The KIV System: Systematic Construction of Verified Software

Wolfgang Reif *
University of Karlsruhe
reif@ira.uka.de

Summary

In this abstract we give a brief overview over the Karlsruhe Interactive Verifier (KIV) and sketch one of its applications: The design and verification of large modular systems.

1 The KIV System

In the KIV system the paradigm of *tactical theorem proving* is applied to realize a deduction-based programming environment for the systematic development of verified software. Basically, it provides a functional programming language PPL (Proof Programming Language), which can be used to implement both the formal and the informal aspects of rigorous program development. Examples are the design, the refinement and the administration of formal specifications, the generation of proof obligations for verification and synthesis, and the design of strategies for proof search, proof management and reuse. Potential users might be interested in implementing their own verification or synthesis strategies in PPL, in combining or comparing these with strategies that are already available, or just in applying the verification and synthesis strategies implemented by the KIV group over the last six years. Most of the verification and synthesis strategies found in the literature, are implemented in the KIV system as well as a new strategy for verifying modular systems. The KIV system is described in [HRS 88], [HRS 90], [HRS 91].

1.1 The System Architecture

The KIV system is a tactical theorem prover in the tradition of the Edinburgh LCF system, [GMW 79], or other systems like Nuprl, [Co 86], Oyster, [Bun 89], Isabelle, [Pau 86] etc. A main characteristic of tactical theorem provers is that they do not only support the

---

* Author's address: Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, Postfach 6980, W-7500 Karlsruhe, FRG, Tel. +721-608-4245. The KIV system is a joint development of the KIV group.

automation of deduction but are also designed for interactive proof engineering if the proof search gets stuck. Furthermore, tactical theorem provers support the definition, extension, and integration of proof methods. Due to these architectural features, tactical theorem provers are very successful in large applications, which cannot be tackled fully automatically. The KIV system exhibits the typical system structure: a logical formalism (Dynamic Logic, see 1.2) is embedded in a functional metalanguage (PPL, see 1.3). In this framework proof methods are represented as PPL programs constructing proofs in the underlying logical formalism. A proof method is implemented in terms of *tactics* and *strategies*. Tactics are used to define the elementary proof steps of the method, whereas strategies reflect its pragmatics. Tactics reduce goals to subgoals. Strategies control the proof search, decide how to select and to combine tactics, keep track of the still unproved subgoals, and are responsible for the interaction with the user. By adding heuristic information to the strategies, the degree of automation may be increased gradually.

1.2 The Logical Basis

Currently, the KIV system is tuned for the construction and verification of Pascal-like, imperative programs and modular systems. The specification language is first-order logic. Since correctness proofs involve both the programs and their specifications, a logic is required, where complex interrelations between programming- and specification language are expressible. Therefore, the KIV system is based on Dynamic Logic (DL, [Ha 79], [Go 82], [HRS 89], [Ste 89]) which is tailor-made for that purpose. DL extends ordinary predicate logic by formulas $[\pi]\varphi$ ("box π φ") and $\langle\pi\rangle\varphi$ ("diamond π φ"), where π is a program, and φ is again a DL-formula. The intuitive meaning of $[\pi]\varphi$ is: "if π terminates, φ holds after execution of π ". The formula $\langle\pi\rangle\varphi$ has to be read as: "π terminates and φ holds after execution of π ". The imperative programs that may occur in such program formulas are built up from skip, abort (the never halting program), assignments, conditionals, while loops, local variables and mutually recursive procedures (allowing value-, reference-, and procedural parameters). In DL many interesting properties of programs are expressible: Examples are partial correctness $\varphi \rightarrow [\pi]\psi$, total correctness $\varphi \rightarrow \langle\pi\rangle\psi$, termination $\langle\pi\rangle$true, non-termination $\neg\langle\pi\rangle$true, the equivalence of two programs π and π' with respect to a program variable x $\langle\pi\rangle x{=}x' \leftrightarrow \langle\pi'\rangle x{=}x'$, more general relations between programs $\langle\pi\rangle\varphi \rightarrow \langle\pi'\rangle\psi$, and the correctness of generic modules, [Re 91], [Re 92].

1.3 The Metalanguage

The metalanguage PPL is a typical functional language enriched by operations to construct formal proofs in DL. Proofs are represented explicitly as proof trees, and may be manipulated by forward- and backward reasoning. The basic rules of the DL calculus can be enriched by arbitrary *user-defined rules*, to support the definition of high-level, application-specific proof steps. However, *validations* have to be provided for these rules in order to guarantee soundness. Validations are proof constructing PPL programs. When called, they try to prove the applications of a user-defined rule in a proof to be sound. The explicite representation of proof trees is a very important prerequisite for proof search optimizations

using learning techniques or reusing proof experience. A special feature of the KIV system are the *metavariables*. These act as placeholders for formulas, programs, terms etc., and may be instantiated later. This feature allows to specify and to prove schematic statements, to postpone decisions, and to specify synthesis problems. For example, the specification of a synthesis problem may be expressed as a total correctness assertion $\varphi \rightarrow \langle \$C \rangle \psi$ with known precondition φ and postcondition ψ but a yet unknown program $C. $C is a metavariable which gets successively instantiated by applying synthesis tactics to the goal.

1.4 The Tool Box of Verification- and Development Strategies

Currently the KIV system offers a number of strategies for classical verification and synthesis. Examples for verification methods are Hoare's invariant assertion method [Ho 69], and Burstall's intermittent assertion method [Bu 74], [HRS 87]. For the construction of programs the KIV system offers an integration of various methods due to Gries, Dijkstra, Smith, Dershowitz, Manna and Waldinger, [HRS 88], [HRS 91], [He 91]. The most elaborate strategy of the KIV system deals with the verification of large modular systems, [Sc 89], [St 90], [Re 91]. This application is sketched in the following section. The KIV system is implemented in Lisp and runs on Sun workstations under Unix. Most of the strategies have a graphical, window oriented interface.

2 Verifying Large Software Systems with the KIV System

The systematic construction and verification of large software systems is a good example where the success of a strategy not only depends on the performance of the automated deduction system used to verify the proof obligations arising during the development. It is also extremely important to consequently pursue a strict decompositional discipline of specifying and programming, in order to bound the number and the complexity of the arising deduction problems. In the sequel we sketch the design discipline supported by the KIV system, as well as the correctness and deduction issues, [Re 91], [Re 92].

2.1 The Discipline of Specifying and Programming

Software systems are described using loose algebraic specifications. The specification language is first-order logic, and is not restricted to universal Horn clauses or equations like in [GTW 78], [EM 85]. This increases the syntactic flexibility in practical applications. The main specification problem is, that the description of a large system is large as well. Therefore, the specification cannot be designed as a monolithic block. Structuring operations are needed to break the specification down into smaller and tractable pieces, starting with the overall structure, and proceeding towards the details. This kind of structuring is usually called *horizontal structuring*. The corresponding operations are disjoint union, enlargement and actualization of parameterized specifications.

After the development of a well-structured specification, the aim is to implement the abstract notions of the specification in a conventional programming language. However, in

general, a direct implementation is not possible, since the abstract notions are too far away from those available in the target programming language. In this situation it is useful to design intermediate specifications which are closer to the implementation level, but still abstract enough to facilitate an implementation of the original specification in terms of the intermediate one. By recursively applying this technique to the intermediate specification, the "vertical" distance between the original specification and the target level can be bridged. This process of successively implementing one specification by another is called *vertical refinement*, and a single step a *vertical refinement step* or a *module*. In the KIV system additional constraints are imposed on the use of vertical refinement steps. These restrictions guarantee the following three design properties for the resulting modular systems:

- *Compatibility of the vertical- and the horizontal structure*: Vertical refinements respect the horizontal structure in the sense that different parts of a specification are implemented by seperate refinements. In this case the refinements can be developed independently.
- *Compositionality of correctness*: The correctness of a modular system can be reduced to the correctness of the single refinements. Consequently, the verification of a large system can be reduced to the verification of smaller parts. This is the key property to control the complexity of the verification task.
- *Substitutivity of correct refinements*: The correctness of a system is not affected if the implementation of one correct refinement is replaced by another.

2.2 Correctness and Automated Verification of Modular Systems

Due to the aforementioned compositionality property, the correctness of large modular systems can be reduced to the correctness of single refinements (modules). This is the key idea to make verification tractable. The correctness of single refinements is translated to a set of proof obligations in DL, which are necessary and sufficient for refinement correctness [Re 91], [Re 92]. For these proof obligations a special proof strategy has been designed and implemented in the KIV system. The strategy is based on symbolic execution and induction, and has been successfully tested in a number of case studies. It carries out the major part (80-95%) of the proof steps on its own, communicates with the user in "critical" situations, keeps track of the yet unsolved subgoals, and produces readable proof transscripts. Currently, this and other strategies are used in the national VSE project (Verification Support Environment). In this project KIV is combined with a CASE tool (EPOS), a formal specification system (SL, von Henke et al.), first-order-, and induction provers (MKRP, INKA, Siekmann et al., [EO 86], [BHHW 86]). It is applied in an industrial context (Dornier / Mercedes Benz), to produce verified software for a national radio network and verified access control software for nuclear power plants.

References

[BHHW 86] Biundo, S., Hummel, B., Hutter, D., Walther, C., The Karlsruhe Induction Theorem Proving System, 8th Int. Conference on Automated Deduction, Springer LNCS 230, 1986

[Bu 74] R. Burstall, Program Proving as Hand Simulation with a little Induction, Information Processing 74, North-Holland Publishing Company (1974)

[Bun 89] Bundy, A. Automatic Guidance of Program Synthesis Proofs. Proc. Workshop on Automating Software Design, IJCAI 89. Kestrel Institute, Palo Alto (1989), pp. 57-59

[Co 86] Constable, R. et al. Implementing Mathematics with the Nuprl Proof Development System. Prentice Hall, Englewood Cliffs (1986)

[EO 86] Eisinger, N., Ohlbach H.-J., The Markgraf Karl Refutation Procedure (MKRP), 8th International Conference on Automated Deduction, J. Siekmann (ed), Springer LNCS 230 (1986), pp. 681-682

[EM 85] Ehrig, H., Mahr, B., Fundamentals of Algebraic Specification 1, Equations and Initial Semantics, EATCS Monographs on Theoretical Computer Science, Vol. 6, Springer 1985

[GMW 79] Gordon,M/Milner,R./Wadsworth,C. Edinburgh LCF. Springer LNCS 78 (1979)

[GTW 78] Goguen, J., Thatcher, J., Wagner, E., An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, Current Trends in Programming Methodology IV, Yeh, R. (Ed.), Prentice-Hall, Englewood Cliffs, 1978, pp. 80-149

[Go 82] Goldblatt, R., Axiomatising the Logic of Computer Programming, Springer LNCS 130

[Ha 79] Harel, D., First-Order Dynamic Logic, Springer, LNCS 68, 1979

[Ho 69] Hoare, C.A.R., An Axiomatic Basis for Computer Programming, Comm. ACM 12 (1969)

[He 91] Heisel, M., Formal Program Development Using Dynamic Logic, Dissertation, University of Karlsruhe, 1991 (in German)

[HRS 87] Heisel, M., Reif, W., Stephan, W., Program Verification by Symbolic Execution and Induction, Proc. 11th German Workshop on AI, K. Morik (eds), Springer Informatik Fachbericht 152, 1987, 201-210.

[HRS 88] Heisel, M., Reif, W., Stephan, W., Implementing Verification Strategies in the KIV system, Proc. 9th International Conference on Automated Deduction, E. Lusk, R. Overbeek (eds), Springer LNCS 310 (1988), pp. 131-140

[HRS 89] Heisel, M., Reif, W., Stephan, W., A Dynamic Logic for Program Verification, Meyer, A., Taitslin, M., Logic at Botik 1989, Pereslavl-Zalessky, USSR, Springer LNCS 363

[HRS 90] Heisel, M., Reif, W., Stephan, W., Tactical Theorem Proving in Program Verification, 10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 1990, Springer LNCS 449, pp. 117-131

[HRS 91] Heisel, M., Reif, W., Stephan, W., Formal Software Development in the KIV System, in Automating Software Design, Lowry McCartney (eds), AAAI press 1991, and Proc. Workshop on Automating Software Design, IJCAI-89, Kestrel Institute, Palo Alto (1989),

[Pau 86] Paulson, L. C., Natural Deduction as Higher-Order Resolution, Journal of Logic Programming, 1986, 3, 237-258.

[Re 91] Reif, W., Correctness of Specifications and Generic Modules, Dissertation, University of Karlsruhe, 1991 (in German)

[Re 92] Reif, W., Correctness of Generic Modules, Symposium on Logical Foundations of Computer Science, Tver, USSR, Springer LNCS, 1992 (to appear)

[Sc 89] Schellhorn, G., Examples for the Verification of Modules in Dynamic Logic, Institut für Logik, Komplexität und Dedutktionssysteme, University of Karlsruhe 1989, (in German)

[St 90] Stenzel, K., Design and Implementation of a Proof Strategy for Module Verification in the KIV System, University of Karlsruhe 1990, (in German)

[Ste 89] Stephan, W., Axiomatising Recursive Procedures in Dynamic Logic, habil. thesis, University of Karlsruhe, 1998 (in German)