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...
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 $\langle \pi \rangle \varphi$ ("box $\pi \varphi$") and $[\pi] \varphi$ ("diamond $\pi \varphi$"), where $\pi$ is a program, and $\varphi$ is again a DL-formula. The intuitive meaning of $[\pi] \varphi$ is: "if $\pi$ terminates, $\varphi$ holds after execution of $\pi$". The formula $\langle \pi \rangle \varphi$ has to be read as: "$\pi$ terminates and $\varphi$ holds after execution of $\pi$". 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 $\langle \pi \rangle \text{true}$, non-termination $\neg \langle \pi \rangle \text{true}$, the equivalence of two programs $\pi$ and $\pi'$ 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 explicit representation of proof trees is a very important prerequisite for proof search optimizations