Available online at www.sciencedirect.com

Procedia Computer Science 4 (2011) 1907–1916

International Conference on Computational Science, ICCS 2011

An Innovative Teaching Tool based on Semantic Tableaux for Verification and Debugging of Imperative Programs

Rafael del Vado Vírseda and Fernando Pérez Morente*

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain, Madrid, C.P. 28040

Abstract

While Computational Logic plays an important role in several areas of Computer Science (CS), most educational software developed for teaching logic is not suitable to be used directly in large portions of the CS education domain where the application of logical notions is usually required. In this paper we describe an innovative methodology based on a logic teaching tool on semantic tableaux that has been developed to help students to use logic as a formal proof technique in other advanced topics of CS, such as the verification of algorithms, the algorithmic debugging of programs, and the derivation of algorithms from logical specifications, which are foundations of good development of software. We present the results of the evaluation of this tool by means of several educational experiences during the academic year 2009/2010. From these results we conclude that the use of the tool in current CS teaching can help our students to understand more advanced CS concepts and clarify the formal process involved in the design and analysis of correct and efficient imperative programs.

Keywords: Tools to aid in teaching, Semantic tableaux, Formal verification, Algorithmic debugging

1. Introduction

Computational Logic is a subject that is taught in the first courses of almost all the Computer Science (CS) departments around the world. The syllabus of the course usually includes the syntax and semantics of propositional and first-order logic, as well as some proof systems such as natural deduction, resolution, or semantic tableaux. In some cases, there is also some time devoted to explain basic concepts of logic programming and practical work using a *Prolog* interpreter. However, while Computational Logic plays an important role in several areas of CS, most of the educational software developed for teaching logic ignores their application in a number of subjects of the CS education domain. Many educational tools in this area (e.g., logic inference assistants and proof visualizers) have been developed with different degrees of success, and their utility has been proved by means of several educational experiments and publications. An extensive collection can be found at `http://www.ucalgary.ca/aslcle/logic-courseware`. Unfortunately, although advanced logical notions are always applied in superior courses, existing logic teaching tools are not suitable for use in those subjects.

*Corresponding authors
Email address: `rdelvado@sip.ucm.es,fperezmo@fdi.ucm.es` (Rafael del Vado Vírseda and Fernando Pérez Morente)

doi:10.1016/j.procs.2011.04.208

The aim of this work is to describe an innovative methodology based on a logic teaching tool that uses semantic tableaux to visualize formal proofs of advanced topics in CS, such as the design of correct and efficient algorithms from logical specifications. A *semantic tableau* [1] is a semantic but systematic method of finding a model for a given set of formulas Γ, usually classified as a refutation system because a theorem φ is proved from Γ by getting its negation $\Gamma \Vdash \neg\varphi$. Our major contribution is the development of new tableau methods that provide semantically reach feedback to the students in order to help them to understand the formal reasoning performed in the process of verifying the correctness of algorithms. Moreover, it allows of performing a declarative debugging of programs following a classical idea from Shapiro [2] that proposes replacing computation traces by computation trees with program fragments attached to their nodes in the debugging process. As a novelty, in this work we propose to use semantic tableaux as computation trees to show the students that they can reason about the results of the execution of a program only considering the meaning of the program itself and ignoring complex operational details.

We have tested the tool during the academic year 2009/2010, performing some educational experiments to estimate the benefits to our students from using the tool as a complement to scheduled regular classes. This evaluation has been carried out by means of several tests, some of them managed in an online platform with open access to the students, and the other ones in a CS laboratory with a controlled group. We show the results of these educational experiments and the benefits of using the tool in the teaching of advanced CS concepts involving the formal verification and the algorithmic debugging of imperative programs. We believe that these educational experiences prove that our methodology based on tableaux provides an excellent training for the students in the practical application of advanced concepts of logic to perform different CS tasks.

2. The Logic Teaching Tool

Solving logical exercises is usually done with pen and paper, but educational tools can offer more useful pedagogical possibilities. The role of this 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 [3].

Our logic teaching tool, named `TABLEAUX` (`gpd.sip.ucm.es/WTCS2011`), is a prototype of an educational application based on propositional and first-order semantic tableaux with equality and unification [1] used as a support for the teaching of deductive reasoning at a very elementary university level for Computer Science students. This tool helps our students 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 visualize and produce their own proof trees.

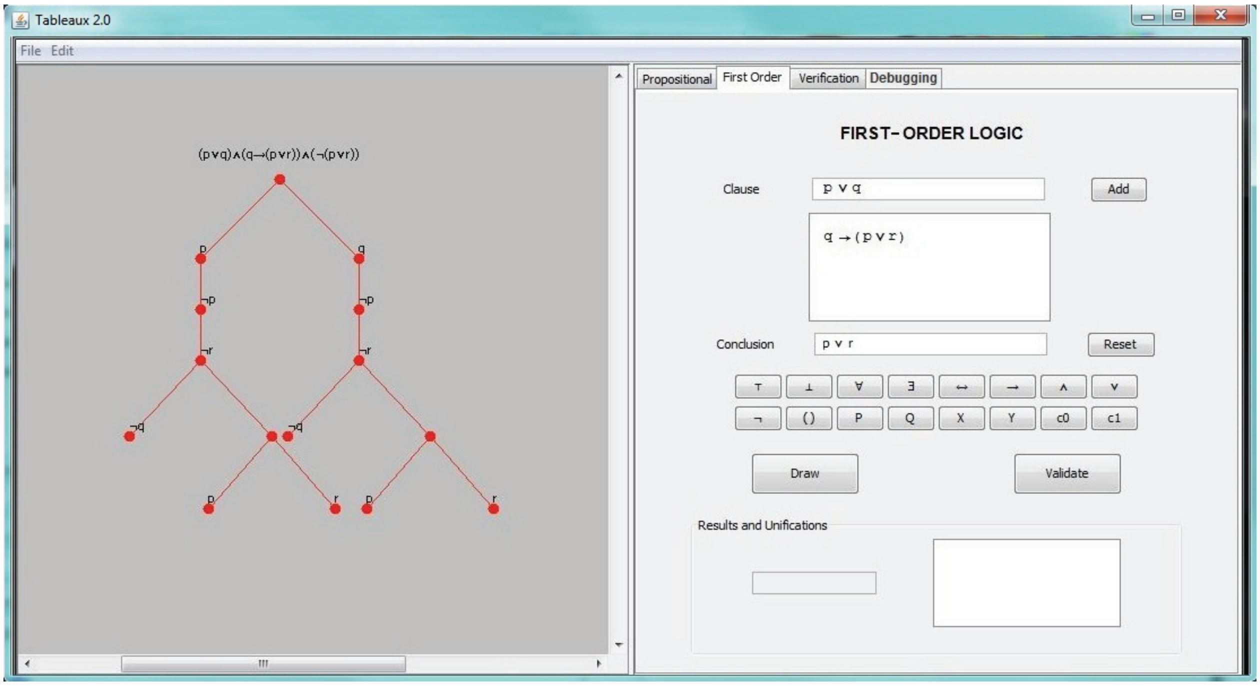
2.1. Tool Usage

The tool consists of two main parts: one that produces propositional and first-order tableaux, and another one based on this tableaux methodology for verification and debugging of algorithms. In both cases, the application possesses a drawing window where the trees will be graphically displayed. The graphical structure of `TABLEAUX` is shown in **Figure 1**. The user interacts with the prover through a graphical interface. We have chosen *Java*, but it is possible to use *Prolog* to write the prover because its declarative character can give us a natural way to write the operations involved in the implementation (see [1] 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 extend, 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 [3, 4, 5]. We have selected the following aspects for a flexible and declarative representation of formulas and tableau rules:



Clause: p ∨ q

q → (p ∨ r)

Conclusion: p ∨ r

Figure 1: The logic teaching tool TABLEAUX for propositional and first-order semantic tableaux.

- **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 **Figure 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* [6], 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 [3, 4, 5]. 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.

- **Checking the tableau for mistakes:** An important functionality of the tool, partly based on the automatic prover, is to check the tableau for errors. Errors can occur in manually created tableaux in the three ways: Syntactic errors in resulting formulas, the wrong output for the correct rule, and applying the wrong rule. When checking the tableau for errors, ideally the application is able to discriminate among these possibilities. To check the tableau, the tool compares every applied rule with the correct and valid rules up to that point, to see if it constitutes a legal action and the results are correct.

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 Dijkstra and others during the 1970s [6]. The tableau proof rules of the algorithm notation used in this paper provides new guidelines for the *verification of algorithms* from specifications (see [6] for more details). We use 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 in algorithms. To be sure of the correctness of an algorithm one has to prove that it meets its *specification* [6]. 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 $\{P\}\ A\ \{Q\}$. Such a triple means that Q holds in any state reached by executing A from an initial state in which P holds. 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 verifies 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 [7] for more details).

As an illustrative example, we consider the formal verification of a simple algorithm *divide* to compute the positive integer (*int*) division between a and b (with quotient c and remainder r), specified as:

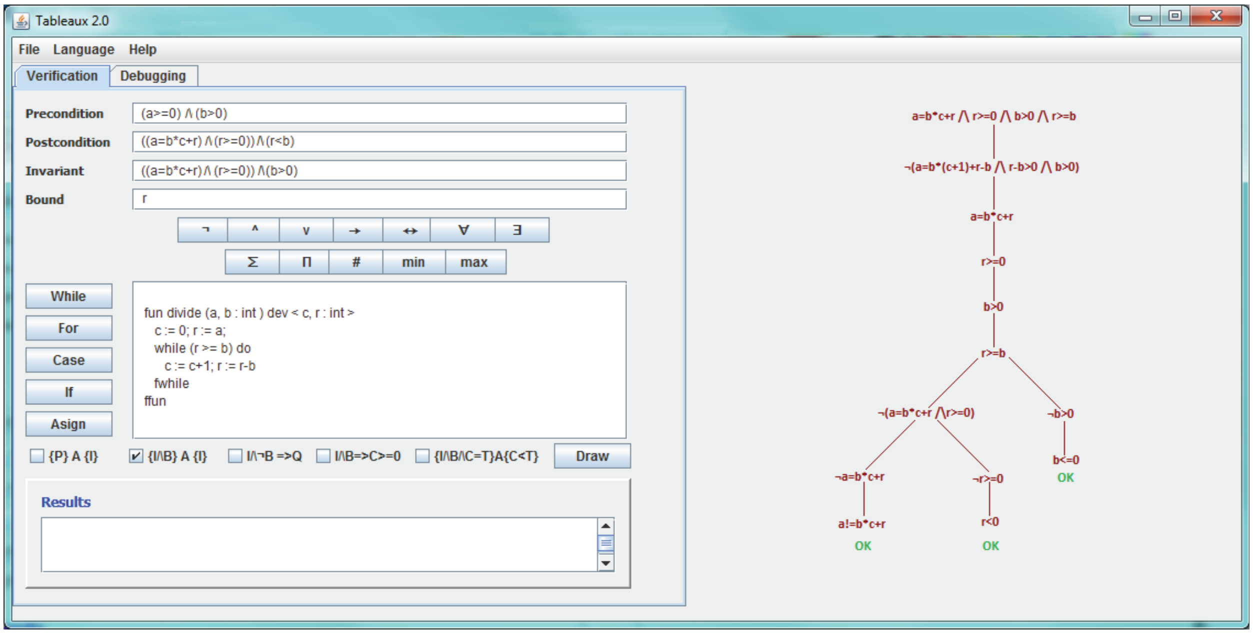
```
{P : $a \geq 0 \wedge b > 0$}
fun divide (a, b : int) dev < c, r : int >
   $c := 0$;  $r := a$;
   {I : $a = b*c+r \wedge r \geq 0 \wedge b > 0$ , C : r}
  while $r \geq b$ do
          $c := c+1$;  $r := r-b$
  fwhile
ffun
{Q : $a = b*c+r \wedge r \geq 0 \wedge r < b$}
```

Following [7], the verification is based on a *loop invariant* I (supplied by the designer 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 [1] for more explanations). We use the notation $R^{e,\ldots}_{x,\ldots}$ to represent the predicate R in which x is replaced by e, etc. For example, we have the following tableau proof (graphically displayed by the tool in **Figure 2**) to verify the preservation of the invariant I in the body of the loop: $\{I \wedge r \geq b\}\ c := c+1; r := r-b\ \{I\} \Leftrightarrow I \wedge r \geq b \Vdash \neg wp\,(c := c+1\,; r := r-b, I) \Leftrightarrow I \wedge r \geq b \Vdash \neg\,(I^{\,c+1,\,r-b}_{\,c,r})$.

$$
\begin{array}{lll}
(1)\ a = b*c+r \wedge r \geq 0 \wedge b > 0 \wedge r \geq b & \{I \wedge r \geq b\} & \\
(2)\ a = b*c+r & (\alpha,1) & \\
(3)\ r \geq 0 & (\alpha,1) & \\
(4)\ b > 0 & (\alpha,1) & \\
(5)\ r \geq b & (\alpha,1) & \\
(6)\ \neg(a = b*(c+1)+r-b \wedge r-b \geq 0 \wedge b > 0) & \{\neg(I^{\,c+1,\,r-b}_{\,c,r})\} & \\
\hline
& & (\beta,6) \\
(7)\ a \neq b*c+r & (8)\ r < b & (9)\ b \leq 0 \\
\checkmark\ (2,7) & \checkmark\ (5,8) & \checkmark\ (4,9)
\end{array}
$$



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}

Figure 2: The logic teaching tool TABLEAUX for the formal verification of algorithms by mean of semantic tableaux.

Finding invariants is an essential part of this educational process. 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 predicate $I'\ :\ a = b * c + r$ as the loop invariant. Then, when they apply the tool to verify the algorithm, they obtain an *open* semantic tableau (indicated by $\times$) for $I' \wedge r < b \Rightarrow Q$:

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

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

We need to insert $\boxed{r \geq 0}$ in I' to close this 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 $\{I'' \wedge r \geq b \wedge C = T\}\ c := c+1; r := r-b\ \{C < T\}$:

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

$\times \Rightarrow$ We need to insert $\boxed{b > 0}$ in I'' to close this 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 there is 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* [8] 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 [2], algorithmic debugging proposes to replace computation traces by *computation trees* with program fragments attached to the nodes. As a novelty, in this work we propose using semantic tableaux as computation trees. As an example, we alter the code of the previous algorithm with two kinds of mistakes:

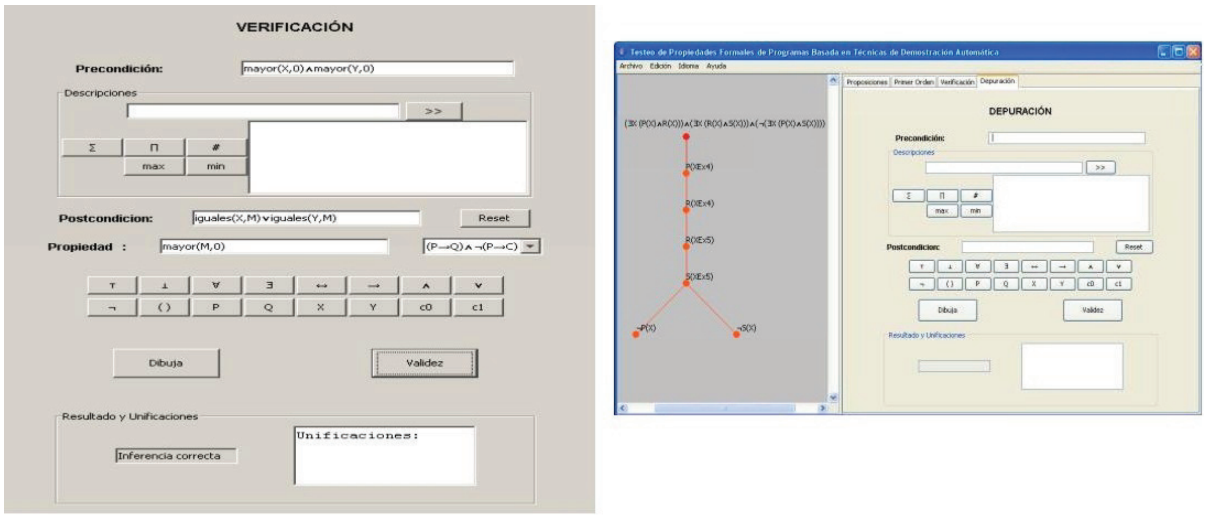
```
{$P : a \geq 0 \wedge b > 0$}
 fun $divide\ (a, b : int)$ dev $< c, r : int >$
   $c := 0$; $\boxed{\texttt{r := 0}}$;                              <-- wrong code!
  {$I : a = b*c+r \wedge r \geq 0 \wedge b > 0\ ,\ C : r$}
   while $r \geq b$ do $\boxed{\texttt{r := r - b}}$ fwhile           <-- 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 again execute the tool. Initially, 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 the erroneous parts of the code used in open branches:

$$\begin{array}{lll}
(1)\ a \geq 0 \wedge b > 0 & & \{P\} \\
(2)\ a \geq 0 & & (\alpha, 1) \\
(3)\ b > 0 & & (\alpha, 1) \\
(4)\ a \neq b*c+r \vee r < 0 \vee b \leq 0 & & \{\neg I\} \\
\hline
 & & (\beta, 4) \\
(5)\ a \neq b*c+r & (6)\ r < 0 & (7)\ b \leq 0 \\
\quad \Big\| \boxed{\texttt{c := 0}} & \quad \Big\| \boxed{\texttt{r := 0 (or a)}} & \checkmark\ (3,7) \\
(5)\ a \neq r & (6)\ 0\ (or\ a) < 0 & \\
\quad \Big\| \boxed{\texttt{r := 0}} & \checkmark\ or\ (\checkmark (2,6)) & \\
(5)\ a \neq 0 & & \\
\quad \Downarrow & & \\
\quad \times \Rightarrow \text{We must replace r := 0 by } \boxed{\texttt{r := a}} \text{ to close} & &
\end{array}$$

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

$$\begin{array}{ll}
(1)\ a = b*c+r \wedge r \geq 0 \wedge b > 0 \wedge r \geq b & \{I \wedge r \geq b\} \\
(2)\ a = b*c+r & (\alpha, 1)
\end{array}$$


VERIFICACIÓN
Precondición:
Descripciones
>>
Σ
Π
#
max
min
Postcondicion:
Reset
Propiedad :
⊤ ⊥ ∀ ∃ ↔ → ∧ ∨
¬ () P Q X Y c0 c1
Dibuja
Validez
Resultado y Unificaciones
Testeo de Propiedades Formales de Programas Basada en Técnicas de Demostración Automática
Archivo Edición Idioma Ayuda
Proposiciones Primer Orden Verificación Depuración
DEPURACIÓN
P(XEx4)
R(XEx4)
R(XEx5)
S(XEx5)
¬P(X)
¬S(X)

Precondición: mayor(X,0)∧mayor(Y,0)

Postcondicion: iguales(X,M)∨iguales(Y,M)

Propiedad : mayor(M,0) (P→Q)∧¬(P→C)

Inferencia correcta

Unificaciones:

(∃X (P(X)∧R(X)))∧(∃X (R(X)∧S(X)))∧(¬(∃X (P(X)∧S(X))))

Figure 3: The logic teaching tool TABLEAUX for verification and debugging of algorithms.

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

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

To close the open branch, we deduce 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 again apply the tool, then no more errors can be found and the five tableaux remain closed. The debugging session is finished.

This algorithmic debugging methodology can be also applied to explain the *derivation of simple algorithms* [7] in the classroom. For example, the semantic tableau for $I \wedge \neg(???) \Rightarrow Q$ allows to our students to derive the repetition condition of the loop:

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

$$
\begin{array}{lll}
(7)\ \ a \neq b*c+r & (8)\ \ r<0 & (9)\ r \geq b \\
 & & \Downarrow \\
\checkmark\ (2,7) & \checkmark\ (3,8) & \times
\end{array}
$$

$$\Downarrow$$

We have derived $\boxed{??? : r \geq b}$

Analogously, we can display a semantic tableau for $\{P\}$??? $\{I\}$ to derive the initialization code of the loop:

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

$$\begin{tikzcd}
(5)\ a \neq b*c+r \arrow[dd, dash] & & (6)\ r < 0 \arrow[d, Rightarrow] & (7)\ b \leq 0 \\
{} & \boxed{\text{r := a}} \arrow[l, Leftarrow] & \times \arrow[l, Leftarrow] \arrow[d, Rightarrow] & \checkmark\ (3,7) \\
(5)\ a \neq b*c+a \arrow[d, Rightarrow] & & (6)\ a < 0 \quad \checkmark\ (2,6) & \\
\times \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \boxed{\text{c := 0}} & & \\
(5)\ a \neq a & & &
\end{tikzcd}$$

$\checkmark \Rightarrow$ We have derived $\boxed{\text{??? : c := 0 ; r := a}}$

During the academic year 2009/2010, we applied the tool to design more illustrative classes of problems from [7]. All these examples provide an excellent training in the reasoning needed for deriving correct programs, as we will see in the next section.

5. Experiences and results

The prototype of the educational tool `TABLEAUX` is available for the students of the topics *Computational Logic* and *Design of Algorithms* in the Computer Science and Software Engineering Faculty of our University through an online educational platform called *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 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`

	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	12.77	3.71	4.81	2.10	1.22	1.73

Figure 4: Means and standard deviations (σ) of the controlled experience.

for the verification and debugging of simple searching and sorting algorithms [7]. We have chosen two groups of students answering the same questions: approximately half of the students work only with 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*

5.2.1. *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 test 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 resource and have used it to complement the rest of the available material.

5.2.2. *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 [1, 7]. 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 **Figure 4** we provide the means and the standard deviations of correct, errors, and *don't knows* answers. First, we observe that students using `TABLEAUX` answer 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 displayed by tool, while the *slides/book* group have to deduce it directly from the code. All the students who used `TABLEAUX` 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 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 innovative educational methodology based on semantic tableaux for a specification language on predicate logic. This is the first step towards the development of a practical teaching technology for formal verification and declarative debugging of algorithms. We have systematically evaluated the proposed methodology 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. We look forward to making good use of what we have learned from this evaluation to improve the tool's usefulness in Computer Science education.

Acknowledgments

This work has been partially supported by the Spanish projects TIN2008-06622-C03-01 (FAST-STAMP), S2009TIC-1465 (PROMETIDOS-CM), and UCM-BSCH-GR58/08-910502 (GPD-UCM).

References

[1] M. Fitting, First-Order Logic and Automated Theorem Proving, Springer, 1990.
[2] E. Shapiro, Algorithmic Program Debugging, MIT Press, Cambridge, MA, USA, 1983.
[3] H. van Ditmarsch, Logic software and logic education, 2005.
[4] B. Lancho, E. Jorge, A. de la Viuda, R. Sanchez, Software tools in logic education: Some examples, Logic Journal of the IGPL 15(4) (2007) 347–357.
[5] E. van der Pluijm, Tableau: Prototype of an educational tool for teaching smullyan style analytic tableaux, Tech. rep., University of Amsterdam (2007).
[6] E. Dijkstra, A Discipline of Programming, Prentice Hall, 1976.
[7] A. Kaldewaij, Programming: The Derivation of Algorithms, Prentice-Hall, 1990.
[8] L. Naish, A declarative debugging scheme, Journal of Functional and Logic Programming, vol. 3, 1997.