KIV 3.0 for provably correct systems

Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel

Angaben zur Veröffentlichung / Publication details:

Balser, Michael, Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel. 1999. "KIV 3.0 for provably correct systems." In Applied Formal Methods — FM-Trends 98: International Workshop on Current Trends in Applied Formal Methods, Boppard, Germany, October 7–9, 1998, edited by Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, 330–37. Berlin: Springer. https://doi.org/10.1007/3-540-48257-1_23.

KIV 3.0 for Provably Correct Systems

Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel

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

1 Synopsis

KIV 3.0 is an advanced tool for engineering high assurance systems. It provides an economically applicable verification technology, and supports the entire design process from formal specifications to executable verified code. In KIV the design process for high assurance systems proceeds as follows.

1. KIV supports both functional and state based software/system design using algebraic specifications or Abstract State Machines (ASMs), respectively. As a first step, predefined theories from a library can be imported. New specifications are added to the hierarchically structured specification graph which is graphically visualized.
2. In addition to the specification, a formal safety/security model is defined. The formulation of extra validation properties helps to detect gross specification errors before it is attempted to prove the main safety/security properties.
3. It has to be shown that the validation and safety/security properties are satisfied by the specification. The necessary formal proofs are done in an interactive graphical proof environment. Proof search is automated to a large extent. Proof engineering facilities help to reveal specification errors. After correcting the specification, invalid proofs can be reused automatically.
4. The components of the hierarchical system specification can be implemented independently (modular) using an imperative programming language. Proof obligations for the correctness of the implementation are generated automatically and have to be verified by the proof component. Again, corrected errors lead to invalidated proofs which can be reused automatically.
5. The whole specification and verification process is guarded by an elaborate correctness management. If, finally, every specification and implementation is in "proved state", it guarantees that there are no inconsistencies and all proof obligations and used lemmas are proved.
6. For use in future projects, specifications and implementations can be added to a library.

KIV 3.0 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 [17] [18] [19] and under http://www.informatik.uni-ulm.de/pm/kiv/kiv.html.

Related interactive systems with major applications in the area of correct software/system designs are ACL2 [13], PVS [14], ISABELLE [15], HOL [7], or VSE [11].

2 Structured Specifications and Implementations

For functional modelling, KIV relies on first-order algebraic specifications to describe hierarchically structured systems in the style of ASL [21]. 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 theories. State-based or reactive systems can be modelled with Abstract State Machines [9] [22], where used data types are again specified algebraically. These formalisms can be used to formulate requirements, to describe (arbitrary) systems or processes, or – more specifically – to specify software systems.

When used for a software system, KIV supports the stepwise refinement and implementation of the specification. Specification components can be implemented using program modules in an imperative programming language. 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 [16]. The correctness of each single module guarantees that the whole implementation fulfills its specification.

Structured specifications and implementations form a development graph. For each component, a theorem base exists where theorems are stored. The proof of the theorem is done relative to its component, and may use theorems of other components lower in the specification hierarchy. A lemma needed in a proof is added to the theorem base of the lowest component in the specification hierarchy where it is valid. This allows the reuse of the lemma in other proofs in other specifications as well. The hierarchical approach makes it feasible to deal with large specifications with thousands of theorems.

3 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 [6] [10].

To automate proofs, KIV offers a number of heuristics, see [18]. Among others, heuristics for induction, unfolding of procedure calls, and quantifier instantiation are provided. Heuristics can be chosen freely, and the choice is alterable any

time during the proof. So called "problem specific" heuristics are easily adaptable to specific applications without changing their implementation. Usually, the heuristics manage to find 80 – 100 % of the required proof steps automatically. First order goals can be forwarded to external automatic theorem provers [2].

KIV's simplifier handles hundreds and even thousands of conditional rewrite rules very efficiently, using discrimination nets and a compilation technique [8] [12] with some extensions like AC-rewriting and forward reasoning. As the structure of a formula assists in understanding its meaning, it is essential that simplification in KIV is structure preserving. Rewrite and simplification rules are explicitly chosen by the user. For standard data types, which are stored in a library, sets of predefined simplifier rules are available for reuse in further projects.

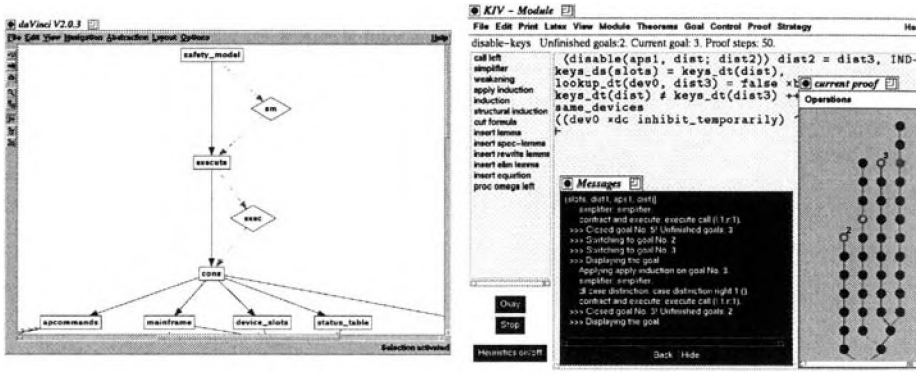

Fig. 1. Two snapshots of the system

4 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 [20]. This goes beyond proof replay (or proof scripts). We found that typically 90% of a failed proof attempt can be reused for the new proof after a correction.

KIV offers a powerful graphical interface (see Fig. 1): The dialog is menu oriented, the structure of specifications and implementations is visualized as a

hierarchical graph using daVinci [4]. Proofs are presented as trees (with colored nodes), where the user can click on nodes to inspect single proof steps. Proof trees are permanently stored in a very efficient manner, 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.

5 Correctness Management

One of the distinguished features of KIV is its correctness management. It ensures that no cyclical proof dependencies exist, and that after modifications to specifications, implementations or theorems only those proofs are invalidated that are really affected by the modification. We distinguish between local and global correctness management.

Local correctness. A theorem base stores all axioms, theorems, and proof obligations for one specification component or implementation module. Local correctness management is concerned with one theorem base only. KIV does not enforce a bottom-up proof strategy. This means that theorems can be used in a proof before they are proved themselves. Of course, it is illegal to use a theorem A as a lemma in the proof of B and at the same time B in the proof of A. To achieve this goal, it is necessary to keep track of each used lemma, rewrite rule, and simplification rule in a proof. A using graph contains the information *which theorem uses which other theorems*. This using graph must be acyclic. When beginning a proof, all theorems that (transitively) use this theorem are "hidden", i.e. they cannot be used as lemmas or rewrite rules. This approach is much more efficient than cycle checking before each application of a theorem, and, of course, much better than checking after a proof is finished. Another task of the local correctness management is to deal with the deletion or modification of a theorem and the effects on other proofs in the same theorem base. A theorem can be modified even if it is used as a lemma in a proof of another theorem. The proofs for both theorems will become invalid, which means that both theorems are considered unproved. However, the invalid proofs are kept for reuse. If all theorems of a theorem base are proved and their proofs are valid, the theorem base is locally correct.

Global correctness. A theorem L (e.g. a rewrite rule) may be used in many proofs in different theorem bases. If the theorem is modified or deleted, these proofs must be invalidated. However, KIV does not automatically store the information in which proofs a theorem is used, but for each proof which theorems it uses. The reasons are efficiency considerations and the additional problems caused by redundant information. Again for efficiency considerations, it is a bad idea to inspect every proof of every theorem base that is based on L. This could involve the inspection of hundreds of proofs.

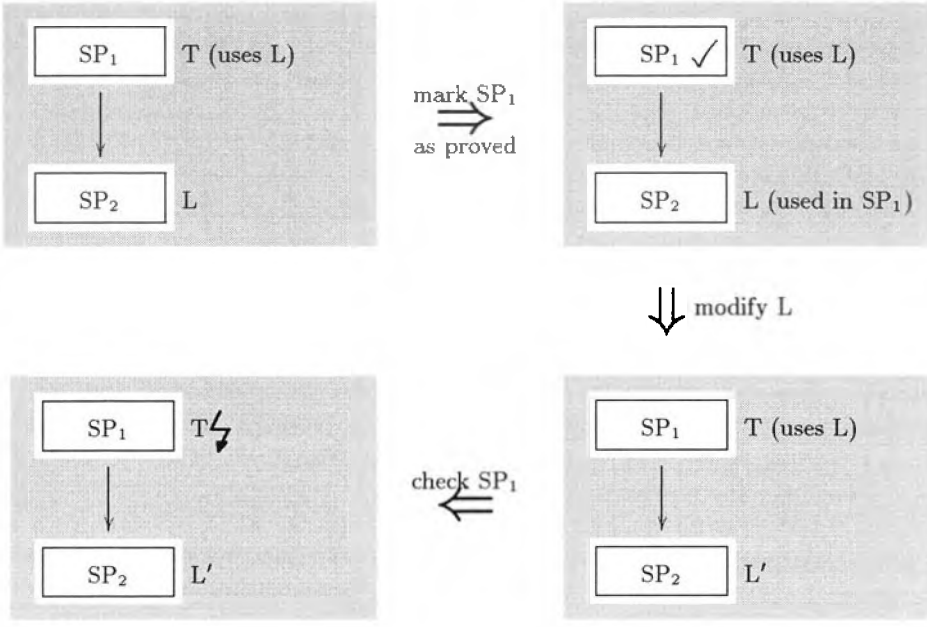

Fig. 2. Correctness Management

KIV delays the invalidation of these proofs and uses the concept of "proved states" for theorem bases. A theorem base can enter proved state if it is locally correct (i.e. all theorems are proved and all proofs are valid) and all theorems from other bases used in some proof exist (and are unchanged). Then some redundant information is recorded. In each theorem base the list of theorems that are used in some proof of the current base (that enters proved state) is stored. If a used theorem is modified or deleted this redundant informations is used to remove the base from proved state. A theorem base in proved state may not be modified. For example, adding a new theorem automatically removes it from proved state.

Consider a situation where a theorem T in SP_1 uses a lemma L from SP_2 (as depicted in the upper left corner in Fig. 2). Entering proved state for SP_1 is initiated by the user. Then the above mentioned checks are performed, and in SP_2 the information is stored that L is used in SP_1 (upper right corner in Fig. 2). It is not stored by which theorems of SP_1 L is used. Finally, SP_1 is marked as proved. If L is modified, SP_1 is removed from the proved state using the information stored in SP_2. This leads to a situation where the proof for T is not marked as invalid but in fact is invalid (lower right corner in Fig. 2). Of course, trying to enter proved state again will fail. The user can run a check on SP_1 that marks T as invalid (lower left corner in Fig. 2). This approach allows complete control over the time when complex and expensive operations are performed. The main point is that for a theorem base in proved state everything is guaranteed correct. If all theorem bases are in proved state, everything is globally correct.

6 Some Applications

Access Control. In this case study, a generic access control model for a computer system (based on [1]) was specified, implemented, and the implementation was proved correct. Furthermore, it was formalized and proved that it is not possible for a user to increase his rights without help from others. All specifications together contain about 1100 lines of text, while the efficient implementation has a size of 1200 lines of text. All in all 837 theorems and lemmas were proved. The overall time needed to complete the case study (including a vast number of modifications, error corrections, and reuses of proofs) was 14 weeks. See [5].

Digital Signatures.[1] In 1997, German Parliament passed the "Signaturgesetz" which is a legal basis for the usage of digital signatures. In order to receive a certificate for critical parts of the software and organization structures related to the application of digital signatures, a formal security model has to be provided. In this case study, it has been shown that it is possible to develop such a formal security model. The process of creating and verifying digital signatures has been formally specified. The integrity and nonrepudiation of signatures has been proved. Overall 800 lines of specification were needed.

Airbag. The decision to fire an airbag in a car is done by software in a black box. While the algorithm is very short (about two pages of text), the software runs on a micro controller with only 16bit integers. Together with the company manufacturing the boxes we formally verified that the algorithm fulfills its specification and never produces an arithmetical overflow. Another result are restrictions on the used parameters that must hold to avoid an arithmetical error. If the parameters are changed (for another type of car), only these restrictions have to be checked again.

Compiler Verification. In [3], the compilation of PROLOG into code for the Warren Abstract Machine (WAM) is described with 11 transformation steps using Abstract State Machines (ASMs, [9]). 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. During the verification, several errors were revealed in the compiler assumptions as well as in the interpreters. See [22].

Safe Command Transfer in a GNC.[1] In cooperation with the company that developed the software (intecs sistemi, Pisa), part of the guidance and navigation control (GNC) system of a spacecraft was developed formally and reevaluated in KIV 3.0. The given safety requirements have been verified, and a prototypical implementation has been proved correct. The major benefits of the formal

[1] Part of the BSI (Bundesamt für Sicherheit in der Informationstechnik) project VSE (Verification Support Environment). The prover of the VSE tool is based on KIV 1.0 and INKA.

verification were the detection of a missing case in the informal specification (based on ESA Software Engineering Standards), and the explicit (and correct) specification of implicit assumptions about the input parameters.

Dynamic Hash Tables. This case study is concerned with the verification of a store structure by dynamic (or extendable) hash tables that never need to be reorganized, even when an arbitrary number of elements is inserted, and although collision lists of bounded size are used. The verification is extraordinary difficult due to the complexity of the insert- and delete-procedure (which use many auxiliary procedures) and to the heavy use of arithmetic (including exponentiation and modulo). Details can be found in [5].

A Library of Reusable Specifications. The reuse of standard data types decreases the time needed to develop the first version of a new structured specification considerably. The specifications are correct, and contain a large set of already proved properties and rewrite rules which increases over time. Our library currently contains specifications for 37 data types with 297 functions and 1977 proved lemmas.

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. W. Ahrendt, B. Beckert, R. Hähnle, W. Menzel, W.Reif, G. Schellhorn, and P. Schmitt. Integrating of Automated and Interactive Theorem Proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume I: Foundations, chapter 3: Interactive Theorem Proving. Kluwer Academic Publishers, 1998.
3. E. Börger and D. 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.
4. 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/.
5. 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.
6. R. Goldblatt. *Axiomatising the Logic of Computer Programming*. Springer LNCS 130, 1982.
7. M. Gordon. HOL: A Proof Generating System for Higher-order Logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification and Synthesis*. Kluwer Academic Publishers, 1988.
8. P. Graf. *Term Indexing*. Springer LNCS 1053, 1996.

9. M. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.
10. M. Heisel, W. Reif, and W. Stephan. A Dynamic Logic for Program Verification. In A. Meyer and M. Taitslin, editors, *Logical Foundations of Computer Science.* Springer LNCS 363, 1989.
11. D. Hutter, B. Langenstein, F. Koob, W. Reif, C. Sengler, W. Stephan, M. Ullmann, M. Wittmann, and A. Wolpers. The VSE Development Method – A Way to Engineer High-Assurance Software Systems. In Bredereke Gotzheim, editor, *GI/ITG Tagung Formale Beschreibungstechniken für verteilte Systeme.* Universität Kaiserslautern, 1995.
12. S. Kaplan. A compiler for conditional term rewriting systems. In *2nd Conf. on Rewriting Techniques anf Applications. Proceedings.* Bordeaux, France, Springer LNCS 256, 1987.
13. M. Kaufmann and J Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4), April 1997.
14. S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Automated Deduction - CADE-11. Proceedings*, Springer LNAI 607. Saratoga Springs, NY, USA, 1992.
15. L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer LNCS 828, 1994.
16. W. Reif. Verification of Large Software Systems. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science. Proceedings.* Springer LNCS 652, 1992.
17. 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.
18. 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.
19. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications.* Kluwer Academic Publishers, 1998.
20. 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, 1993.
21. D. T. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. In *Coll. on Foundations of Computation Theory*, Springer LNCS 158. Linköping, Sweden, 1983.
22. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science (J.UCS)*, 1997. available at http://hyperg.iicm.tu-graz.ac.at/jucs/.