MOPS: Verifying Modula-2 programs specified in VDM-SL

Thomas Kaiser[1] Bernd Fischer[2] Werner Struckmann[3]

Introduction

Almost all computer programs contain errors, at least initially. The traditional approach to discover these errors is testing. However, since testing can only be used to show the presence of errors but not their *absence*, other approaches as *program verification* are pursuit. It is an exact, formal method to prove for all possible inputs the consistency between the specification of a program and its implementation. A *verification system* automates parts of the verification task. The architecture of verification systems usually comprises two different tiers, a predicate transformer or *verification condition generator*, and a prover. The verification condition generator takes the program and the specification and computes a set of logical expressions called *proof obligations*. These are then *discharged*, either automatically, by the prover, or manually, by the software engineer. If all obligations are discharged the program is proven correct with respect to the specification (assuming that the underlying calculus is sound). However, the failure to discharge an obligation does not always mean that the program contains an error. It may also indicate that the specification is incomplete or not adequate, or that the prover is too weak. The reason for the two-tiered architecture is purely pragmatic. Any specification language which is expressive enough to capture "interesting" requirements (and thus to describe "interesting" programs) is undecidable. Hence, any prover is too weak for a fully automatic system. In contrast to that, the generation of verification conditions is decidable and a fully automatic verification condition generator can be implemented, even for real programming languages.

The *Modula Proving System* (MOPS) is a Hoare-calculus based program verification system for a large subset of the programming language Modula-2 which uses VDM-SL [6] as specification language. The main goal of MOPS is to demonstrate the feasibility and viability of a Hoare-style verification system for a real imperative programming language, including pointers, arrays, and other data structures. MOPS also provides support for the modular and partial verification of large systems and includes hooks for specification-based code reuse systems as for example NORA/HAMMR [3]. Finally, MOPS demonstrates the combination of a verification system with an established specification language which exists outside the verification system itself.

MOPS is built according to the two-tiered architecture outlined above and comprises a weakest precondition predicate transformer and a rather weak rewrite-based prover; however, stronger off-the-shelf provers can be incorporated relatively easy. The predicate transformer used in MOPS supports only proofs of partial correctness, i.e., reasoning about termination cannot be done within MOPS. However, this allows us to use a simpler calculus and also yield simpler proof obligations.

[1]dvg, Postfach 72 11 07, D-30539 Hannover
[2]RIACS/NASA Ames Research Center, fisch@ptolemy.arc.nasa.gov
[3]Technische Universität, Institut für Software, Abteilung Programmierung, D-38092 Braunschweig

MOPS essentially follows the more traditional approach to verify programs after the implementation is completed instead of developing proof and program hand-in-hand, as for example advocated by the KIV-system [11]. However, we believe that the traditional approach is better suited for the incremental or even partial verification of large systems as the users can easily restrict the verification to the critical parts of a system.

The current version of MOPS supports almost the entire Modula-2 programming language as defined in [12], including pointers and data structures. The only language constructs not yet supported are variant record types, procedure types, and procedures as parameters, i.e., higher-order procedures cannot be verified. The verification of REAL-arithmetics is idealized and ignores possible rounding errors. Modula-2 also relies heavily on the use of standard libraries, e.g., for input/output, systems programming, and parallel programming. MOPS does not provide specific support for these modules but programs built on top of them can be verified as usual (except for input/output) after these modules have been re-specified using the modular verification techniques described below.

Calculus

MOPS is built upon the Hoare-calculus. The theoretical foundations and the fundamental verification algorithms based on this calculus can be found in, e.g., [1, 2, 5]. We extended these foundations into a calculus for the programming language Modula-2 by adding further proof rules and extending the underlying logic. Adding new statements to the language means adding new proof rules to the calculus. This is relatively straightforward and as long as the new rules are sound and the statements are disjoint from the core, the extended calculus remains obviously sound. Adding data types, however, extends the underlying logic and can easily compromise its soundness.

The starting point for the verification of arrays, records and pointers has been the proof system given in [10]. For MOPS, this system was extended to support explicit memory deallocation via the `DISPOSE`-procedure in the Modula-2 system module. Obviously, pointers introduce the same *aliasing problem* as arrays, i.e., a memory location can be addressed by different names. The main idea in [10] is to treat all pointers of a particular type as a single dynamic array and thus to handle pointer aliasing with the same mechanism as array aliasing. This approach, however, critically relies on Modula-2's pointer discipline which guarantees that two pointers refer to the same memory location only if one of them has—directly or indirectly—been assigned to the other. It can thus not be applied to languages as C which allow pointer arithmetics. The complete axioms and proof rules for this approach are given in [7].

Hoare-style calculi are usually defined over the classical, two-valued predicate calculus. This implies that expressions are always assumed to be defined which in turn requires all semantic functions to be total. Since MOPS uses VDM-SL as specification language, it is natural to base the calculus on the logic LPF (Logic of Partial Functions) underlying VDM-SL. This does not affect the verification condition generator; however, the proof obligations are now LPF-formulae. Semantically, this provides an encapsulation of all partiality reasoning within the proof theory for LPF or an off-the-shelf translation from LPF to the classical predicate calculus. Moreover, partial correctness becomes a stronger result than in the classical case as it implies the absence of run-time errors caused by application of partial functions to arguments outside their domain, e.g., division by zero.

Intuitively, our calculus should be sound and relatively complete with respect to LPF; we expect the formal proofs to be straightforward adaptations from the classical proofs in the literature. Obviously, however, the calculus is not relatively complete with respect to the classical predicate logic.

Specification and Verification

MOPS supports the verification of arbitrary program segments and not only, e.g., procedures or modules. This precludes considering the implementation as the final refinement of a specification module as for example in KIV but requires a direct embedding of the VDM-SL specification into the Modula-2 code. Syntactically, this is achieved by enclosing the VDM-SL expressions within formal comments `(*{` and `}*)` such that the annotated program can still be compiled and executed by any Modula-2 compiler. MOPS thus assumes the syntactic correctness of the Modula-2 program. Since the VDM-SL specification can be extracted from the annotated program automatically and shown consistent using external tools, MOPS also assumes the syntactic correctness and internal consistency of the VDM-SL specification. Such embedding approaches date back at least to the ANNA-system [9] and have also been used in the specification languages in the Larch-tradition, e.g., in the Penelope-system [4].

MOPS uses `entry`/`exit`-tags as shown below to mark the verification segments; these can be nested to break large proofs into manageable pieces. Loop invariants, which must be provided as usual in Hoare-style calculi, and additional `assert`-tags are used to aid the proof construction. Joint scoping allows the specification to refer to program variables but not vice versa.

```
...
(*{ entry sum_loop
     pre  sum  = 0
     post sum' = n * (n+1) div 2      }*)
(*{ loopinv sum = ((i - 1) * i) div 2 }*)
FOR i := 1 TO n DO
    sum := sum + i;
END;
(*{ exit sum_loop }*)
...
```

Verification segments also provide convenient hooks for specification-based retrieval as the `pre`/`post`-pair already comprises the crucial part of a retrieval query. By changing the `entry`-tag into the VDM-SL operation signature `sum_loop(n:int) ext rw sum:int` a retrieval system as NORA/HAMMR [3] (which also uses VDM-SL as specification language) can extract a full query and search a library for semantically matching, verified components. This allows a smooth integration of reuse without compromising program correctness, thus reducing the overall verification effort.

The main problem of embedding an existing specification language into a verification system (as opposed to defining a specialized behavioral interface specification language) is to define a suitable between the constructs of the implementation and specification languages. Fortunately, VDM-SL's meta-language heritage makes this task easier and most constructs (e.g., base types) map rather straightforward. Procedures, however, are slightly more complicated. A Modula-2 `PROCEDURE` with a return value and call-by-value-parameters only but without side effects can be specified via a VDM-SL *function*. Procedures with side effects are specified by *operations*; all global variables of a Modula-2 module are automatically mapped on a single *state*. Call-by-reference parameters have no direct correspondence in VDM-SL; they require generating a (local) state.

Large systems are inevitably split into several separate modules and MOPS supports the verification of such modular systems. Procedure specifications can be separated from their corresponding implementations by including them into the definition modules only. The implementations are then verified against their definitions. Client modules which import a specified procedure automatically import the associated function specification and thus need to verify only the particular call. Thus, the verification can be modularized. Figure 1 illustrates this concept.

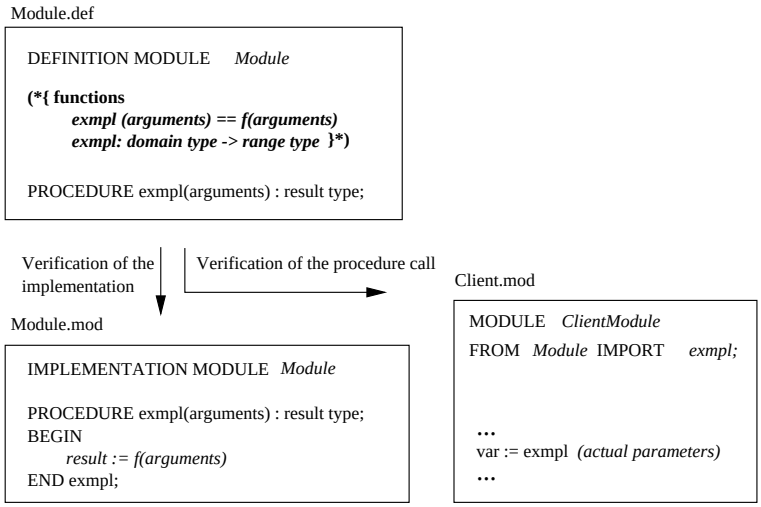

Figure 1: *Modular Verification*

If a procedure contains no call-by-reference parameters, its specification can be separated entirely from the Modula-2 declaration, even beyond the file boundary of the definition module, and moved into a completely seperated specification file containing a pure VDM-SL module. The correspondence of these files is guaranteed by extending the Modula-2 naming conventions (see figure 2). This allows a subsequent specification of existing modules, e.g., standard library modules, without any changes to the definition modules. This is required for the timestamp-based module consistency mechanism employed by most Modula-2 compilers.

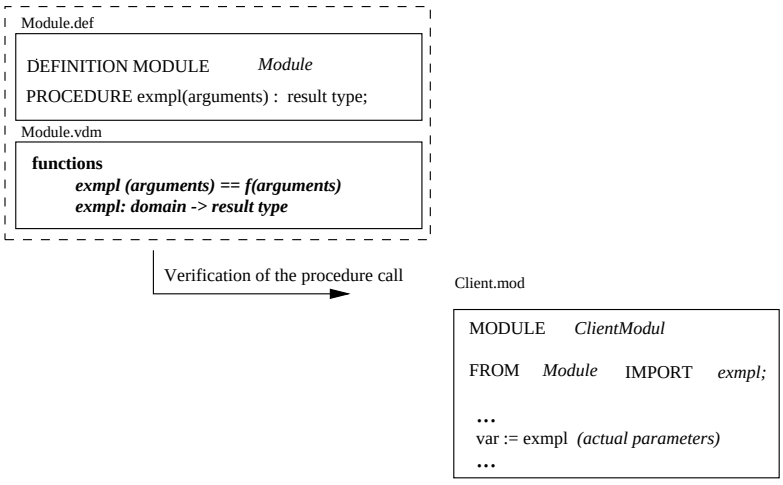

Figure 2: *Subsequent specification*

In MOPS, a Modula-2 client module can import arbitrary objects from arbitrary other modules. In particular, it can also access symbols from pure VDM-SL modules which are not associated with any definition or implementation modules. Hence, VDM-SL can be used as shared language to define theories supporting the verification.

Experiences and Conclusions

The MOPS-system is implemented in the functional programming language SML. We have tested it successfully on small and mid-size programs, including the usual sorting examples. [7, 8] contain a series of increasingly sophisticated variants of the quicksort-algorithm, including the median-of-three pivot selection strategy and the use of selection sort and bubblesort for small subarrays. The quicksort-implementations work on open arrays of element-records and sort by one of the record components. The base variant consists of more than 300 lines of Modula-2 code and VDM-SL specification. MOPS generates 23 proof obligations and discharges 14 by plain rewriting. By encapsulation of the variation into separate verification segments, the number of emerging proof obligations for the variants can generally be kept small; however, MOPS does not provide any proof management. Currently, we work on the specification and verification of the well known LZW compression and decompression algorithms [13].

MOPS has deliberately been designed as a "small tool". It combines established techniques as Hoare-style reasoning and specification-based reuse with established implementation and specification languages as Modula-2 and VDM-SL. This conceptual simplicity is—in our opinion—a major contribution of MOPS and makes it also suitable for educational purposes. Future work on MOPS includes the combination with fully automated theorem provers and the migration from the programming language Modula-2 to Java.

References

[1] K. R. Apt. Ten years of Hoare's logic: A survey—part I. *ACM Trans. on Prog. Lang. and Systems*, 3:431–483, 1981.

[2] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, New York, 1991.

[3] B. Fischer, J. M. Ph. Schumann, and G. Snelting. Deduction-based software component retrieval. In *Automated Deduction - A Basis for Applications*, pages 265–292, Dordrecht, 1998. Kluwer.

[4] D. Guaspari, C. Marceau, and W. Polak. Formal verification of Ada. *IEEE Trans. Software Engineering*, 16(9):1058–1075, 1990.

[5] B. Hohlfeld and W. Struckmann. *Einführung in die Programmverifikation.* BI-Wissenschaftsverlag, Mannheim, 1992.

[6] C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall International Series in Computer Science. Prentice Hall, New York, 2nd edition, 1990.

[7] Th. Kaiser. Behandlung von Datenstrukturen in einem VDM-basierten Prädikatentransformer für Modula-2, September 1998. Diplomarbeit, Technische Universität Braunschweig.

[8] Th. Kaiser, B. Fischer, and W. Struckmann. The Modula-2 Proving System MOPS. Informatik-Bericht Nr. 2000-01, Technische Universität Braunschweig, Mai 2000.

[9] D. Luckham, F. W. von Henke, B. Krieg-Brückner, and O. Owe. *ANNA - A Language for Annotating Ada Programs*, volume 260 of *Lect. Notes Comp. Sci.* Springer, 1987.

[10] D. C. Luckham and N. Suzuki. Verification of Array, Record and Pointer Operations in PASCAL. *ACM Trans. on Prog. Lang. and Systems*, 1(2):226–244, 1979.

[11] W. Reif. Formale Methoden für sicherheitskritische Software – Der KIV-Ansatz. *Informatik Forsch. Entw.*, 14(4):193–202, 1999.

[12] N. Wirth. *Programmieren in Modula-2.* Springer, Berlin, 1991.

[13] J. Ziv and A. Lempel. A Universal Algorithm for Data Compression. *IEEE Trans. Information Theory*, 23:337–343, 1977.