Formal System Development with KIV

Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums

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

1 Overview

KIV is a tool for formal systems development. It can be employed, e.g.,

- for the development of safety critical systems from formal requirements specifications to executable code, including the verification of safety requirements and the correctness of implementations,
- for semantical foundations of programming languages from a specification of the semantics to a verified compiler,
- for building security models and architectural models as they are needed for high level ITSEC [7] or CC [1] evaluations.

Special care was (and is) taken to provide strong proof support for all validation and verification tasks. KIV can handle large scale formal models by efficient proof techniques, multi-user support, and an ergonomical user interface. It has been used in a number of industrial pilot applications, but is also useful as an educational tool for formal methods courses. Details on KIV can be found in [9] [10] [11] and under http://www.informatik.uni-ulm.de/pm/kiv/.

2 Specification Language

KIV supports both the functional and the state-based approach to describe hierarchically structured systems.

The functional approach uses higher-order algebraic specifications. The first-order part is a subset of CASL [2]. Specifications are built up from elementary 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 modules with imperative programs. 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.

For the state-based approach KIV uses abstract state machines (ASMs, [5]) over algebraically specified data types. The semantics of an ASM is the set of

traces generated by the transition relation defined by the rules of the ASM. ASMs are implemented using a compositional refinement notion. Correctness of a refinement requires the commutativity of diagrams for corresponding traces. An important instance of ASM refinement is compiler correctness, e.g. for Prolog [13]. A modularization theorem allows to decompose the correctness proof for each refinement into subproofs for arbitrary subdiagrams [4]. Correctness proofs use Dynamic Logic, a program logic for imperative programs.

In KIV, formal specifications of software and system designs are represented explicitly as directed acyclic graphs called *development graphs*. Each node corresponds to a specification component, an ASM, a module or an ASM refinement. Each node has a *theorem base* attached. Theorem bases initially contain axioms, ASM rules and automatically generated proof obligations for refinements. The theorem base also stores theorems added by the user (proved and yet unproved ones), and manages proofs and their dependencies.

3 Proof Support

In KIV, proofs for specification validation, design verification, and program verification are supported by an advanced interactive deduction component based on *proof tactics*. It combines a high degree of automation with an elaborate interactive proof engineering environment. Deduction is based on a sequent calculus with proof tactics like simplification, lemma application, and induction for first-order reasoning and a proof strategy based on symbolic execution and induction for the verification of implementations with imperative programs using Dynamic Logic [6].

To automate proofs, KIV offers a number of *heuristics* [10]. 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. Additionally, a 'problem specific' heuristic exists which is easily adaptable to specific applications. Usually, the heuristics manage to find 80 – 100 % of the required proof steps automatically.

The conditional rewriter in KIV (called *simplifier*) handles thousands of rules very efficiently, using the compilation technique of [8] with some extensions like AC-rewriting and forward reasoning. As the structure of a formula helps to understand its meaning, the KIV simplifier preserves this structure. The user explicitly chooses the rewrite and simplification rules. Different sets of simplification rules can be chosen for different tasks.

Frequently, the problem in engineering high assurance systems is not to verify proof obligations affirmatively, but rather to interpret failed proof attempts that may indicate 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. This counter example can be traced

backwards through the proof to the earliest point of failure. Thereby the user is assisted in the decision whether the goal to prove is not correct, proof decisions were incorrect, or there is a flaw in the specification. After a correction the goal must be re-proved. Here another interesting feature of KIV, the strategy for proof reuse, can be used. Both successful and failed proof attempts are reused automatically to guide the verification after corrections [12]. This goes far beyond proof replay or proof scripts. We found that typically 90% of a failed proof attempt can be recycled for the verification after correction.

The *correctness management* in KIV ensures that changes to or deletions of specifications, modules, and theorems do not lead to inconsistencies, and that the user can do proofs in any order (not only bottom up). It guarantees that only the minimal number of proofs are invalidated after modifications, that there are no cycles in the proof hierarchy and that finally all used lemmas and proof obligations are proved (in some sub-specification).

4 User Interface

KIV offers a powerful graphical user interface which has been constantly improved over the years. The intuitive user interface allows easy access to KIV for first time users, and is an important prerequisite for managing large applications. The interface is object oriented, and is implemented in Java to guarantee platform independency.

The top-level object of a development, the development graph, is displayed using daVinci [3], a graph visualization tool which automatically arranges large graphs conveniently. The theorem base, which is attached to each development node, is arranged in tables, and context sensitive popup menus are provided for manipulation. While proving a theorem, the user is able to restrict the set of applicable tactics by selecting a context, i.e. a formula or term in the goal, with the mouse. This is extremely helpful for applying rewrite rules, as the set of hundreds of rewrite rules is reduced to a small number of applicable rules for the selected context. Proofs are presented as trees, where the user can click on nodes to inspect single proof steps.

In large applications, the plentitude of information may be confusing. Therefore, important information is summarized, and more details are displayed on request. Different colors are used to classify the given information. Additionally a special font allows the use of a large number of mathematical symbols.

KIV automatically produces LaTeX documentation for a development on different levels of detail. Specifications, implementations, theorem bases, proof protocols, and various kinds of statistics are pretty printed. The user is encouraged to add comments to specifications, which are also used to automatically produce a data dictionary. As several users may work simultaneously on a large project, the documentation facilities of KIV are very important. The automatically extracted information can also be included into reports.

References

1. CCIB-98-026. *Common Criteria for Information Technology Security Evaluation, Version 2.0*. ISO/IEC JTC 1, May 1998. available at http://csrc.nist.gov/cc.
2. CoFI: The Common Framework Initiative. Casl – the CoFI algebraic specification language tentative design: Language summary, 1997. Available at http://www.brics.dk/Projects/CoFI.
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*, LNCS 894, Berlin, 1994. Princeton (USA), Springer. http://www.informatik.uni-bremen.de/~davinci/.
4. G. Schellhorn. *Verifikation abstrakter Zustandsmaschinen*. PhD thesis, Universität Ulm, Fakultät für Informatik, 1999. (to appear, in German).
5. Y. Gurevich and J. Huggins. The semantics of the c programming language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M.M. Richter, editors, *Computer Science Logic*. Springer LNCS 702, 1993.
6. D. Harel. *First Order Dynamic Logic*. LNCS 68. Springer, Berlin, 1979.
7. ITSEC. *Information Technology Security Evaluation Criteria, Version 1.2*. Office for Official Publications of the European Communities, June 1991.
8. S. Kaplan. A compiler for conditional term rewriting systems. In *2nd Conf. on Rewriting Techniques and Applications. Proceedings*. Bordeaux, France, Springer LNCS 256, 1987.
9. 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*, LNCS 1009. Springer, Berlin, 1995.
10. W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *COMPASS'95 – Tenth Annual Conference on Computer Assurance*, Gaithersburg (MD), USA, 1995. IEEE press.
11. 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, Dordrecht, 1998.
12. W. Reif and K. Stenzel. Reuse of Proofs in Software Verification. In R. Shyamasundar, editor, *Foundation of Software Technology and Theoretical Computer Science. Proceedings*, LNCS 761, pages 284–293, Berlin, 1993. Bombay, India, Springer.
13. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science (J.UCS)*, 3(4):377–413, 1997. Available at http://hyperg.iicm.tu-graz.ac.at/jucs/.