Proving system correctness with KIV 3.0

Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel

Angaben zur Veröffentlichung / Publication details:

Reif, Wolfgang, Gerhard Schellhorn, and Kurt Stenzel. 1997. “Proving system correctness with KIV 3.0.” In Automated Deduction - CADE-14: 14th International Conference on Automated Deduction, Townsville, North Queensland, Australia, July 13 - 17, 1997, proceedings, edited by William McCune, 69–72. Berlin: Springer. https://doi.org/10.1007/3-540-63104-6_10.

Proving System Correctness with KIV 3.0

Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel

Abt. Programmiermethodik
Universität Ulm, D-89069 Ulm, Germany
email: {reif,schellhorn,stenzel}@informatik.uni-ulm.de

1 Synopsis

KIV 3.0 is an advanced tool for engineering high assurance systems. It supports:

- hierarchical formal specification of software and system designs
- specification of safety/security models
- proving properties of specifications
- modular implementation of specification components
- modular verification of implementations
- incremental verification and error correction
- reuse of specifications, proofs, and verified components

KIV 3.0 provides an economically applicable verification technology. It supports the entire design process from formal specifications to executable verified code. It is ready for use, and has been tested in a number of industrial pilot applications. However, it can also be used as a pure specification environment with a proof component. Furthermore, KIV serves as an educational and experimental platform in formal methods courses. Details on KIV can be found in [8], [9] and under http://www.informatik.uni-ulm.de/pm/kiv/kiv.html.

2 System Overview

Specification and System Development. KIV relies on first-order algebraic specifications to describe hierarchically structured systems in the style of ASL, [11]: Specifications are built up from elementary first-order specifications with the operations enrichment, union, renaming, parameterization and actualization. Specifications have a loose semantics and may include generation principles to define inductive data types. Specification components can be implemented by stepwise refinement using program modules. The designer is subject to a strict decompositional design discipline leading to modular systems with compositional correctness. As a consequence, the verification effort for a modular system becomes linear in the number of its modules. Specifications and modules both have *theorem bases* attached to them. The theorem base of a specification contains the axioms, additional theorems (proved and yet unproved ones) and proofs. For a module the theorem base contains automatically generated proof obligations, which have to be proved to guarantee the correctness of the module, and again additional theorems and proofs.

Correctness Management. In KIV the user can freely create, change or delete specifications, modules, and theorems. Theorems can be proved in any order (not only bottom-up). An elaborate correctness management ensures that changes do not lead to inconsistencies. In particular it guarantees, that

- all specifications and theorems are correctly typed after changes to specifications
- there are no cycles in the proof hierarchy
- all lemmas used in a proof can be found in a theorem base of some subspecification (and have not been modified)
- only a minimal number of proofs are invalidated after modifications
- eventually all theorems and proof obligations are proved.

Interactive Theorem Proving. KIV offers an advanced interactive deduction component based on proof tactics. It combines a high degree of automation with an elaborate interactive proof engineering environment. KIV can be used for specification validation and design verification as well as for program verification. The interactive proof strategy is based on a sequent calculus. For first-order reasoning the proof tactics include rewriting, forward reasoning and induction. For the verification of implementations with imperative programs the proof strategy is based on symbolic execution and induction using Dynamic Logic ([5]).

To automate proofs KIV offers a number of *heuristics*, see [9]. Among others, heuristics for induction, unfolding of procedure calls and quantifier instantiation are provided. Heuristics can be chosen freely, and changed any time during the proof. They may be adapted to specific applications without changing the implementation. Usually, the heuristics manage to find 80 – 100 % of the required proof steps automatically.

Efficient Simplification. One highlight of KIV is its conditional rewriter. It handles hundreds and even thousands of rules very efficiently, using the compilation technique of [7] with some extensions like AC-rewriting and forward reasoning. The user chooses rewrite and simplification rules explicitly. For standard data types, which are stored in a library, sets of predefined simplifier rules are available for reuse in further projects. (The KIV library currently contains specifications for 28 data types with 217 functions and 1317 proved lemmas.) The notion of *local* vs. *global* rules allows the selection of different sets of simplification rules for different tasks. For example, the extensionality on sets, $s_1 = s_2 \leftrightarrow \forall\, x.\ x \in s_1 \leftrightarrow x \in s_2$ is used "locally" to prove the correctness of other rewrite rules about sets, but not anywhere else (i. e. "globally"), since it is normally a bad idea to replace an equation like $s = s_1 \cup s_2$ by $\forall\, x.\ x \in s \leftrightarrow x \in s_1 \vee x \in s_2$. The compilation technique sorts rewrite rules depending on their leading function symbols into a tree-like structure (discrimination nets[1]). This allows for a fast elimination of non-applicable rewrite rules. As an example, of 125 rewrite rules dealing with integers, only 21 have *abs* (absolute value) as their leading function symbol, and only 3 of these deal with

[1] To be precise: compiled nondeterministic perfect AC discrimination nets, see [6]

division. As a last step, this tree-structure is compiled into functional code for each function symbol. The resulting code is almost as efficient as a hand-coded functional program. A recursive definition of the Fibonacci-numbers rewrites fib(fib(6)) to 377 in 0.22 seconds with 776 calls to 'fib' and 2470 calls to '+' (on a SPARC 20, naturals are represented as succ(succ(...(0)))).

Proof Engineering. Frequently the problem in engineering high assurance systems is not to verify proof obligations affirmatively but rather to interpret failed proof attempts indicating errors in specifications, programs, lemmas etc. Therefore KIV offers a number of *proof engineering* facilities to support the iterative process of (failed) proof attempts, error detection, error correction and re-proof. Dead ends in proof trees can be cut off, proof decisions may be withdrawn both chronologically and non-chronologically. Unprovable subgoals can be detected by automatically generating counter examples. Another interesting feature of KIV is its strategy for proof reuse. Both successful and failed proof attempts are reused automatically to guide the verification after correction ([10]). This goes beyond proof replay (or proof scripts). We found that typically 90% of a failed proof attempt can be recycled for the verification after correction.

KIV offers a powerful graphical interface: The dialog is menu oriented, the structure of specifications and implementations is visualized with [3] as a hierarchical graph. Proofs are presented as trees, where the user can click on nodes to inspect single proof steps. Proof trees are stored in a very efficient manner on a hard disk, so that a partial proof can be continued at a later time. The system can generate LaTeX output of specifications, theorems and proofs, which is very helpful for documentation purposes. Furthermore, statistical data (about lines of specification, number of axioms, proofs, proof steps etc) is computed automatically.

3 Some Recent Applications

Access Control. A generic access control model (based on [1]) was specified (1100 lines), implemented (1200 lines), and the security model and implementation proved correct (837 proofs) in 14 weeks. See [4].

Safe Command Transfer in a GNC.[2] In cooperation with the company (intecs sistemi, Pisa) that developed the software, part of the guidance and navigation control (GNC) system of a space craft was specified and the safety requirements verified. One error was detected in the informal specification (based on ESA Software Engineering Standards).

Airbag. The decision to fire an airbag in a car is done by software in a black box. Together with the company manufacturing the boxes we formally verified that the algorithm fulfills its specification and never produces an arithmetical overflow.

[2] Part of the BSI (Bundesamt für Sicherheit in der Informationstechnik) project VSE

Compiler Verification. In [2], the compilation of PROLOG into code for the Warren Abstract Machine (WAM) is described with 11 transformation steps. Meanwhile, we have formalized and verified 6 transformation steps (with 1500 lines of specification and 700 lines of code). The most complex verification step requires an invariant which covers about three pages of text. Verification revealed several errors, see [12].

References

1. M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A Calculus for Access Control in Distributed Systems. In J. Feigenbaum, editor, *CRYPTO '91.* Springer LNAI 576, 1991.
2. Egon Börger and Dean Rosenzweig. The WAM—definition and compiler correctness. In Christoph Beierle and Lutz Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Studies in Computer Science and Artificial Intelligence.* North-Holland, Amsterdam, 1995.
3. M. Fröhlich and M. Werner. Demonstration of the interactive graph visualization system *daVinci.* In R. Tamassia and I. Tollis, editors, *DIMACS Workshop on Graph Drawing '94. Proceedings*, Springer LNCS 894. Princeton (USA), 1994. http://www.informatik.uni-bremen.de/~davinci/.
4. T. Fuchß, W. Reif, G. Schellhorn, and K. Stenzel. Three Selected Case Studies in Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report.* Springer LNCS 1009, 1995.
5. R. Goldblatt. *Axiomatising the Logic of Computer Programming.* Springer LNCS 130, 1982.
6. P. Graf. *Term Indexing.* Springer LNCS 1053, 1996.
7. S. Kaplan. A compiler for conditional term rewriting systems. In *2nd Conf. on Rewriting Techniques anf Applications. Proceedings.* Bordeaux, France, Springer LNCS 256, 1987.
8. W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report.* Springer LNCS 1009, 1995.
9. W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *Tenth Annual Conference on Computer Assurance*, IEEE press. NIST, Gaithersburg (MD), USA, 1995.
10. W. Reif and K. Stenzel. Reuse of Proofs in Software Verification. In R. Shyamasundar, editor, *Foundation of Software Technology and Theoretical Computer Science. Proceedings*, Springer LNCS 761. Bombay, India, 1993.
11. D. T. Sanella and M. Wirsing. A kernel language for algebraic specification and implementation. In *Coll. on Foundations of Computation Theory*, Springer LNCS 158. Linköping, Sweden, 1983.
12. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science (JUCS)*, 1997. to appear.