Proving System Correctness with KIV

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 [Rei95], [RSS95], [RS95], 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, [SW83]: 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. The structure of specifications and implementations is visualized with [FW94] as a *development graph*. An example is shown in fig. 1. In this graph, boxes correspond to algebraic specifications and arrows indicate the "is subspecification of" relation. Diamonds are program

modules with the export interface above the module, and import below. 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. The interactive proof strategy is based on induction, symbolic evaluation of definitions and programs and on simplification in first-order theories. To automate proofs, KIV offers a number of *heuristics*, see [RSS95]. These can be chosen freely, and changed any time during the proof. Heuristics may be adapted to specific applications without changing the implementation. Usually, the heuristics manage to find 80 – 100 % of the required proof steps automatically. One highlight of KIV is its conditional rewriter. It handles hundreds and even thousands of rules very efficiently, using the compilation technique of [Kap87] with some extensions like AC-rewriting or forward reasoning.

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. Proof trees can be inspected using a graphical interface (see fig. 1). Dead ends 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 ([RS95]). 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.

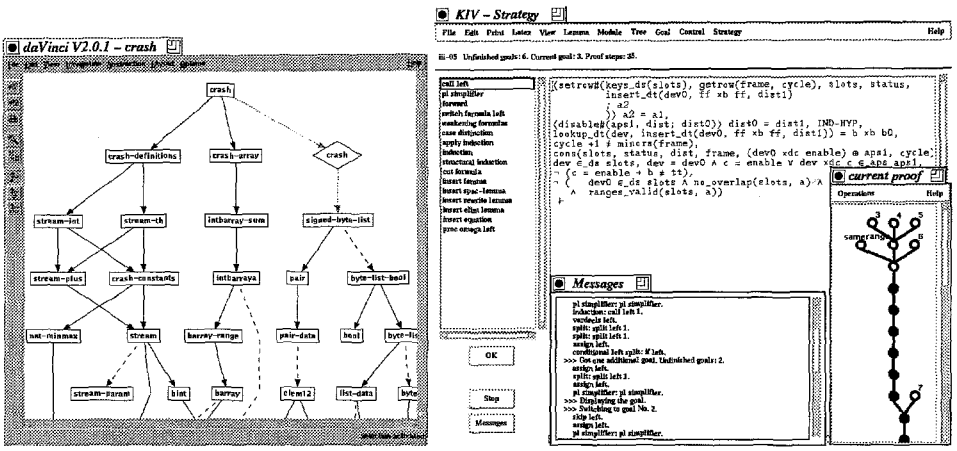

Fig. 1. Two snapshots of the system

3 Some Applications

This section lists some of the applications done with KIV. All of them made strong usage of the features described above, and, additionally, often motivated further improvements.

Safe Command Transfer in a GNC. 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 treated formally, and reevaluated in KIV 3.0 at the University of Ulm. The given safety requirements have been verified, and a prototypical implementation has been proved correct. The major benefits of the formal verification were the detection of an error in the informal specification, and the explicit (and correct) specification of implicit assumptions.

Access Control. In this case study a generic access control model (based on [ABLP91]) is specified, implemented, and the implementation is 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 reuse of proofs) was 14 weeks. See [FRSS95].

Compiler Verification. Currently we work on a case study dealing with the compilation of PROLOG into code for the Warren Abstract Machine (WAM). In [BR94], the semantics of PROLOG is defined by a simple interpreter, which is refined in 11 steps to an interpreter of WAM machine code. Meanwhile, we have formalized 6 of the 12 levels with 1500 lines of specification. The transitions between level 1/2 (PROLOG search tree vs. stack discipline, [SA96]), 2/3 (reuse of choicepoints), 3/4 (determinacy detection), 4/5 (compilation of predicate structure), and 5/6 (switching) could be proven correct. In the course of

verification several errors were revealed in the compiler assumptions as well as in the interpreters.

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 28 data types with 217 functions and 1317 proved lemmas.

A Booking System. A booking system for a national radio network was a formal redevelopment of an important part of an industrial project. The vast number of possible operations makes the specification (and implementation) large: The specification contains 3400 lines, and the implementation 7100 lines of text, of which 3600 lines where proved correct with an effort of one person year. This project is the largest single application carried out with KIV so far.

References

[ABLP91] 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.

[BR94] Egon Börger and Dean Rosenzweig. A mathematical definition of full PROLOG. *Science of Computer Programming*, 1994.

[FRSS95] 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.

[FW94] 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.

[Kap87] S. Kaplan. A compiler for conditional term rewriting systems. In *2nd Conf. on Rewriting Techniques anf Applications. Proceedings*. Bordeaux, France, Springer LNCS 256, 1987.

[Rei95] 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.

[RS95] W. Reif and K. Stenzel. Reuse of Proofs in Software Verification. In J. Köhler, editor, *Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs*. Montreal, Quebec, 1995.

[RSS95] 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.

[SA96] G. Schellhorn and W. Ahrendt. Verification of a Prolog Compiler – First Steps with KIV. Ulmer Informatik-Berichte 96-05, Universität Ulm, Fakultät für Informatik, 1996.

[SW83] 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.