About ✓eriFun

Christoph Walther and Stephan Schweitzer

Fachgebiet Programmiermethodik
Technische Universität Darmstadt
{chr.walther,schweitz}@informatik.tu-darmstadt.de

Abstract. ✓eriFun is a semi-automated system for the verification of functional programs. It has been used so far in an industrial IT-security project concerned with electronic payment in public networks as well as for teaching semantics and verification in university courses both at the undergraduate and at the graduate level. On the development it has been attempted to achieve a high degree of automatization, to provide the system with a clear and intuitive user interface, and to care for an transparent mode of operation, as all these features strongly support the work with a system in particular for non-expert users.

1 Motivation

The motivation for the development of ✓eriFun is twofold: Since we are interested in methods for automating reasoning tasks which usually require the creativity of a human expert, we felt the need for having an experimental base of easy access which we can use to evaluate new ideas of our own and also proposals known from the literature. The second reason originates from our teaching experiences: As the motivation of the students largely increases if they can gather *practical* experiences with the principles and methods taught, ✓eriFun has been developed as a small, highly portable system, with an elaborated user interface and a simple base logic. It nevertheless allows the students to perform ambitious verification case studies within the restricted time frame of a course.

2 A Sketch of the Base Logic

The system's object language consists of a definition principle for data structures defined in the spirit of abstract data types, a definition principle for procedures based on recursion, ternary *if*-conditionals and functional composition, and a definition principle for statements (called "lemmas" in the system) about the data structures and the procedures[1]. The definition principle for lemmas allows universal quantification only and uses ternary *if*-conditionals and the truth values (of the predefined data structure `bool`) to represent connectives. The general form of a lemma is given by `lemma name <= all` x_1`:`s_1`,...,`x_n`:`s_n `body`,

[1] The present implementation does not support higher-order language features as procedures-as-parameters, etc.

F. Baader (Ed.): CADE-19, LNAI 2741, pp. 322–327, 2003.

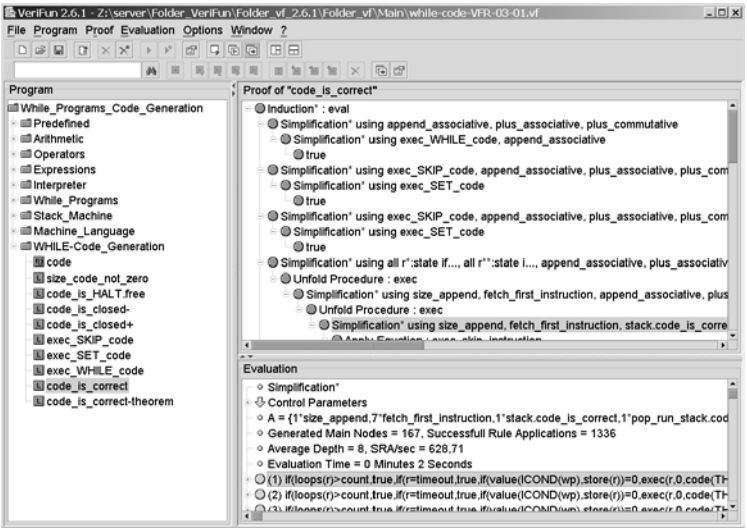

Fig. 1. ✓eriFun's Main System Window

where `body` is a boolean term build with (some of) the variables $\mathtt{x}_1,...,\mathtt{x}_n$ and the function symbols given by the data structures and the procedure definitions.

The proof of a lemma usually requires induction, and therefore a step formula defines the general format of the proof obligations processed by the system. A step formula is represented by a *sequent* $\langle H; \forall IH \vdash goal \rangle$, where H denotes the set of hypotheses, IH is the set of induction hypotheses (with universally quantified non-induction variables), and *goal*, called the *goalterm* of the sequent, represents the induction conclusion. The induction hypotheses and the goalterm are boolean terms, and the hypotheses are literals.

The set of sequents defines the language of the *HPL-calculus* (abbreviating *Hypotheses*, *Programs* and *Lemmas*), which is the calculus in which the lemmas are proved. The application of a proof rule of this calculus to a sequent yields a set of sequents, the truth of which entails the truth of the sequent to which the proof rule has been applied. A proof in the *HPL*-calculus is represented by a (proof)tree, the nodes of which are given by sequents. The root node of a prooftree for a lemma with body *goal* is given by the sequent $\vdash goal$, and the successor nodes are given by the sequents resulting from a proof rule application to the father node sequent. A prooftree is *closed* iff each leaf is a sequent with goalterm `true`, and a closed prooftree is *finished* iff each lemma which is used when building the closed prooftree is *verified*. A lemma with body *goal* is *verified* iff the system holds a finished prooftree for the sequent $\vdash goal$, and a sequent is true if the prooftree rooted in the sequent is finished.

The *HPL*-calculus provides a set of 13 proof rules to edit prooftrees. For instance, the *Induction* rule creates the base and the step cases for a sequent wrt. some induction axiom, the *Use Lemma* rule applies a lemma to a sequent, the *Apply Equation* rule modifies a sequent by the application of a (conditional)

equation, the *Case Analysis* rule performs a case analysis wrt. some caseterm, etc. Goalterms are simplified by the so-called *computed* proof rules. E.g., the *Simplification* rule rewrites a goalterm using the definitions of the data structures and procedures, the hypotheses and the induction hypotheses of the sequent and the lemmas already verified. The other computed rules perform a similar rewrite, however with restricted performance. The sequence of rewrite steps performed by a computed proof rule is called a *symbolic evaluation*.

3 Working with ✓eriFun

After starting, ✓eriFun comes up with the *Main System Window*, cf. Fig. 1. This window is separated into 3 subwindows: The *Program Window* displays the *actual program*, i.e. the system and the user defined data structures, procedures and lemmas, the *Proof Window* displays prooftrees, and the *Evaluation Window* displays symbolic evaluations. A user starts working with the system by insertion of data structures, procedure definitions and lemmas into the actual program.

Program elements are assigned a *program state* to indicate the progress of verification. The program states are displayed by colors in the *Program Window* and are automatically updated as work progresses. If the system holds a closed prooftree for a lemma, this lemma has state *verified* if this prooftree is also finished, and otherwise has state *developed*. If a non-closed prooftree exists, the lemma has state *ready*, and it is assigned the program state *ignored* if it refers to some procedure the termination of which has not been proved so far. The program states for the procedures are assigned in a similar way.

Interactive Verification. If the program state of a lemma changes to *ready* (or is immediately assigned the state *ready* upon insertion), an *initial prooftree* is generated for this lemma. The initial prooftree consists of the root node sequent $\vdash$ *goal* only, where *goal* is the body of the lemma. By applying the **Proof** command to a non-*ignored* lemma, the lemma's prooftree is displayed in the *Proof Window*, cf. Fig. 1. The user now selects some leaf of the prooftree and chooses a *HPL*-proof rule from the *Proof Rules* submenu. The system then modifies the prooftree according to the chosen rule and displays the updated prooftree in the *Proof Window*. Starting with the initial prooftree, a prooftree is developed manually by successively applying proof rules of the *HPL*-calculus to its leaves until the prooftree becomes closed.

The *computed* proof rules are implemented by an automated theorem prover, called the *Symbolic Evaluator*. When the user chooses such a proof rule, the *Symbolic Evaluator* applies the inference rules of the *evaluation calculus* to the goalterm until eventually a goalterm is obtained to which no further evaluation rule can be applied. This goalterm defines the result of the (computed) proof rule application, and the list of terms obtained defines a *symbolic evaluation* of the initial goalterm. The *Symbolic Evaluator* is a completely automated tool on which the ✓eriFun user has no control. A proofnode sequent to which a computed proof rule has been applied is assigned a symbolic evaluation of the proofnode's goalterm, which can be explored in the *Evaluation Window*, cf. Fig.1.

Semi-automated Verification. In order to support the user, √eriFun provides several automated features. The general idea is to let the system always try an automated verification. If it fails, the user has to step in to support the system by some prooftree edit or the formulation of a useful lemma. Then the system takes over control again, the user may step in another time etc., until eventually verification succeeds. By applying the **Verify** command to a lemma having an initial prooftree, the system creates a *proof plan* to develop the initial prooftree. The generation of proof plans is implemented by the so-called *next-rule-heuristic*. This heuristic decides which of the proof rules is heuristically the most successful one to be applied to a leaf of a prooftree. In case of a parameterized proof rule, the heuristic also computes the required input. E.g., if the system selects **Induction**, it also "guesses" the induction axiom and the variables to induct upon. Having developed the prooftree with the selected rule, the next-rule-heuristic is applied to all leaves created by this rule application and so on, until the prooftree becomes closed or the heuristic fails for some leaf to decide. In the latter case, the user has to select a node in the *Proof Window* for pruning some unwanted branch of the prooftree, if necessary, and then to choose a proof rule to be applied to a user selected leaf. After the prooftree is developed by the chosen proof rule, the system takes over control again by applying the next-rule heuristic to each leaf of the prooftree just developed. √eriFun provides no control commands (except disabling induction hypotheses upon symbolic evaluation), thus leaving the proof rules as the only means for the user to control the system's behavior.

Upon insertion of a new procedure into the actual program, the system's automated termination analysis is activated immediately to verify the procedure's termination. If not successful, the user has to edit some of the prooftrees belonging to the procedure's termination hypotheses, or (if the system even failed to generate termination hypotheses) has to provide a termination function. Based on the submitted termination function, the system creates a set of termination hypotheses for the procedure. To each prooftree of a termination hypothesis (being it generated automatically or after the submission of a termination function), the system applies the next-rule-heuristic. In this way, the termination of a procedure is verified similar to the verification of a lemma.

Also the work of the *Symbolic Evaluator* is supported by heuristics. E.g., the *lemma-filter-heuristic* excludes verified lemmas from the computation of a symbolic evaluation if they do not seem to contribute to a proof. This heuristic yields a significant speed up of the system (what is in particular important when working interactively), which otherwise may be swamped by a huge amount of verified lemmas in larger case studies. Equality reasoning is implemented in the *Symbolic Evaluator* by conditional term rewriting, where the orientation of the equations is also computed by appropriate heuristics.

Miscellaneous Features. The user's work is supported by several additional features. E.g., with the **Create Lemma** command, prooftrees may be duplicated in order to explore an alternative development of a proof. Program debugging

is supported by the **Refute** command which provides a means to "run" the procedures and to "evaluate" lemmas after instantiating the variables by certain terms. For working on larger case studies, ✓veriFun supports the creation and the use of *proof libraries*. Using the **Import** command, program elements and prooftrees may be imported from a file into the actual program in order to reuse previously accomplished work.

4 Applications and Availability

✓veriFun provides a high degree of automatization. Quite often a value of more than 80% is obtained as proved in several cases, e.g. [4],[8],[9][2]. ✓veriFun has been used so far in an industrial IT-security project concerned with electronic payment in public networks [3], in particular for the investigation of a public key infrastructure [2]. The system has been also used in practical courses at the graduate level, cf. [7], for proving e.g. the correctness of a first-order matching algorithm, the *RSA* public key encryption algorithm and the unsolvability of the *Halting Problem*, as well as recently in an undergraduate course about *Algorithms and Data Structures*, where more than 400 students take their first steps in computer-aided verification of simple statements about *Arithmetic*, *Linear Lists* and the verification of algorithms like *Insertion Sort* and *Mergesort*. ✓veriFun comes as a JAVA application which the students can run after a 1 MB download on their home PC (whatever platform it may use) to work with the system whenever they like to do so. A tutorial [5] gives an introduction into the use of the system, and a detailed account on the system's operation and its logical foundations can be found in a user guide [6]. This material and the ✓veriFun system are obtainable from the web [1].

References

1. http://www.informatik.tu-darmstadt.de/pm/verifun/.
2. J. Buchmann, M. Ruppert, and M. Tak. FlexiPKI - Realisierung einer flexiblen Public-Key-Infrastruktur. Technical Report TI-22/99, Theoretische Informatik, Technische Universität Darmstadt, 2003.
3. W. Stephan and R. Vogt. FairPay - Security Engineering für den elektronischen Zahlungsverkehr. In K. P. Jantke, W. S. Wittig, and J. Herrmann, editors, *Von e-Learning bis e-Payment - Tagungsband LIT'02*, Berlin, 2003. Akademische Verlagsgesellschaft.
4. C. Walther and S. Schweitzer. A Machine Supported Proof of the Unique Prime Factorization Theorem. Technical Report VFR 02/03, Programmiermethodik, Technische Universität Darmstadt, 2002.

[2] This is a rough estimate. For problems from number theory, the value may drop below 70% in extreme cases. On smaller case studies, like soundness and completeness of a tautology checker or the usual sorting algorithms, the automatization degree is even more than 90%. The automatization degree is defined by the ratio $\frac{\textbf{automated } \textit{HPL}\text{-proof rule applications}}{\textbf{all } \textit{HPL}\text{-proof rule applications}}$, when all required lemmas are specified.

5. C. Walther and S. Schweitzer. The √veriFun Tutorial. Technical Report VFR 02/04, Programmiermethodik, Technische Universität Darmstadt, 2002.
6. C. Walther and S. Schweitzer. √veriFun User Guide. Technical Report VFR 02/01, Programmiermethodik, Technische Universität Darmstadt, 2002.
7. C. Walther and S. Schweitzer. Verification in the Classroom. Technical Report VFR 02/05, Programmiermethodik, Technische Universität Darmstadt, 2002.
8. C. Walther and S. Schweitzer. A Machine-Verified Code Generator. Technical Report VFR 03/01, Programmiermethodik, Technische Universität Darmstadt, 2003.
9. C. Walther and S. Schweitzer. A Verification of Binary Search. In D. Hutter and W. Stephan, editors, *Mechanizing Mathematical Reasoning: Techniques, Tools and Applications*, volume 2605 of *LNAI*, pages 1–18. Springer-Verlag, 2003.