A Logic Teaching Tool Based on Tableaux for Verification and Debugging of Algorithms*

Rafael del Vado Vírseda,
Eva Pilar Orna, Eduardo Berbis, and Saúl de León Guerrero

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
`rdelvado@sip.ucm.es`,
{`evapilar.orna,eberbis,sauldeleonguerrero`}`@gmail.com`

Abstract. While logic plays an important role in several areas of Computer Science (CS), most educational software developed for teaching logic ignores their application in a more large portion of the CS education domain. In this paper we describe an innovative methodology based on a logic teaching tool on semantic tableaux to prepare students for using logic as a formal proof technique in other topics of CS, such as the formal verification and the declarative debugging of imperative programs, which are at the basis of a good development of software.

Keywords: Logic teaching software, Tableaux, Verification, Debugging.

1 Motivation

Computer Science universities often teach a first year undergraduate course on mathematical logic. The syllabus of the course usually includes syntax and semantics of propositional and predicate logic, as well as some proof systems such as natural deduction, resolution, and semantic tableaux. In some cases, there is also some lecture devoted to explain basic concepts on logic programming and practical work using a *Prolog* interpreter.

Most students find the high degree of rigour required in the learning of these contents daunting. In order to provide learning support to our students, proof visualization tools are always helpful. Many tools for teaching logic have been developed in the last two decades (see `http://www.ucalgary.ca/aslcle/logic-courseware`). Most of these tools concern the construction of proofs in a formal logic using semantic tableaux [2,5,7]. A *semantic tableau* [3] is a semantic but systematic method of finding a model of a given set of formulas Γ. A semantic tableau is a refutation system in the sense that a theorem φ is proved from Γ by getting its negation $\Gamma \Vdash \neg\,\varphi$.

* This work has been partially supported by the Spanish projects TIN2008-06622-C03-01 (FAST-STAMP), S2009/TIC-1465 (PROMETIDOS), UCM-BSCH-GR58/08-910502 (GPD-UCM), and PIMCD 2010/97 (Project for the Innovation and Improvement of the Educational Quality).

While logic plays an important role in several areas of Computer Science, most of the didactic software developed for teaching logic ignores the application of logic in other topics of Computer Science. We believe that our students could obtain more benefits from the techniques they learn with these tools, thanks to the possibility of applying them in a variety of contexts in advances courses. Hence, there is a need for a prototype tool allowing experiments on teaching logic in a more large portion of the Computer Science education domain, where the language and the implementation should be accessible enough and popular to ensure that they will be used into the future, and that they remain available in other courses. These motivated us to write this paper.

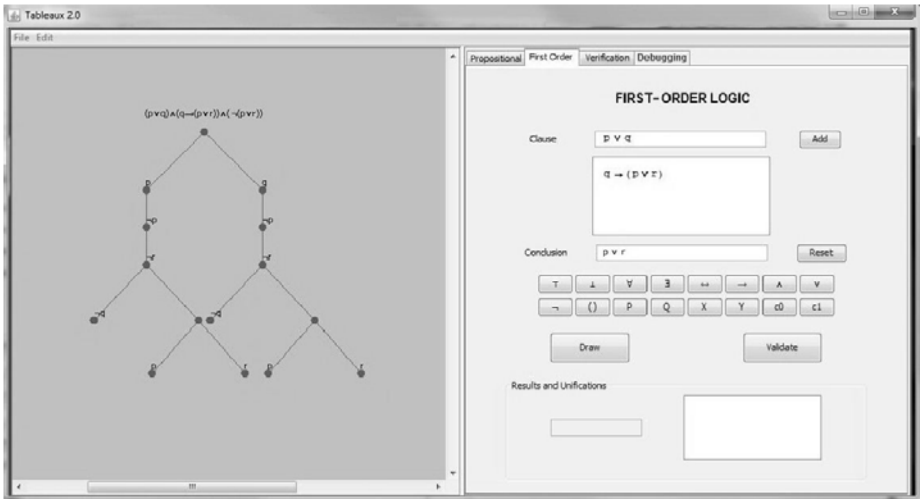
The aim of this work is to describe an innovative methodology based on a logic teaching tool on sematic tableaux, called `TABLEAUX`, to prepare our students for using logic as a formal proof tool in other areas of Computer Science, with a special emphasis on the design of algorithms and software engineering. Good algorithm design is crucial for the performance of all software systems. For this reason, an ability to create and understand formal proofs is essential for correct program development.

The major contribution of this paper is the development of new tableau methods that give semantically rich feedback to our students for the formal verification and the algorithmic debugging of programs. In this sense, `TABLEAUX` shows to be a good tool for (a little more) advances students, whose logical skills go beyond the rudiments that the user-level interaction with other logic teaching tools can develop. For instance, our tool is used for logic-based methodologies, such as program derivation, reasoning from specifications and assertions, loop invariants, bound functions, etc. This includes topics in areas whose skills and concepts are essential to programming practice independent of the underlying paradigm, as the analysis and the design of correct and efficient algorithms [4].

2 The Logic Teaching Tool

Solving logical exercises is usually done with pen and paper, but educational tools can offer useful pedagogical possibilities. The role of the educational software is to facilitate the student's grasp of the target procedures of education, and to provide teamwork and communication between teachers and students.

Our logic teaching tool `TABLEAUX` (see `http://www.fdi.ucm.es/profesor/rdelvado/TICTTL2011/`) is a prototype of an educational application based on propositional and first-order semantic tableaux with equality and unification [3] used as a support for the teaching of deductive reasoning at a very elementary university level for Computer Science students. This tool helps our student to learn how to build semantic tableaux and to understand the philosophy of this proof device using it not only to establish consistency/inconsistency or to draw conclusions from a given set of premises but also for verification and debugging purposes as we propose in this paper. Our first year students have learnt tableau calculus in the classroom and this software has helped them to easily understand advanced concepts and to produce their own trees.



Clause: p ∨ q

q → (p ∨ r)

Conclusion: p ∨ r

Fig. 1. The logic teaching tool `TABLEAUX`

2.1 Tool Usage

The tool consists of two main parts: one that produces tableaux and another based on the tableaux method for verification and debugging of algorithms. In both cases, the application possesses a drawing window where the trees will be graphically displayed. The structure of `TABLEAUX` is shown in **Fig. 1**. The user interacts with the provers through a graphical interface. We have chosen *Java*, but is possible to use *Prolog* to write the provers because its declarative character can give us a natural way to write the operations involved in the implementations (see [3] for more details).

2.2 Tool Implementation

An important design consideration in the tool implementation is that the code must be easy to maintain and extent, guaranteeing its future development and support in a sufficiently large portion of the Computer Science education domain. We have made the choice of an open source *Java* code, facilitating the addition of new features for the verification and debugging of algorithms, and enabling changes to the tableau ruleset to accommodate these new methods and applications. This makes `TABLEAUX` more interesting for an educator to invest in the application and extension of this tool.

Specific details on the straightforward aspects of a tableaux tool's development are described in [2,5,7]. We have selected the following aspects for a flexible and declarative representation of formulas and tableau rules:

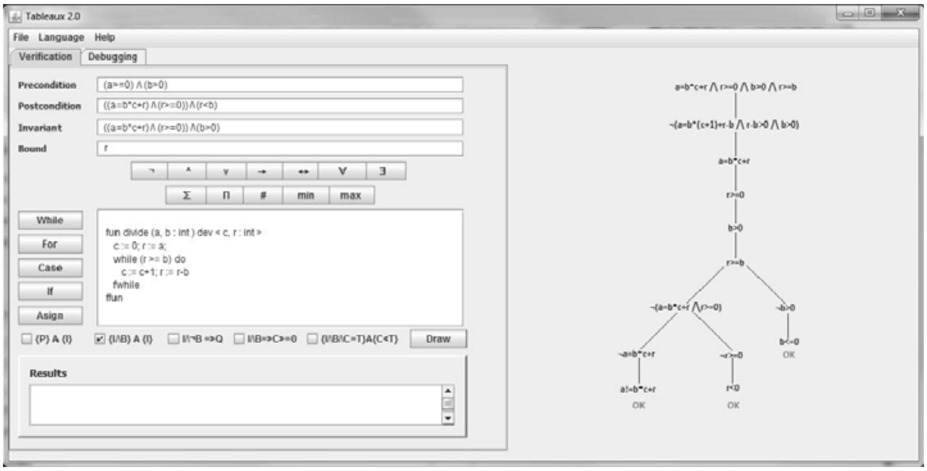
- **Parsing and tokenizing formulas:** We have set up the tool in a declarative way defining all symbols that can be part of a well-formed formula in a symbol library and a graphical interface (see **Fig. 1.**). The symbol library that is available to create formulas is declarative and extensible. The basic building block of the tool is the formula, represented internally as a parse that holds the formula's syntactic structure. By changing or extending the recursive definition and the symbol library, it is easy to expand the set of symbol strings accepted as well-formed to include *Hoare logic*, which is the basis for program verification.

- **Automatic tableau constructor:** The current implementation of the automatic prover built into the tool is straightforward and similar to other tableaux tools [2,5,7]. The automatic prover checks the rules applicable for a branch in the tableau, and selects the best one using a simple heuristic. Adapting the prover to give new alternative proofs for verification and debugging is explained in the following sections.

3 Verification of Algorithms

The main novelty of the `TABLEAUX` tool is to train our students in the art and science of specifying correctness properties of algorithms and proving them correct. For this purpose, we use the classical approach developed by Edsger W. Dijkstra and others during the 1970s [1]. The proof rules (semantics) of the algorithm notation used in this paper (see [4] for more details) provides the guidelines for the *verification of algorithms* from specifications. We use Edsger W. Dijkstra's guarded command language to denote our algorithms. Algorithms A are represented by functions `fun` A `ffun` that may contain variables (x, y, z, etc.), value expressions (e) and boolean expressions (B), and they are built out of the skip (`skip`) and assignment statements (x `:=` e) using sequential composition (S_1`;`S_2), conditional branching (`if` B `then` S_1 `else` S_2 `fif`), and `while`-loops (`while` B `do` S `fwhile`). This language is quite modest but sufficiently rich to represent sequential algorithms in a succinct and elegant way.

It becomes obvious that neither tracing nor testing can guarantee the absence of errors. To be sure of the correctness of an algorithm one has to prove that it meets its *specification* [4]. A specification of an algorithm A consists of the definition of a *state* space (a set of program variables), a *precondition* P and a *postcondition* Q (both predicates expressing properties of the values of variables), denoted as $\{P\}\ A\ \{Q\}$. An algorithm together with its specification is viewed as a theorem. The theorem expresses that the program satisfies the specification. Hence, all algorithms require proofs (as theorems do). Our tool verify algorithms according to their specification in a constructive way based on semantic tableaux $P \Vdash \neg\, wp(A, Q)$, where $wp(A, Q)$ is the *weakest precondition* of A with respect to Q, which is the 'weakest' predicate that ensures that if a state satisfies it then after executing A the predicate Q holds (see [4] for more details). We can use `TABLEAUX` to mechanize many of the boring and routine aspects of this verification process.



Precondition (a>=0) ∧ (b>0)
Postcondition ((a=b*c+r) ∧ (r>=0)) ∧ (r<b)
Invariant ((a=b*c+r) ∧ (r>=0)) ∧ (b>0)
Bound r

```
fun divide (a, b : int ) dev < c, r : int >
  c := 0; r := a;
  while (r >= b) do
    c := c+1; r := r-b
  fwhile
ffun
```

☐ {P} A {I} ☑ {I∧B} A {I} ☐ I∧¬B =>Q ☐ I∧B=>C>=0 ☐ {I∧B∧C=T}A{C<T}

Fig. 2. The logic teaching tool `TABLEAUX` for verification of algorithms

As an illustrative example, we consider the formal verification of an algorithm to compute the positive integer division (quotient and remainder), specified as:

```
{ $P\ :\ a \geq 0\ \wedge\ b > 0$ }
fun  divide ( $a, b\ :\ int$ )  dev  $< c, r\ :\ int >$
   $c\ :=\ 0;\quad r\ :=\ a;$
  { $I\ :\ a = b * c + r\ \wedge\ r \geq 0\ \wedge\ b > 0\ ,\ C : r$ }
  while  $r \geq b$  do
             $c\ :=\ c + 1;\quad r\ :=\ r - b$
  fwhile
ffun
{ $Q\ :\ a = b * c + r\ \wedge\ r \geq 0\ \wedge\ r < b$ }
```

Following [4], the verification is based on a *loop invariant* I (supplied by a human or by some invariant-finding tool), a *bound function* C (for termination), and the following five proofs:

- $\{P\}\ c := 0\,;\, r := a\ \{I\}$.
- $\{I\ \wedge\ r \geq b\}\ c := c + 1;\ r := r - b\ \{I\}$.
- $I\ \wedge\ r < b \Rightarrow Q$.
- $I\ \wedge\ r \geq b \Rightarrow C \geq 0$.
- $\{I\ \wedge\ r \geq b\ \wedge\ C = T\}\ c := c + 1;\ r := r - b\ \{C\ <\ T\}$.

Our tool represents each of these proofs as a *closed semantic tableau* (✓). We assume the reader is familiar with the classical tableau-building rules (α and β), equality ($=$), and closure rules (see [3] for more explanations). We also use the notation $R^{e,\cdots}_{x,\dots}$ to represent the assertion R in which x is replaced by e, etc. For example, we have the following proof (see also **Fig. 2.**) to verify the preservation of the invariant in the body of the loop $\{I\ \wedge\ r \geq b\}\ c := c + 1;\ r := r - b\ \{I\}$:

$I \wedge r \geq b \Vdash \neg\, wp\,(c := c+1\,;\, r := r-b, I) \Leftrightarrow I \wedge r \geq b \Vdash \neg\,(I\,{}^{c+1,\,r-b}_{c,\,r})$:

$$\begin{array}{lll}
(1) & a = b*c+r \wedge r \geq 0 \wedge b > 0 \wedge r \geq b & \{I \wedge r \geq b\} \\
(2) & a = b*c+r & (\alpha, 1) \\
(3) & r \geq 0 & (\alpha, 1) \\
(4) & b > 0 & (\alpha, 1) \\
(5) & r \geq b & (\alpha, 1) \\
(6) & \neg\,(a = b*(c+1)+r-b \wedge r-b \geq 0 \wedge b > 0) & \{\neg\,(I\,{}^{c+1,\,r-b}_{c,\,r})\} \\
\hline
\end{array} \; (\beta, 6)$$

$$\begin{array}{ccc}
(7)\;\; a \neq b*c+r & (8)\;\; r < b & (9)\; b \leq 0 \\
\checkmark\; (2,7) & \checkmark\; (5,8) & \checkmark\; (4,9)
\end{array}$$

We can use the tool to guide our students to obtain loop invariants from specifications. For example, if we only provide to our students the postcondition Q, they usually infer only an incomplete assertion $I'\ :\ a = b*c+r$ as the loop invariant. Then, when they apply the tool to verify the algorithm, they obtain an *open semantic tableau* ($\times$) for $I' \wedge r < b \Rightarrow Q$:

$$\begin{array}{lll}
(1) & a = b*c+r \wedge r < b & \{I' \wedge r < b\} \\
(2) & a = b*c+r & (\alpha, 1) \\
(3) & r < b & (\alpha, 1) \\
(4) & \neg\,(a = b*c+r \wedge r \geq 0 \wedge r < b) & \{\neg Q\} \\
\hline
\end{array} \; (\beta, 4)$$

$$\begin{array}{ccc}
(5)\;\; a \neq b*c+r & (6)\;\; r < 0 & (7)\; r \geq b \\
 & \Downarrow & \\
\checkmark\; (2,5) & \times & \checkmark\; (3,7) \\
 & \Downarrow &
\end{array}$$

We need to insert $\boxed{r \geq 0}$ in I' to close the tableau

From the open branch, our students learn to complete the invariant with $I''\ :\ a = b*c+r \wedge r \geq 0$. However, they still have an open tableau for $\{I'' \wedge r \geq b \wedge C = T\}\ c := c+1;\ r := r-b\ \{C < T\}$:

$$\begin{array}{lll}
(1) & a = b*c+r \wedge r \geq 0 \wedge r \geq b \wedge r = T & \{I'' \wedge r \geq b \wedge C = T\} \\
(2) & a = b*c+r & (\alpha, 1) \\
(3) & r \geq 0 & (\alpha, 1) \\
(4) & r \geq b & (\alpha, 1) \\
(5) & r = T & (\alpha, 1) \\
(6) & r-b \geq T & \{\neg\,(C < T)^{c+1,r-b}_{c,r}\} \\
(7) & b \leq 0 & (=, 5, 6) \\
 & \Downarrow &
\end{array}$$

$\times \Rightarrow$ We need to insert $\boxed{b > 0}$ in I'' to close the tableau

Finally, they learn to insert $b > 0$ in the assertion I'' to complete the loop invariant I. If they apply the tool again, all the tableaux remain closed and the formal verification session finishes.

4 Algorithmic Debugging

Debugging is one of the essentials parts of the software development cycle and a practical need for helping our students to understand why their programs do not work as intended. In this section we apply the ideas of *algorithmic debugging* [6] as an alternative to conventional approaches to debugging for imperative programs. The major advantage of algorithmic debugging compared to conventional debugging is that allows our students to work on a higher level of abstraction. In particular, we have successfully applied our tool based on semantic tableaux for the algorithmic debugging of simple programs to show how one can reason about such programs without operational arguments. Following a seminal idea from Shapiro [8], algorithmic debugging proposes to replace computation traces by *computation trees* with program fragments attached to the nodes. As novelty, in this work we propose to use computation trees as semantic tableaux. As an example, we alter the code of the previous algorithm with two mistakes:

```
{ $P$ : $a \geq 0 \wedge b > 0$ }
fun divide ( $a, b$ : int ) dev < $c, r$ : int >
   $c := 0;$ $\boxed{\texttt{r := 0}}$;                                  <-- wrong code!
  { $I$ : $a = b*c+r \wedge r \geq 0 \wedge b > 0$ , $C : r$ }
  while $r \geq b$ do $\boxed{\texttt{r := r - b}}$ fwhile             <-- missing code!
ffun
{ $Q$ : $a = b*c+r \wedge r \geq 0 \wedge r < b$ }
```

If we try to verify this erroneous algorithm, we can execute again the tool. Now, TABLEAUX displays an open tableau $P \Vdash \neg I$ for debugging $\{P\}\, c := 0; r := 0\, \{I\}$, instead of $P \Vdash \neg(I^{0,0}_{c,r})$. However, the weakest precondition $I^{0,0}_{c,r}$ is built from (5) and (6), step by step, to identify erroneous parts of the code in open branches:

$$\begin{array}{lll}
(1) & a \geq 0 \wedge b > 0 & \{P\} \\
(2) & a \geq 0 & (\alpha, 1) \\
(3) & b > 0 & (\alpha, 1) \\
(4) & a \neq b*c+r \vee r < 0 \vee b \leq 0 & \{\neg I\}
\end{array}$$

$$(\beta, 4)$$

$$\begin{array}{lll}
(5)\ a \neq b*c+r & (6)\ r < 0 & (7)\ b \leq 0 \\
\mid & \mid & \\
\boxed{\texttt{c := 0}} & \boxed{\texttt{r := 0 (or a)}} & \checkmark\ (3,7) \\
\mid & \mid & \\
(5)\ a \neq r & (6)\ 0\ (or\ a) < 0 & \\
\mid & & \\
\boxed{\texttt{r := 0}} & \checkmark\ or\ (\checkmark(2,6)) & \\
\mid & & \\
(5)\ a \neq 0 & & \\
\Downarrow & &
\end{array}$$

$\times \Rightarrow$ We must replace r := 0 by $\boxed{\texttt{r := a}}$ to close the tableau

After this correction, we obtain a closed tableau. Now, we can execute again the tool to perform the algorithmic debugging of $\{I \wedge r \geq b\}\ r := r - b\ \{I\}$:

$$\begin{array}{lll}
(1) & a = b * c + r \wedge r \geq 0 \wedge b > 0 \wedge r \geq b & \{I \wedge r \geq b\} \\
(2) & a = b * c + r & (\alpha, 1) \\
(3) & r \geq 0 & (\alpha, 1) \\
(4) & b > 0 & (\alpha, 1) \\
(5) & r \geq b & (\alpha, 1) \\
(6) & a \neq b * c + r \vee r < 0 \vee b \leq 0 & \{\neg I\}
\end{array}$$

$$\begin{array}{ccc}
\hline
\multicolumn{3}{r}{(\beta, 6)} \\
(7)\ a \neq b * c + r & (8)\ r < 0 & (9)\ b \leq 0 \\
| & | & \\
\|\boxed{\texttt{r := r - b}} & \|\boxed{\texttt{r := r - b}} & \checkmark\ (4,9) \\
| & | & \\
(7)\ a \neq b * (c - 1) + r & (8)\ r < b\ \ \checkmark(5,8) & \\
\Downarrow & & \\
\times \Rightarrow \text{We must insert } \boxed{\texttt{c := c + 1}} \text{ to close with (2)} & &
\end{array}$$

To close the open branch, we infer that we need to insert new code. This particular incompleteness symptom could be mended by placing $c := c + 1$ in the body of the loop. If we apply again the tool, no more errors can be found and the five tableaux remain closed. The debugging session has finished.

5 An Educational Experience with `TABLEAUX`

The prototype of the educational tool `TABLEAUX` is available for the students of the topics *Computational Logic* and *Methodology and Design of Algorithms* in the Computer Science and Software Engineering Faculty of the Complutense University of Madrid through its Virtual Campus. The following results are based on the statistics from the 186 students who took the course in 2009/2010.

5.1 Design of the Experiences

We have carried out two educational experiences:

- One **non-controlled** experience: All the students may access the Virtual Campus and participate freely in the experience: download and use the tool, and answer different kinds of tests.
- One **controlled** experience: Two groups of students must answer a test limited in time and access to material.

With respect to the **non-controlled** experience, the students may freely access the Virtual Campus without any restriction of time or material (slides, bibliography, and the tool) and answer the questions of several tests. For each of the following topics in Computer Science and Software Engineering we have provided a test that evaluates the knowledge of our students applying different kinds of semantic tableaux. The students may use these tests to verify their understanding of the different concepts. The questions are structured in three blocks: *propositional and predicate logic*, *specification and verification* of algorithms, and *debugging and derivation* of imperative programs. The resolution of the tests by the students is controlled by the Virtual Campus with the help of an interactive tutoring system. In the **controlled** experience we try to evaluate more objectively the usefulness of the tool. In particular we have chosen the application of TABLEAUX for the verification and debugging of simple searching and sorting algorithms [4]. We have chosen two groups of students answering the same questions: approximately half of the students works only on the slides of the course and the books at class; and the other half works only with the tool at a Computer Laboratory.

5.2 Results

Non-controlled Experience: We outline here the main conclusions from the results of the **non-controlled** experience. With respect to the material the students used to study, as long as the exercises were more complicated the use of the tool (simulations, cases execution, and tool help) increased considerably. Better results were obtained in the verification and debugging of searching and sorting problems (linear and binary search, insertion and selection sort). The tool helped our students to visualize array manipulations in array assignments. In the rest of the algorithms (slope search and advanced sorting algorithms) they used only the class material or bibliography. When answering the tests questions, the students were also asked whether they needed additional help to answer them. In the case of linear and binary search they used the tool as much as the class material, which means that visualization of their own proof tableaux were a useful educational complement. We can conclude that the students consider the tool as an interesting material and have used it to complement the rest of the available material.

Controlled Experience: The **controlled** experience was carried out with 59 students. We gave 32 of them only the slides of the course and the books of the bibliography [3,4]. The rest were taken to a Computer Laboratory, where they could execute the TABLEAUX tool. We gave the same test to both groups, consisting of 18 questions, 12 of them on specification aspects of the algorithms (inference of invariants and bound functions), and the rest on their verification and debugging from the code. In **Fig. 3.** we provide the means and the standard deviations of correct, errors, and *don't knows* answers. First, we observe that students using the TABLEAUX tool answer in mean more questions than the other ones. In addition, they make less errors than the others. This is due to the fact that most of the students of the *tableaux/tool* group perform the analysis of the algorithms directly from the corresponding semantic tableau, while the *slides/book* group had to hardly deduce it directly from the code. All the students who used the TABLEAUX tool indicated the benefits of using tableaux to understand the code of the algorithms from their specifications. Therefore, we can conclude that the methodology proposed in this work constitutes a good

	correct		errors		don't knows	
	mean	σ	mean	σ	mean	σ
slides/books	9.36	2.35	6.23	2.37	3.21	2.82
tableaux/tool	11.82	2.97	4.81	2.10	1.22	1.73

Fig. 3. Means and standard deviations (σ) of the controlled experience

complement to facilitate the comprehension of the design and analysis of programs. In addition, the methodology based on tableaux has helped us to detect in the students difficulties applying the formal techniques to derive correct and efficient imperative programs from specifications.

6 Conclusions

We have presented an educational prototype tool based on semantic tableaux for a specification language on predicate logic. This is the first step towards the development of a practical reasoning tool for formal verification and declarative debugging of algorithms. We have systematically evaluated the proposed method to confirm that a tableaux tool is a good complement to both the class explanations and material, making easier the visualization of proofs in the reasoning needed for the design of correct and efficient imperative programs.

References

1. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
2. van Ditmarsch, H.: Logic software and logic education. Conta-ins a Comprehensive List of Educational Logic Software (2005)
3. Fitting, M.: First-Order Logic and Automated Theorem Proving. Graduate Texts in Computer Science. Springer, Heidelberg (1990)
4. Kaldewaij, A.: Programming: The Derivation of Algorithms. Prentice-Hall International Series in Computer Science (1990)
5. Lancho, B.P., Jorge, E., de la Viuda, A., Sanchez, R.: Software Tools in Logic Education: Some Examples. Logic Journal of the IGPL 15(4), 347–357 (2007)
6. Naish, L.: A Declarative Debugging Scheme. Journal of Functional and Logic Programming 3 (1997)
7. van der Pluijm, E.: TABLEAU: Prototype of an Educational Tool for Teaching Smullyan Style Analytic Tableaux, University of Amsterdam (2007)
8. Shapiro, E.Y.: Algorithmic Program Debugging. MIT Press, Cambridge (1983)