abstraction under consideration and are well suited to requirements definition and to specification and verification at the design level.

This is not the place to describe the language in detail, though we believe it is sufficiently perspicuous that the reader will find it easy to understand the examples given below. The language provides types, variables and constants of those types, the connectives of propositional calculus, the quantifiers of first-order predicate calculus, and also a first-order encoding of second-order expressions. Thus, functions of a particular signature are constants of a functional type. Free variables are implicitly universally quantified over the formulae in which they occur.

Specifications are structured into modules with explicit import and export lists (the USING and EXPORTING clauses, respectively). Module definitions may be parameterized by types and constants (including function constants), but must be instantiated for use elsewhere. Inside a module definition, its parameters are uninterpreted – so that generic theorems may be stated and proved. These proofs will be valid in all instantiations of the module. Actual parameters must, of course, match the type of their corresponding formal, but in many situations it is necessary that they should satisfy additional *semantic* constraints. These constraints may be stated in the ASSUMING clause of a module definition; when the module is instantiated, the formulae of the ASSUMING clause are also instantiated and constitute proof obligations in that context. The module inductions shown in the examples has three such assumptions, which are discharged when it is instantiated in module sum.

4. Proof Support

In order to prove a theorem, the user cites instances of axioms and previously-proved theorems from which the result follows by propositional reasoning. The user is expected to provide appropriate substitutions for the existentially quantified variables of the conclusion and for the universally quantified variables of the formulae cited as premisses. The prover reduces the user's proof from first (or even second) order into propositional calculus. Skolemizing as it does so. The reduced formula, which is never exposed to the user, is submitted to the underlying prover (a decision procedure for propositional calculus, linear arithmetic, and equality on uninterpreted function symbols).

The benefit of this approach is that the prover is very fast, and completely deterministic in its behavior: the outcome does not depend on *how* a formula is presented but only on its content - logically equivalent formulae always yield the same result. The disadvantage is in the quantity and detail of the information which must be supplied by the user. We are currently starting to develop proof construction and debugging aids to assist the user in these tasks. The speed of the prover is important here, since it allows the user to maintain a continuous high level of concentration.

5. Program Verification

Our conviction that the user should only be expected to reason about formulae that he himself created led us to reject the use of verification conditions for code proof, in favor of an approach based on Hoare sentences. By exploiting the module parameterization features of the language, we are able to introduce state objects (program variables), operations with side effects, and Hoare sentences into the specification language. Programming language constructs are treated as operations and their semantics are described in terms of Hoare sentences. Support for code proof in a subset of Pascal has been provided in this way, and is described in a companion paper by Friedrich von Henke.

6. MLS Flow Analysis

SRI's information-flow analyzer for SPECIAL has been widely used in analysis of the security of certain systems. The new system includes a similar tool for the new language. Both the tool and the language were designed to avoid certain flaws and inadequacies found in the earlier system. The flow-analyzer (known as the MLS checker) takes a specification module and generates a new module containing a set of formulae sufficient to establish the security (in a certain, limited, but well-defined sense) of the original specification. This automatic generation of formulae is, of course, contrary to our proclaimed intention of allowing the user to reason entirely with his own formulae. However, the derivation of the generated formulae is straightforward (and the MLS checker provides extensive commentary on their construction) and they are generated mechanically only to ensure completeness and reliability. The user is required, as always, to guide the proofs of the formulae that are generated, but these are invariably trivial and the mechanically generated proof clauses included in the module usually suffice.

7. Current Status

The present, experimental, version of the system is coded in Maclisp, uses a customized EMACS as its user-interface, and runs on DEC-20 type hardware under TOPS-20 and TENEX. All the features described here are supported, though development work continues.

The system was conceived and developed by Judith Crow, Dorothy Denning, Peter Ladkin, Michael Melliar-Smith, John Rushby, Richard Schwartz, Rob Shostak, and Friedrich von Henke.

8. Examples

THe module **sum** employs proof by induction to establish that the sum of the first n natural numbers has its well known closed-form. The induction scheme is provided by the module **inductions** and serves to demonstrate the second-order capabilities of the specification language. The module **morenats** merely serves to introduce some required functions on the natural numbers.

```
inductions: MODULE [dom: TYPE, first: dom, next: function[dom -> dom]]

        (* Defines an induction principle for domain dom,
              with successor function next *)
ASSUMING

  a, d1, d2: VAR dom

  measure: VAR function[dom -> nat]

  dom_well_founded: FORMULA
    (EXISTS measure : (FORALL a : measure(a) < measure(next(a))))

  first_has_no_pred: FORMULA NOT (EXISTS a : first = next(a))

  nonfirst_has_pred : FORMULA d2 /= first IMPLIES (EXISTS d1 : d2 = next(d1))

THEORY

  p: VAR function[dom -> bool]

  induction: AXIOM
    p(first) AND (FORALL d1 : p(d1) IMPLIES p(next(d1)))
      IMPLIES (FORALL d2 : p(d2))

END inductions
```

```
morenats: MODULE  (* Defines some properties of the natural
                     numbers additional to the interpreted theory *)

EXPORTING succ

THEORY

  n: VAR nat
  natnonnegative: AXIOM n >= 0
  succ: function[nat -> nat] = (LAMBDA n -> nat : n + 1 )
  pred: function[nat -> nat]
  natminus: AXIOM (n ~= 0) IMPLIES pred(n) = n - 1
  square: function[nat -> nat]
  square_zero: AXIOM square(0) = 0
  square_succ: AXIOM square(succ(n)) = square(n) + 2 * n + 1
  natsuccpred: LEMMA n ~= 0 IMPLIES n = succ(pred(n))

PROOF

  natproof: PROVE natsuccpred FROM natminus

END morenats
```

```
sum: MODULE  (* Proves the closed-form formula for the
                sum of the first n natural numbers *)

USING morenats, inductions[nat, 0, succ]

EXPORTING sigma    (* sigma(n) is the sum of first n natural numbers *)

THEORY

  i: VAR nat
  sigma: function[nat -> nat]
  sigma_zero: AXIOM sigma(0) = 0
  sigma_i: AXIOM sigma(succ(i)) = sigma(i) + succ(i)

PROOF

  n: VAR nat

  discharge1: PROVE
    dom_well_founded {measure <- (LAMBDA n -> nat : n)}

  discharge2: PROVE first_has_no_pred FROM natnonnegative {n <- a@*}

  discharge3: PROVE nonfirst_has_pred {d1 <- pred(d2@*)}
    FROM natsuccpred {n <- d2@*}

  basis: LEMMA 2 * sigma(0) = square(0) + 0

  basis_step: PROVE basis FROM sigma_zero, square_zero

  inductive_step: LEMMA
    2 * sigma(i) = square(i) + i
      IMPLIES 2 * sigma(succ(i)) = square(succ(i)) + succ(i)

  induction_step: PROVE inductive_step FROM sigma_i, square_succ {n <- i}

  closed_form: THEOREM 2 * sigma(i) = square(i) + i

  the_result: PROVE closed_form FROM
    p1: induction {p <- (LAMBDA n -> bool : 2 * sigma(n) = square(n) + n),
                   d2 <- i},
    basis,
    inductive_step {i <- d1@p1}

END sum
```