Universidad Complutense Madrid

Facultad de Informática

PROYECTO DE SISTEMAS INFORMÁTICOS

CURSO 2010 / 2011

TABLEAUX VERIFICATION TOOL

Autores: Eduardo Berbis González

Saúl de León Guerrero

Eva Pilar Orna Ruiz

Director: Rafael del Vado Vírseda

AUTORIZACIÓN

Los ponentes Eduardo Berbis González, con DNI 02909467J , Saúl de León Guerrero, con DNI 50754645Q , y Eva Pilar Orna Ruiz, con DNI 50899536F, autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos tanto la propia memoria, como el código, la documentación y el prototipo desarrollado.

Eduardo Berbis González Saúl de León Guerrero Eva Pilar Orna Ruiz

En Madrid, a 1 de Julio de 2011

AGRADECIMIENTOS

- A Pablo Pizarro Moleón, por la calidad y el diseño del logo del sistema.
- A Federico Mon Trottu, por el testeo del sistema en Linux y por la documentación y guiado al crear un instalador en dicho sistema operativo.
- A Ignacio Navarro O'Connell, por el testeo del sistema en Windows.

ÍNDICE GENERAL

RESUMEN

Mientras la lógica juega un papel muy importante en varias áreas de la ciencia informática, la mayoría del software educativo desarrollado para la enseñanza lógica ignora su aplicación en una parte más amplia del dominio de la enseñanza de la ciencia informática. En este trabajo, describimos una novedosa metodología cimentada en una herramienta de enseñanza lógica. Dicha enseñanza lógica está basada en tableaux semánticos para presentar a los estudiantes una nueva aplicación de la lógica como técnica de prueba formal en otros ámbitos de la ciencia informática, tales como la verificación formal y la depuración declarativa de programas imperativos, las cuales representan la base de un buen desarrollo del software.

PALABRAS CLAVE

Verificación formal, depuración algorítmica, tableaux, ciencia informática, programas imperativos.

ABSTRACT

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

KEY WORDS

Formal verification, declarative debugging, tableaux, computer science, imperative programs

CAPÍTULO 1: INTRODUCCIÓN

1.1 - ESTADO DEL ARTE Y MOTIVACIÓN

Las universidades de ciencias informáticas suelen impartir el primer año a sus alumnos un curso introductorio sobre lógica matemática. El plan de estudios del curso suele incluir sintaxis y semántica proposicional y de predicados lógicos, así como algunos sistemas de prueba, tales como la deducción natural, resolución y tableaux semánticos. En algunos casos, hay también algunas clases dedicadas a explicar los conceptos básicos de la programación lógica y clases prácticas usando un intérprete de Prolog.

Los estudiantes se dan cuenta de la dificultad que conlleva el aprendizaje de estos contenidos. Para proporcionar soporte al aprendizaje de estos estudiantes, las herramientas de visualización son siempre de ayuda. En las últimas dos décadas se han desarrollado muchas herramientas de enseñanza lógica. La mayoría de estas herramientas se preocupan de la construcción de pruebas de lógica formal usando tableaux semánticos. Un tableaux semántico es un método semántico pero sistemático de encontrar un modelo de un conjunto de fórmulas Γ dadas. Un tableaux semántico es un sistema para refutar que un teorema φ es demostrando desde Γ mediante su negación $\Gamma \Vdash \neg \varphi$.

Mientras la lógica juega un papel muy importante en muchas áreas de la informática, la mayoría del software didáctico desarrollado para la enseñanza de la lógica ignora la aplicación de la lógica en otros temas de la informática. Como estudiantes que somos, creemos que se podrían obtener mayores beneficios de las técnicas aprendidas con

estas herramientas, gracias a la posibilidad de aplicarlas en numerosos contextos en cursos sucesivos. Por lo tanto creemos que existe la necesidad de un prototipo de herramienta que permita experimentar en la enseñanza lógica en más campos de la enseñanza en informática, donde en lenguaje y la implementación deberían ser suficientemente accesibles y populares para asegurar su uso en un futuro y que permanecerán disponibles en otros cursos. Esto es lo que nos motivó a desarrollar este proyecto.

1.2 - OBJETIVOS DEL TRABAJO

El objetivo de este trabajo es describir una metodología innovadora basada en una herramienta de enseñanza lógica en tableaux semánticos, llamada TVT, para preparar a los alumnos para usar la lógica como una herramienta de verificación formal en otras áreas de la informática, poniendo un especial énfasis en el diseño de algoritmos e ingeniería del software. Un buen diseño del algoritmo es crucial para el desarrollo de todos los sistemas software. Por esta razón , la habilidad de crear y entender verificaciones formales es esencial para un desarrollo correcto de los programas.

La mayor contribución de este proyecto es el desarrollo de un nuevo método de tableaux que da información que semánticamente es muy rica a los alumnos para la verificación formal y la depuración de los programas algorítmicos. En este sentido, creemos que TVT será una buena herramienta para los estudiantes, cuyas habilidades lógicas van más allá de las nociones básicas que la interacción a nivel de usuario con otras herramientas se pueden desarrollar. Por ejemplo , nuestra herramienta es usada para metodologías basadas en la lógica, tales como derivación de programas, razonar a partir de especificaciones y asignaciones, invariantes de bucles, funciones de cota, etc. Este proyecto incluye temas de las áreas cuyas habilidades y conceptos son esenciales para la práctica independiente de la programación de la base paradigma, así como el análisis y diseño de algoritmos correctos y eficientes.

1.3 - METODOLOGÍA DEL TRABAJO

Con la intención de tener un ritmo de trabajo asequible, sistemático, bien organizado, transparente y fácilmente reutilizable, se optó por una metodología basada en distintos módulos o paquetes.

Cada uno de ellos fue perfilado y estructurado por todos los integrantes del grupo, acordando horizontalmente cuál debía de ser el espacio de estados tanto de la información de entrada como de la de salida. Así pues, se consiguió una rápida y cristalina comunicación entre los diferentes módulos, creando ciertas fronteras para lo esperado en la entrada y lo exigido en la salida. El propio desarrollo de cada módulo fue asignado individualmente a un miembro del grupo de trabajo, estando dicha asignación sometida a consenso.

Como se anticipó anteriormente, para abordar la consecución de dicho sistema software, éste se dividió en diferentes módulos o paquetes inicialmente, y en relación directa con los objetivos del mismo. Cabe destacar que durante su elaboración, se otorgó cierta flexibilidad a la naturaleza inicial de dichos módulos (dando un margen de maniobra para posibles imprevistos).

Para alcanzar la consumación de los distintos paquetes, se siguieron las siguientes pautas:

1. Análisis de la naturaleza del módulo: *¿de dónde vengo? ¿A dónde voy?* En esta primera fase, se estudia la necesidad del módulo, delimitando lo que el propio módulo espera recibir como entrada y lo que se espera de él en su salida (creación de fronteras).
2. Estudio de la estructura de datos: *¿qué necesito?* En esta segunda fase, se analizan los requisitos del proceso modular, descubriendo sus necesidades y particularidades. De esta manera, se elige una estructura de datos que inequívocamente cubra dichos requisitos (diseño del módulo y de su comportamiento).

3. Implementación del módulo: *¿cómo consigo mis objetivos?* En esta fase, se confeccionan los algoritmos necesarios para alcanzar los objetivos del módulo (elaboración del código a compilar).
4. Verificación: *¿puedo asegurar mis objetivos?* En esta fase, se aprueba la corrección del módulo si resuelve fielmente las necesidades por las cuales su creación fue justificada (logro de objetivos).
5. Documentación: *¿cómo comparto mi éxito?* En esta última fase, se elabora una memoria del módulo, haciendo uso de manuales, diagramas de clases, diagramas de flujo y javadoc, entre otras (necesario para posibles correcciones, reusabilidad, mantenimiento futuro y ampliaciones del propio módulo).

1.4 - ESTRUCTURA DEL TRABAJO

En esta sección, se describe a grandes rasgos la estructura del presente documento. Entre sus páginas se pueden hallar 6 capítulos principales en los cuales se detalla toda la información asociada al desarrollo del proyecto.

Al final del documento se pueden encontrar unos anexos, una bibliografía y un glosario que complementan dichos capítulos, cuyo contenido es el siguiente:

1. Introducción: expone una breve introducción del proyecto realizado, compartiendo la motivación del mismo, así como la metodología y la estructura de trabajo empleadas.
2. Verificación y depuración basadas en tableaux semánticos: detalla el motor sintáctico del proyecto dando las bases y el origen del mismo, e ilustrando con ejemplos guiados la verificación y depuración basadas en tableaux semánticos de algoritmos escritos bajo su sintaxis.
3. La herramienta TVT: muestra los diagramas de clases de todos los módulos que conforman el proyecto (ilustrando en determinados casos su comportamiento con diagramas de flujo), e informa de las tecnologías y plataformas utilizadas para la culminación del mismo.
4. Verificación y depuración de algoritmos con TVT: revela cómo la herramienta TVT verifica y depura algoritmos a través de dos ejemplos guiados paso a paso.
5. Manual de usuario: instruye en el manejo de la herramienta, indicando paso por paso y punto por punto todas las funcionalidades que TVT pone a disposición del usuario, incluyendo también un ejemplo guiado.

6. Conclusiones y trabajo futuro: redacta las conclusiones del sistema y sus principales aportaciones, así como expone un posible trabajo futuro con potenciales extensiones de la herramienta.

CAPÍTULO 2: VERIFICACIÓN FORMAL Y DEPURACIÓN ALGORÍTMICA DE PROGRAMAS IMPERATIVOS BASADA EN TABLEAUX SEMÁNTICOS

2.1 - ESPECIFICACIÓN Y VERIFICACIÓN FORMAL DE ALGORITMOS BASADA EN TABLEAUX SEMÁNTICOS

La principal novedad que proponemos en este proyecto de Sistemas Informáticos es la de utilizar el método de los tableaux semánticos para ayudar a los estudiantes de Informática a especificar formalmante sus propios programas imperativos mediante predicados lógicos, para a continuación, poder probar su corrección a partir de esta especificación y utilizarla en su diseño y depuración.

Para alcanzar este propósito, vamos a utilizar la aproximación clásica inicialmente desarrollada por Dijkstra durante los años 70 en (Dijkstra, 1976) para la verificación formal de algoritmos. En concreto, vamos a usar el lenguaje de comandos con guardas de (Dijkstra, 1976) para escribir nuestros algoritmos.

Un algoritmo *A* se representa mediante una función de la forma:

fun A ffun

la cual puede contener variables (*x*, *y*, *z*, etc.), expresiones numéricas (*e*), y expresiones booleanas (*B*). Para representar su código, usaremos la instrucción nula *skip*, instrucciones de asignación *x := e*, composición secuencial (*S; S'*), instrucciones condicionales (*if B then S else S' fif*), e instrucciones iterativas en forma de un bucle *while* con la sintaxis *while B do S fwhile*. El lenguaje elegido es bastante modesto, pero

suficientemente rico para representar algoritmos secuenciales de un modo conciso y elegante. Es posible escribir algoritmos más complicados aplicando técnicas de refinamiento sucesivo y diseño descendente sobre el código concreto del algoritmo.

Resulta claro que ni el análisis de la traza de un cómputo ni el testeo en base a un cierto banco de pruebas puede garantizar la ausencia de errores en el código de los algoritmos. Para estar completamente seguros de la corrección de un algoritmo, es necesario demostrar formalmente que su código satisface su *especificación* (Dijkstra, 1976).

La especificación de un algoritmo A consiste en la definición de un *estado* (un conjunto de variables), una *precondición* P y una *postcondición* Q (ambos predicados expresando propiedades sobre los valores de las variables del algoritmo), denotado como $\{P\}\, A\, \{Q\}$. Dicha terna significa que Q se verifica en cualquier estado alcanzado por la ejecución de A a partir de un estado inicial en el cual P se verifica.

Un algoritmo junto con su especificación puede ser visto como un teorema matemático. Dicho teorema expresa que el algoritmo satisface la especificación. Así, al igual que los teoremas requieren demostraciones matemáticas, los algoritmos también los necesitan para probar su corrección.

La metodología que se aplica en este proyecto de Sistemas Informáticos permite verificar algoritmos a partir de su especificación de una manera totalmente constructiva a través de la obtención de un tableau semántico de la forma $P \Vdash \neg\, wp(A, Q)$, donde $wp(A, Q)$ es la denominada precondición más débil del algoritmo A con respecto a la postcondición Q, la cual se define como el predicado menos restrictivo que asegura que si un estado lo satisface entonces después de ejecutar

el algoritmo A el predicado Q se verifica (el lector puede consultar (Kaldewaij, 1990) para obtener más detalles al respecto de todas estas definiciones).

2.2 - UN EJEMPLO DE VERIFICACIÓN FORMAL BASADA EN TABLEAUX SEMÁNTICOS

En esta sección vamos a considerar un ejemplo concreto de algoritmo que nos va a permitir ilustrar la metodología propuesta en este proyecto de Sistemas Informáticos para la verificación formal de programas imperativos basada en tableaux semánticos.

Vamos a considerar la verificación formal de un algoritmo muy sencillo *divide* que nos permita computar la división entera positiva (*int*) entre dos números enteros positivos a y b (siendo b distinto de cero, y siendo c el cociente y r el resto de la división). El código y la especificación de este algoritmo es el siguiente:

$\{P : a \geq 0 \wedge b > 0\}$
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
$\{Q : a = b*c + r \wedge r \geq 0 \wedge r < b\}$

De acuerdo con (Kaldewaij, 1990), la verificación formal de este algoritmo iterativo se basa en la formulación de un predicado *invariante* I para el bucle (proporcionado por el diseñador del algoritmo o por alguna herramienta específica de análisis de código), una *función de cota* C (para demostrar la terminación), y las siguientes cinco *pruebas formales*:

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

· $\{I \wedge r \geq b \wedge C = T\}\, c := c + 1;\ r := r - b\, \{C < T\}$

La metodología que hemos seguido en el presente trabajo para implementar la herramienta TVT que se va a presentar en los siguientes capítulos, nos permite representar cada una de estas pruebas formales mediante un tableau semántico cerrado.

Asumimos que el lector está familiarizado con las reglas clásicas de construcción de tableaux semánticos (α and β), de igualdad (=), y reglas de cierre (el lector puede consultar (Fitting, 1990) para obtener más explicaciones sobre todas estas nociones). Vamos a usar también la notación $R^{e,\ldots}_{x,\ldots}$ para representar un predicado R en el cual la variable x ha sido reemplazada por la expresión e, etc.

Por ejemplo, podemos considerar la siguiente prueba formal en forma de tableau semántico para verificar la preservación del invariante I por el cuerpo del bucle:

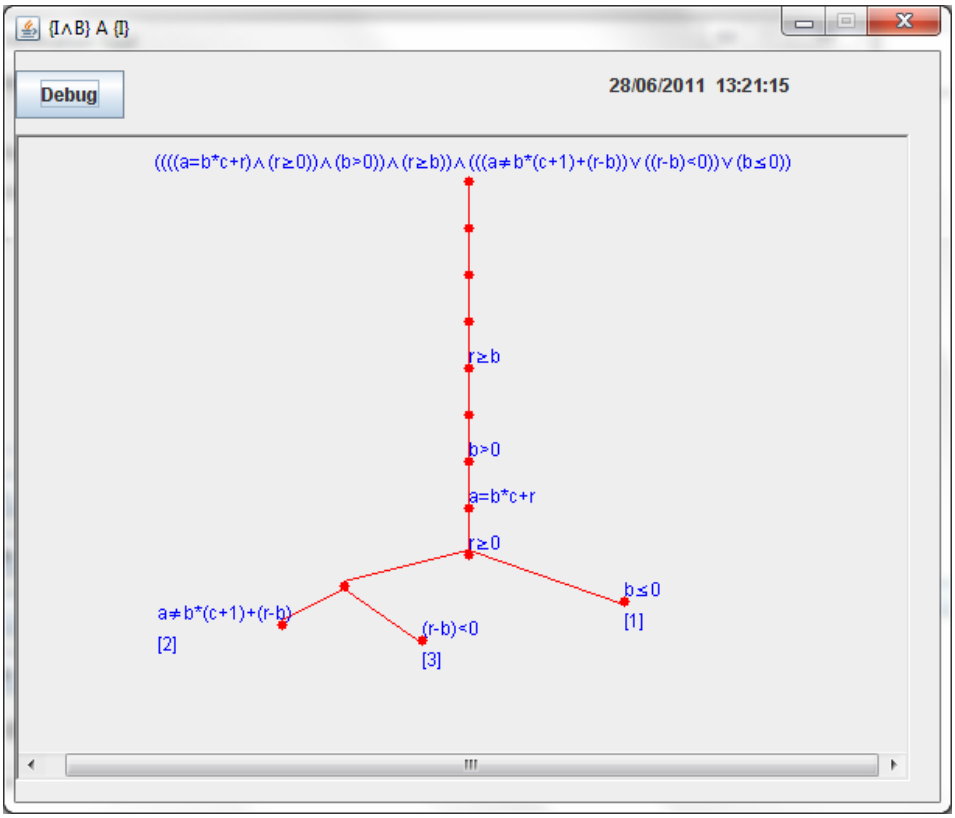
$$\{I \wedge r \geq b\}\, c := c + 1;\ r := r - b\, \{I\} \iff$$

$$I \wedge r \geq b \Vdash \neg wp(c := c + 1;\ r := r - b, I) \iff$$

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

(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)	$\neg(a = b*(c+1)+r-b \wedge r-b \geq 0 \wedge b > 0)$	$\{\neg(I^{c+1,r-b}_{c,r})\}$
		$(\beta, 6)$
(7)	$a \neq b*c+r$	✓ (2,7)
(8)	$r < b$	✓ (5,8)
(9)	$b \leq 0$	✓ (4,9)

La representación gráfica que la herramienta TVT hace de este tableau semántico es la siguiente (como se explicará en los capítulos siguientes de esta memoria):



La búsqueda de invariantes es una de las partes esenciales de todo este proceso educacional. Podemos usar la metodología basada en tableaux semánticos propuesta en este trabajo así como la herramienta TVT para guiar a nuestros estudiantes en la obtención de invariantes correctos a partir de sus especificaciones.

Por ejemplo, si solo proporcionamos a un estudiante la postcondición Q, entonces usualmente suelen inferir como invariante el siguiente predicado (incompleto):

$$I' : a = b * c + r$$

Si ahora el alumno trata de aplicar la herramienta para verificar el algoritmo, entonces se obtiene un tableau semántico *abierto* (indicado mediante ×) para verificar $I' \wedge r < b \Rightarrow Q$:

$$
\begin{array}{lll}
(1)\ a = b*c+r \wedge r < b & & \{I' \wedge r < b\} \\
(2)\ a = b*c+r & & (\alpha,1) \\
(3)\ r < b & & (\alpha,1) \\
(4)\ \neg(a = b*c+r \wedge r \geq 0 \wedge r < b) & & \{\neg Q\} \\
\hline
(5)\ a \neq b*c+r & (6)\ r < 0 & (7)\ r \geq b \\
 & \Downarrow & \\
\checkmark\ (2,5) & \times & \checkmark\ (3,7) \\
 & \Downarrow &
\end{array}
\quad (\beta,4)
$$

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

A partir de la rama abierta (6), nuestros estudiantes aprenden que deben completar el invariante con

$$I'': a = b * c + r \wedge r \geq 0$$

Sin embargo, todavía existe un tableau que permanece abierto. En concreto, el tableau semántico que correspondería a la siguiente prueba formal:

$$\{I'' \wedge r \geq b \wedge C = T\}\ c := c + 1;\ r := r - b\ \{C < T\}$$

$$
\begin{array}{ll}
(1)\ a = b*c+r \wedge r \geq 0 \wedge r \geq b \wedge r = T & \{I'' \wedge r \geq b \wedge C = T\} \\
(2)\ a = b*c+r & (\alpha,1) \\
(3)\ r \geq 0 & (\alpha,1) \\
(4)\ r \geq b & (\alpha,1) \\
(5)\ r = T & (\alpha,1) \\
(6)\ r - b \geq T & \{\neg(C < T)^{c+1,r-b}_{c,r}\} \\
(7)\ b \leq 0 & (=,5,6) \\
\Downarrow & \\
\times \Rightarrow \text{We need to insert } \boxed{b > 0} \text{ in } I'' \text{ to close this tableau} &
\end{array}
$$

Por último, los alumnos aprenden que deben insertar $b > 0$ en el predicado I'' para poder completar el invariante I del bucle. Si ahora aplican la herramienta de nuevo, entonces todos los tableaux permanecen cerrados y la sesión de verificación formal concluye con éxito.

Nuestra metodología también puede aplicarse en la verificación formal de *algoritmos recursivos*. Por ejemplo, podemos considerar ahora la verificación formal de la siguiente versión recursiva del algoritmo *divide*:

$\{P : a \geq 0 \wedge b > 0\}$
fun $divide\,(a, b : int)$ dev $< c, r : int >$
  if $a < b$ then $c := 0;\ r := a;$
  else $< c, r > := divide\,(a - b, b);$
       $c := c + 1$
  fif
ffun
$\{Q : a = b * c + r \wedge r \geq 0 \wedge r < b\}$

De acuerdo con (Kaldewaij, 1990), la verificación formal de programas recursivos se basa ahora en las siguientes seis pruebas formales:

- $P \Rightarrow a < b \vee a \geq b$
- $P \wedge a < b \Rightarrow Q_{c,r}^{0,a}$
- $P \wedge a \geq b \Rightarrow P_{a,b}^{a-b,b}$
- $P \wedge a \geq b \wedge Q_{a,b}^{a-b,b} \Rightarrow Q_{c,r}^{c+1,r}$
- $P \Rightarrow C \geq 0$

· $P \wedge a \geq b \Rightarrow C_{a,b}^{a-b,b} < C$

Por ejemplo, tenemos el siguiente tableau semántico para verificar la corrección de la *llamada recursiva* en el código del algoritmo:

(1) $a \geq 0 \wedge b > 0 \wedge a \geq b \wedge a - b = b * c + r \wedge r \geq 0 \wedge r < b$	$\{P \wedge B_{nt} \wedge Q_{a,b}^{a-b,b})\}$
(2) $a \geq 0$	$(\alpha, 1)$
(3) $b > 0$	$(\alpha, 1)$
(4) $a \geq b$	$(\alpha, 1)$
(5) $a = b * (c+1) + r$	$(\alpha, 1)$
(6) $r \geq 0$	$(\alpha, 1)$
(7) $r < b$	$(\alpha, 1)$
(8) $\neg(a = b * (c+1) + r \wedge r \geq 0 \wedge r < b)$	$\{\neg(Q_{c,r}^{c+1,r})\}$

$(\beta, 8)$

(9) $a \neq b * (c+1) + r$	(10) $r < 0$	(11) $r \geq b$
✓ (5,9)	✓ (6,10)	✓ (7,11)

2.3 - DEPURACIÓN ALGORÍTMICA DE PROGRAMAS BASADA EN TABLEAUX SEMÁNTICOS

La *depuración de programas* es una de las partes esenciales del ciclo de desarrollo del software y una necesidad práctica para ayudar a nuestros estudiantes a entender por qué sus programas no funcionan como querían. En esta sección vamos a aplicar de manera novedosa las ideas de la *depuración algorítmica* (Naish, 1997) como una alternativa a otros enfoques más convencionales utilizados en la depuración de programas imperativos.

La mayor ventaja de la depuración algorítmica, comparada con la depuración convencial de programas, es que permite a nuestros estudiantes trabajar con un mayor nivel de abstracción. En particular, hemos aplicado satisfactoriamente la herramienta basada en tableaux semánticos implementada en este proyecto de Sistemas Informáticos a la depuración algorítmica de programas imperativos sencillos, con el fin de demostrar cómo es posible razonar sobre estos programas sin necesidad de utilizar complicadas argumentaciones operacionales basadas en trazas.

Siguiendo una idea original de Shapiro (Shapiro, 1983), la depuración algorítmica propone reemplazar trazas de cómputo por *árboles de cómputo* con fragmentos de programa asociados a sus nodos. Como novedad en este proyecto de Sistemas Informáticos, proponemos usar tableaux semánticos como árboles de cómputo.

Para ilustrar todas estas ideas, vamos a modificar el código del algoritmo de división entera que hemos verificado en la sección anterior con dos tipos de errores:

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

Si tratamos de verificar formalmente el código erróneo de este algoritmo, podemos utilizar de nuevo la metodología expuesta en la sección anterior basada en el uso de tableaux semánticos. En primer lugar, vamos a construir un tableau abierto $P \Vdash \neg I$ para poder depurar las instrucciones iniciales del programa erróneo, es decir, para poder demostrar formalmente el paso de verificación $\{P\}\, c := 0; r := 0\, \{I\}$. En esta ocasión vamos a construir el tableau $P \Vdash \neg I$ en lugar del tableau $P \Vdash \neg(I^{0,0}_{c,r})$. Sin embargo, la precondición más débil $I^{0,0}_{c,r}$ se va a construir a partir de los pasos del tableau (5) y (6), paso por paso, con el fin de poder identificar las partes erróneas del código que hemos usado en cada una de las ramas abiertas:

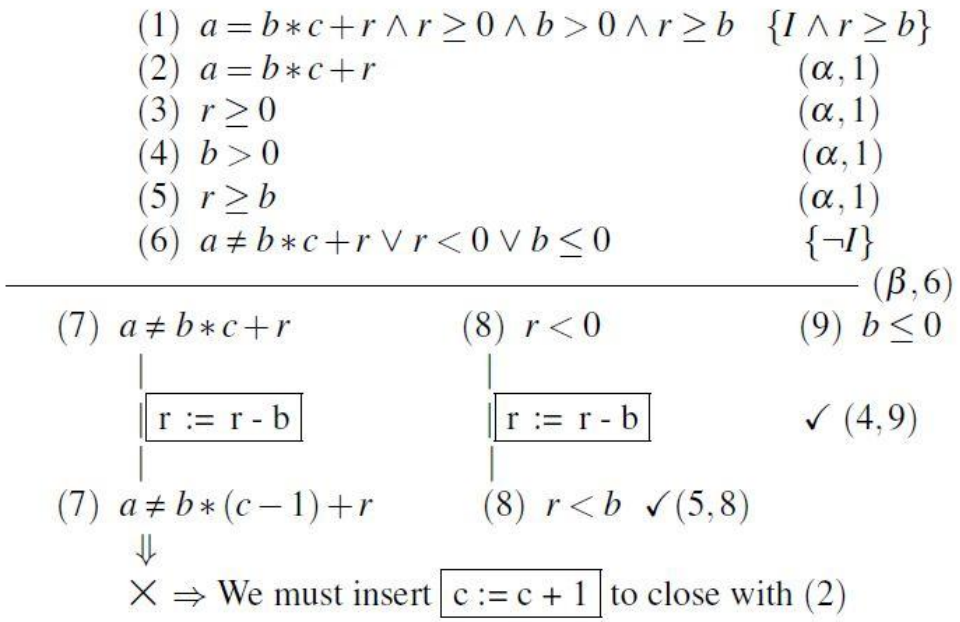
(1) $a \geq 0 \wedge b > 0$ $\{P\}$
(2) $a \geq 0$ $(\alpha, 1)$
(3) $b > 0$ $(\alpha, 1)$
(4) $a \neq b * c + r \vee r < 0 \vee b \leq 0$ $\{\neg I\}$
$(\beta, 4)$

(5) $a \neq b * c + r$ | c := 0 | (5) $a \neq r$ | r := 0 | (5) $a \neq 0$ ⇓ × ⇒ We must replace r := 0 by r := a to close

(6) $r < 0$ | r := 0 (or a) | (6) $0\ (or\ a) < 0$ | ✓ *or* (✓(2,6))

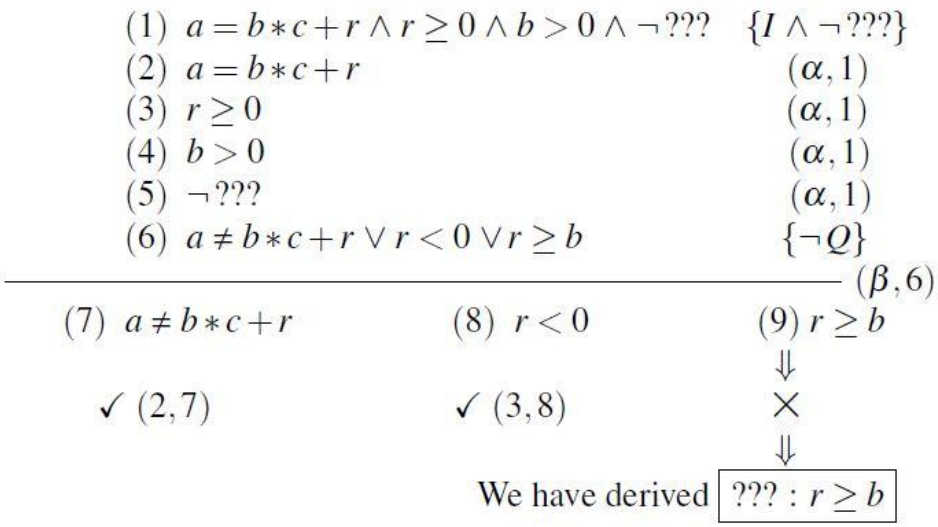
(7) $b \leq 0$ | ✓ (3,7)

Tras esta corrección, obtenemos ya un tableau cerrado. Sin embargo, aun necesitamos comprobar el resto de los pasos de verificación. Por ejemplo, podemos tratar de realizar ahora la depuración algorítmica de $\{I \wedge r \geq b\}\, r := r - b\, \{I\}$ estudiando su correspondiente tableau semántico:

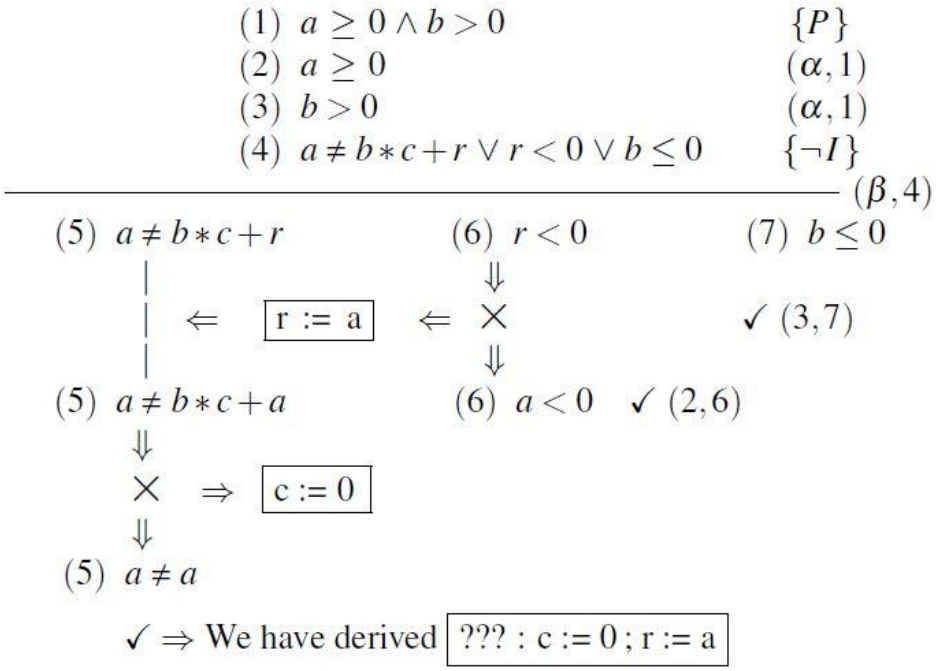


Para poder cerrar la rama abierta (7), deducimos que es necesario insertar una nueva pieza de código en el cuerpo del bucle del programa. En concreto, este síntoma tan particular de incompletitud puede ser evitado si introducimos la nueva instrucción $c := c + 1$ en el cuerpo del bucle. Ahora, si volvemos a aplicar nuestra metodología basada en tableaux semánticos sobre los restantes pasos de verificación, vemos que ya no encontramos nuevos errores, ya que los cinco tableaux semánticos permanecen cerrados. Por tanto, nuestra sesión de depuración ha terminado.

Nuestra metodología de depuración algorítmica puede aplicarse también para explicar en las clases la *derivación de algoritmos simples* (Kaldewaij, 1990). Por ejemplo, el siguiente tableau semántico para $I \wedge \neg(???) \Rightarrow Q$ permite a nuestro estudiantes derivar la condición de repetición del bucle:



Análogamente, podemos dibujar un tableau semántico para verificar el paso $\{P\}$??? $\{I\}$, y así poder derivar el código de inicialización ??? del bucle:



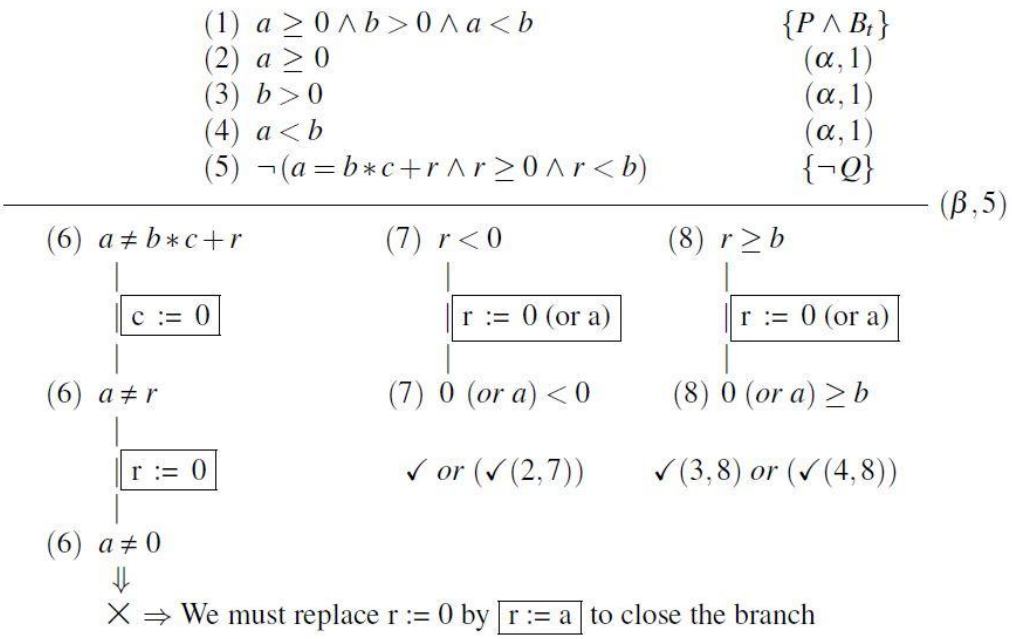
Durante el curso académico 2010/2011 hemos aplicado esta metodología basada en tableaux semánticos para diseñar clases de problemas más ilustrativas de (Kaldewaij, 1990). Todos estos ejemplos

proporcionan un excelente entrenamiento en el razonamiento necesario para poder diseñar programas correctos y eficientes.

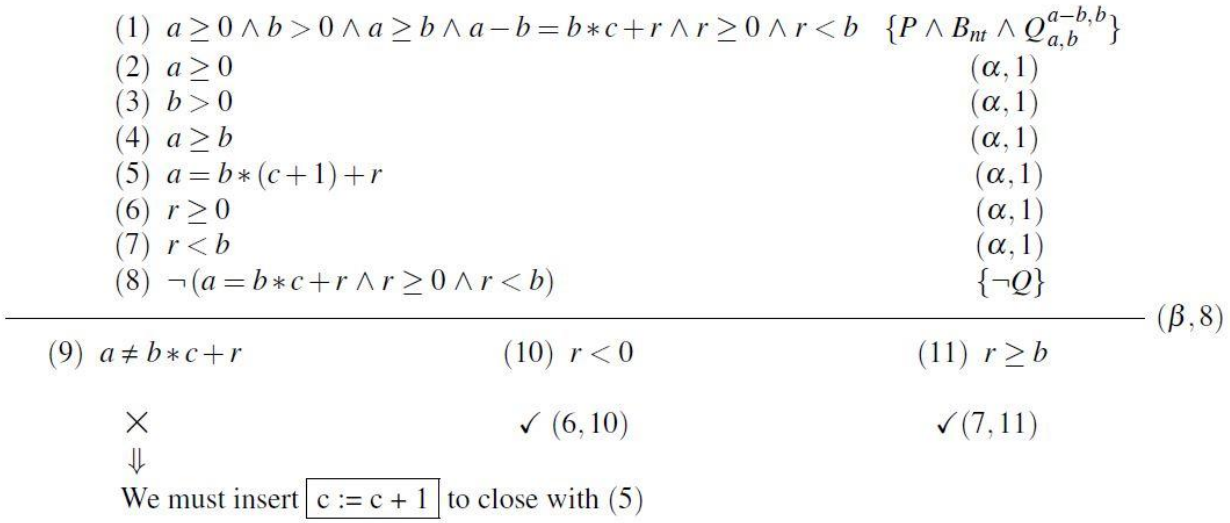
Finalmente, podemos aplicar también la técnica de depuración algorítmica basada en tableaux semánticos sobre algoritmos recursivos. Por ejemplo, vamos a alterar el código del algoritmo recursivo que hemos visto en la sección anterior para realizar la división entera con dos tipos de errores:

$\{P : a \geq 0 \wedge b > 0\}$
fun $divide\,(a, b : int)$ dev $< c, r : int >$
  if $a < b$ then $c := 0$; $\boxed{r := 0}$; ←-- *wrong code!*
  else $\boxed{< c, r > := divide\,(a-b, b)}$; ←-- *missing code!*
  fif
ffun
$\{Q : a = b*c+r \wedge r \geq 0 \wedge r < b\}$

Si tratamos de verificar esta versión errónea del algoritmo recursivo, podemos utilizar nuestra metodología y trazar el siguiente tableau semántico:



Tras esta corrección, obtenemos ya un tableau cerrado. Sin embargo, necesitamos comprobar el resto de tableaux para terminar la sesión de depuración algorítmica:



Para cerrar la única rama abierta (9), deducimos que es necesario insertar código nuevo en el programa, la instrucción de asignación $c := c + 1$. Con ello conseguimos cerrar los seis tableaux semánticos correspondientes a la verificación formal del algoritmo sin encontrar nuevos errores. Podemos concluir que el programa recursivo es ahora correcto, y por tanto, la sesión de depuración ha concluido.

CAPÍTULO 3: LA HERRAMIENTA TVT

3.1 - INTRODUCCIÓN A LA HERRAMIENTA

La herramienta TVT (Tableaux Verification Tool) está pensada y diseñada para ser de una utilidad académica para los alumnos a la hora de estudiar asignaturas que tengan que ver con el diseño de algoritmos iterativos y su verificación o su derivación.

Para ello, se dispone de una sencilla interfaz de usuario con las opciones comunes de cualquier aplicación de escritorio, pero la descripción de dicha interfaz y su uso está más desarrollada en el manual de usuario, que se encuentra en el capítulo quinto de la memoria.

En este capítulo, por otro lado, se desarrollarán explicaciones sobre los módulos más significativos de la herramienta, su esqueleto y su funcionamiento básico sin llegar a un nivel demasiado bajo, como sería la implementación de dichos paquetes.

La herramienta TVT se organiza en 8 paquetes diferentes que se encargan de las diferentes operaciones internas para verificar algoritmos iterativos, para mostrar por pantalla los resultados, y para hacer una depuración del algoritmo que en cada caso se esté usando.

Dichos paquetes son GUI, encargado de la interfaz gráfica de la ventana principal del programa, Indexicall, cuya función junto con el paquete WeakestPrecondition, LexLogAnalyzer y FormulaSimplifier es la de tratar las cadenas de entrada que se forman mediante los 5 pasos de verificación de algoritmos iterativos para poder trabajar con ellas más adelante, Drawing, que se encarga de mostrar por pantalla los tableaux semánticos fruto de la descomposición de cadenas hecha anteriormente,

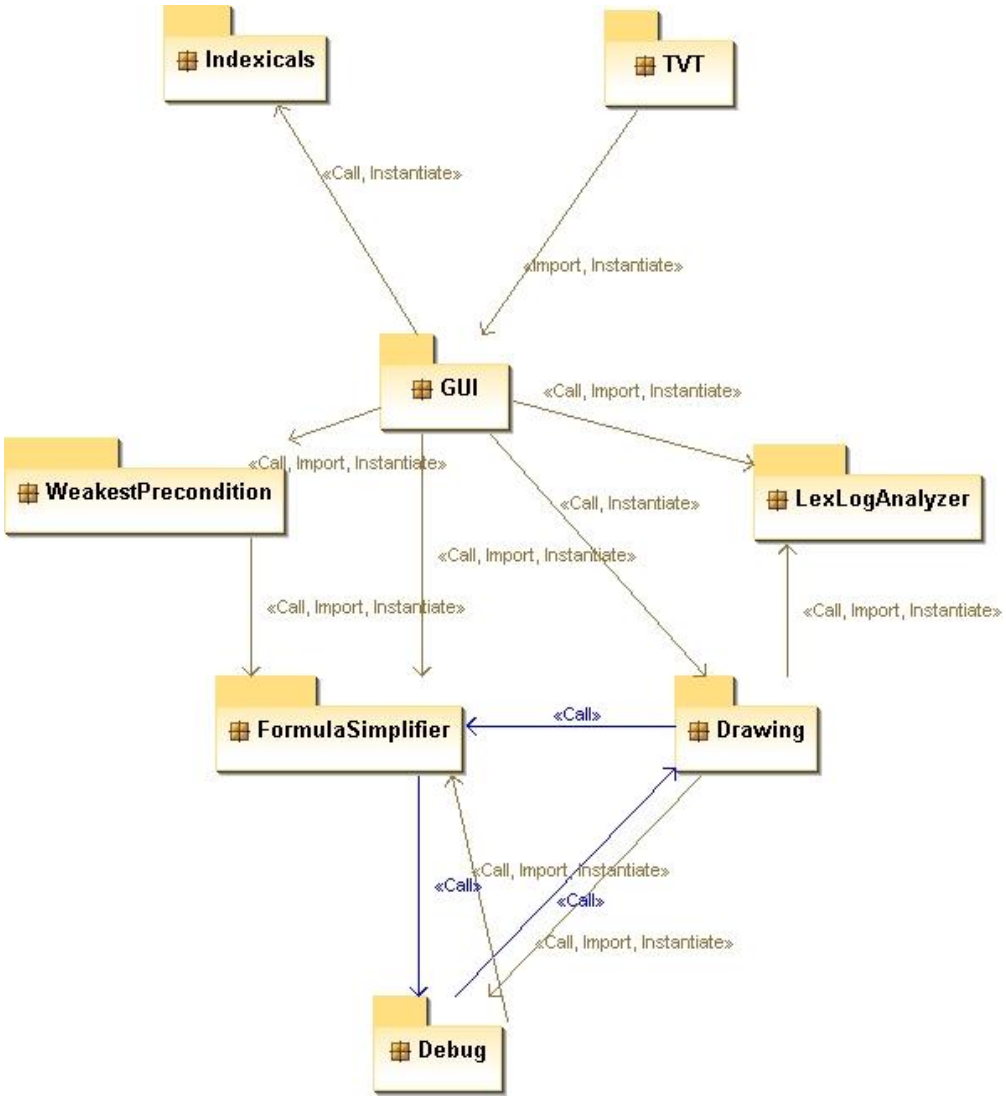
Debug, cuya misión es la de hacer la depuración de cada árbol para comprobar que todas las ramas se cierran correctamente para demostrar que el algoritmo es correcto, y por último TVT, que es la llama principal de la aplicación.

A continuación, se procede a desarrollar el cómo se relacionan los paquetes entre si y los elementos más importantes de cada uno de ellos.

3.2 - DIAGRAMAS DE CLASES

3.2.1 Diagrama de clases general

Estos son los diagramas de clases pertenecientes a nuestro proyecto. El primero muestra los diferentes paquetes como se relacionan entré si, mostrando la herencia, asociaciones y dependencias entre ellos. Los restantes diagramas corresponden a los diferentes paquetes de los que se compone el proyecto, mostrando cómo se relacionan las clases que los forman entre sí o con otras clases del proyecto.



3.2.2 Diagrama de Clases de Paquete GUI

El paquete GUI es el que gestiona la interfaz de usuario principal, en la que el mismo deberá introducir un algoritmo iterativo así como las propiedades lógicas y poder iniciar desde éste módulo todas las operaciones necesarias para verificar dicho algoritmo.

Contiene la clases PrincipalW, que es la encargada de lanzar el dibujado de árboles, así como la edición de algoritmos y las diferentes llamadas a cada una de las subclases encargadas de introducir código preestablecido en el cuerpo del programa, o símbolos lógicos y matemáticos en las fórmulas lógicas.

El proceso, tal y como se muestra en la figura (num de figura) es el siguiente. Una vez introducidos los datos de entrada, y seleccionado uno de los árboles de verificación, tras un checkeo básico de posibles errores de entrada se simplifica la cadena del String perteneciente al árbol que va a ser dibujado separando los rangos(indexicall), simplificándolos(FormulaSimplifier) y dividiéndolo en fórmulas alfas o betas(LexLogAnalizer) según corresponda. Previamente se ha calculado la precondición más débil si es necesaria (WeakestPrecondition).

Una vez conseguido esto, y si no se han dado errores, la herramienta procederá a la ejecución de dibujar el árbol resultante.

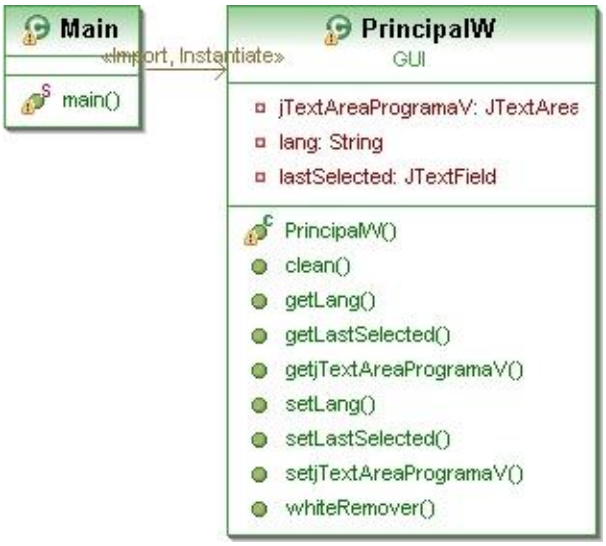
Además de esta función, que es la principal y más importante, en el paquete GUI se pueden hacer operaciones menores de cara al usuario, como introducir porciones de código en base a unos esqueletos, es decir, funciones, bucles, condicionales, así como los símbolos lógicos necesarios para definir la precondición, postcondición, invariante y cota del bucle.

Por último, se agregó la funcionalidad necesaria para poder tratar con archivos de texto por comodidad a la hora de hacer ejercicios,

cargarlos, guardarlos, y poder enviarlos por correo o lo que fuera necesario, tanto por equipo docente como por el alumnado.

3.2.3 Diagrama de Clases de Paquete TVT

Este paquete simplemente es el enlace principal con el resto de la aplicación. Es el que contiene la función Main que llama a PrincipalW de GUI.



3.2.4 Diagrama de Clases de paquete Indexicalls

La función principal de este paquete es la de tomar una cadena de tipo ($x_1 \square x_2 \square x_3 \square \ldots \square x_n$) donde $\square$ puede ser {<, >, <=, >=} y separarla en una sucesión de fórmulas alfa consecutivas equivalente de la manera ($(x_1 \square x_2) \wedge (x_3 \square x_4) \wedge \ldots \wedge (x_{n-1} \square x_n)$)

Para ello se hace una llamada a doIndexicall, un método perteneciente a este paquete que es el principal encargado de aplicar el algoritmo de simplificación.

3.2.5 Diagrama de Clases de paquete FormulaSimplifier

Este módulo se encarga de simplificar una fórmula representada por una cadena de caracteres.

Para el caso de fórmulas de lógica proposicional, simplificaría la fórmula a tratar reduciéndola a únicamente cláusulas del tipo α ($i.e.$ $((p \wedge q) \wedge r)$), del tipo β ($i.e.$ $((p \vee q) \vee r)$) o una combinación de ambos tipos ($i.e.$ $((p \wedge q) \vee r)$). De esta manera, el algoritmo liberaría dicha fórmula de todas las conectivas binarias que no fueran del tipo α y β. Las siguientes conectivas binarias son soportadas: $\wedge, \vee, \rightarrow, \leftrightarrow, \oplus$.

Si en el proceso se encontrara con ecuaciones o inecuaciones susceptibles de ser simplificadas ($i.e.$ $\neg(a = b) \sim (a \neq b)$; $\neg(a < b) \sim (a \geq b)$), también las trataría. Los siguientes operadores lógicos son soportados: $<, >, \leq, \geq, =, \neq$.

Nótese que las ecuaciones aritméticas no se negarían. Siendo éstas una parte integrante de una fórmula lógica mayor, sería ésta la susceptible de ser negada si así fuera necesario. Cabe destacar que las fórmulas precisan de una prioridad establecida por el usuario por pares de literales, tal y como se aprecia en los ejemplos iniciales.

El flujo de ejecución del paquete de *FormulaSimplifier* una vez recibe una fórmula como entrada sigue los siguientes pasos:

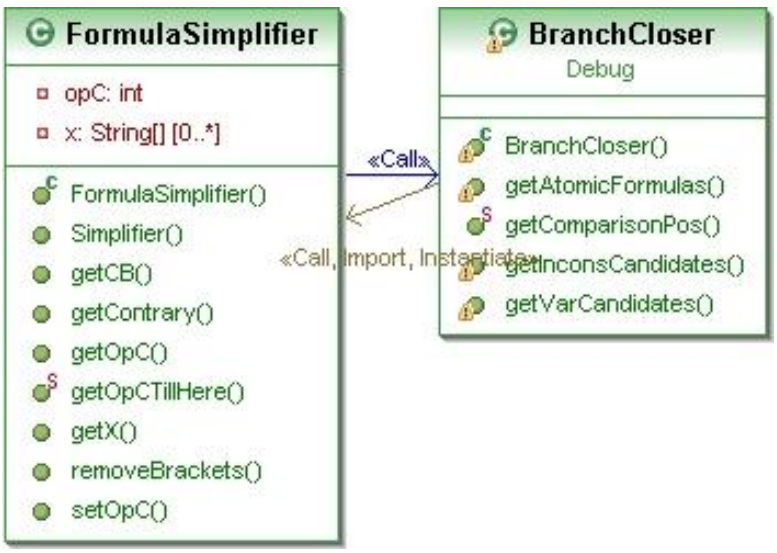
1. Simplifica todas las apariciones en la fórmula de los siguientes tipos de negaciones, derivando la misma a cada una de sus subcláusulas:
   a) $\neg(\neg([\ldots]))$
   b) $\neg(([\ldots]))$

2. Mientras existan conectivas binarias del tipo $\leftrightarrow, \rightarrow, \oplus$ o negaciones del tipo $\neg([\ldots])$, simplificará sus cláusulas hasta reducir toda la

fórmula a sólo cláusulas del tipo α y β y negaciones directas de literales (dicha simplificación está dotada de cierta prioridad, definida por el orden de conectivas expuesto al inicio de este punto).

Es necesario aclarar que las negaciones de cuantificadores (existenciales, universales, etc.), no son tratadas del mismo modo, sino de manera externa, preservando así la legibilidad e interpretación de los mismos.

A lo largo de todas las simplificaciones, se recurren a algoritmos que aumentan la claridad de la información, eliminando paréntesis sobrantes y negaciones del tipo ¬¬[…] .



3.2.6 Diagrama de Clase de Paquete LexLogAnalyzer

LexLogAnalyzer tiene como misión fundamental subdividir la cadena simplificada por los módulos Indexicall y FormulaSimplfiera y si es necesario WeakestPrecondition y convertirlo en una estructura de datos de tal manera que sea posible tratar con ella de manera que se pueda dibujar después el tableaux.

Para ello, en la clase lexLogAnalyzer el String que tiene por entrada el método analyzeLog es subdividido utilizando como separadores los paréntesis(no se usó prioridad de operadores lógicos para facilitar la entrada de datos al usuario) entre 2 formulas, agrupándose a izquierda, de la siguiente manera:

((((a□b) □c) □d) □e) □ ((a□b) □c)

Donde □ es ^ o ˅ según se dé.

Como estructuras de datos se utilizaron auxiliarmente una clase Data, que contiene los dos Strings correspondientes a la formula dividida, y una clase Key, que contiene información relevante a la posición en el árbol gracias a su atributo “n”.

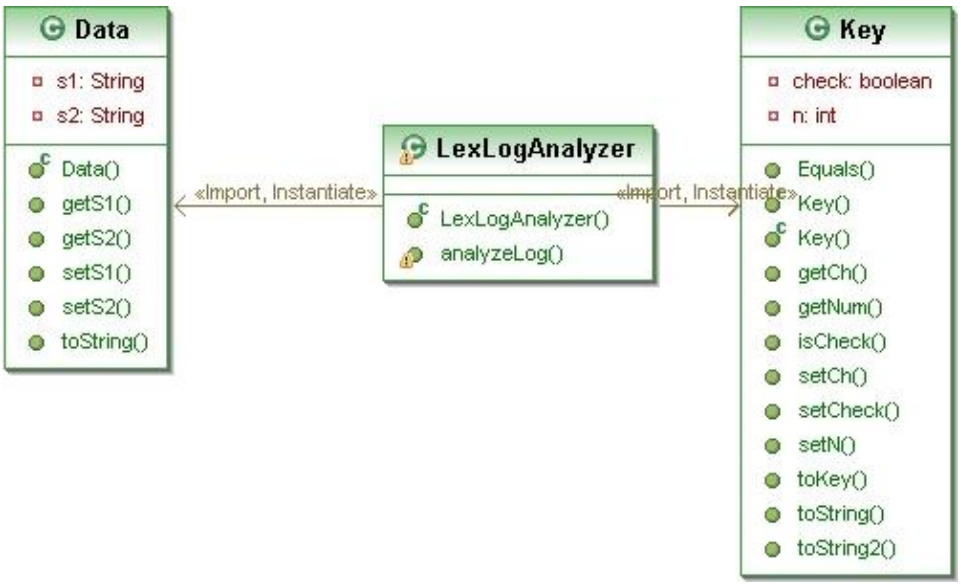
Dicha Key es la clave que un HashMap utilizara para almacenar las diferentes Datas, de manera que cada clave incluye información sobre la posición que ocupa en el árbol(en función de cómo de interna sea la subfórmula) y está asciada a una Data que son sus dos Strings. Por ejemplo: ((a^b) ˅(c^d))

Daría como resultado el siguiente HashMap(no tiene por qué estar en orden):

Key	Data
1, b	(a^b), (c^d)
11, a	a, b
12, a	c ,d

"a" o "b" indica si es alfa o beta la fórmula, y el número entero es la clave que indica la posición en el árbol y a su vez, quien es el padre de dicha subfórmula. De ésta manera es muy sencillo e intuitivo comprender a nivel de implementación como está formado un árbol antes de dibujarlo.

Para ir dividiendo el String de entrada, se usó una pila auxiliar hasta que estuviera vacía, momento en que toda subformula había sido tratada.

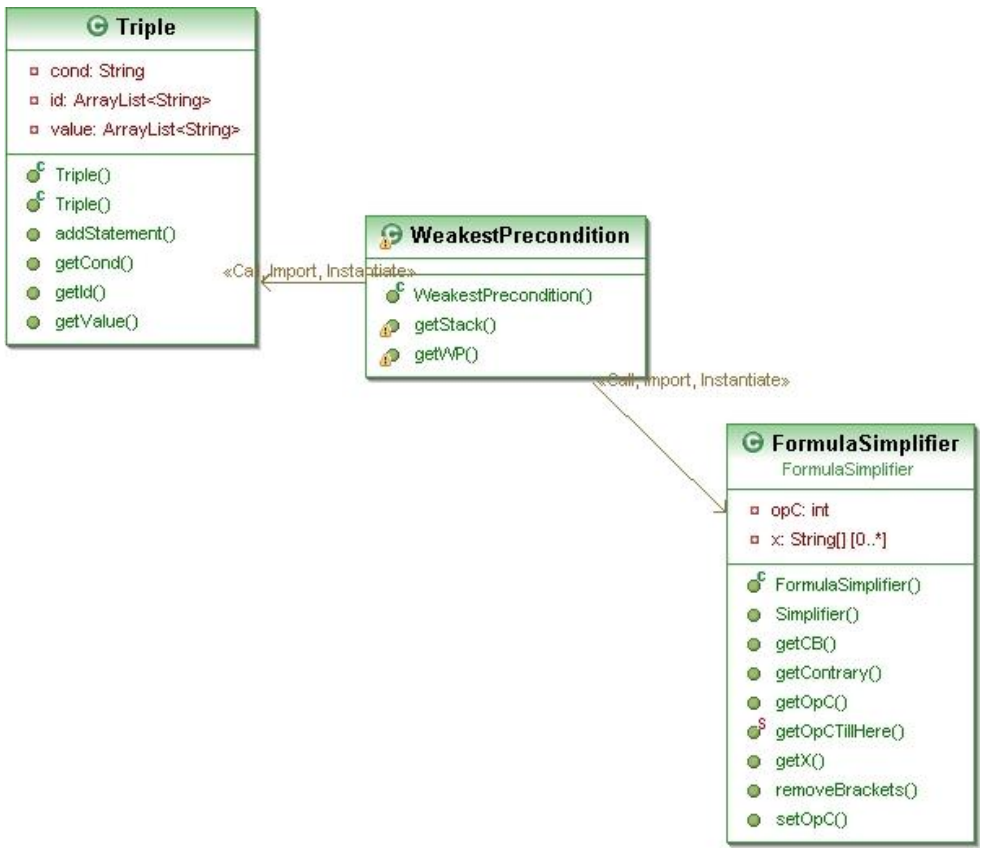


3.2.7 Diagrama de Clase de Paquete WeakestPrecondition

El cometido de este paquete es el de calcular la precondición más débil para aquellos árboles en los que se precise.

Haciendo uso de una clase auxiliar denomada *Triple*, cuya función es la de almacenar una tupla de 3 elementos, a saber: *id*, que contiene el identificador de una asignación (el nombre de su variable, parte izquierda); *value*, que contiene el valor asociado a una asignación (el valor de su variable, parte derecha); y *cond,* cuya misión es la de almacenar la condición que determina si el valor de *id* debe ser sustituido o no por el de *value.*

Así pues, dependiendo de si la asignación está sujeta a posibles condiciones (dependen de la condición de una instrucción *if*) o no (son asignaciones libres, sin dependencias de instrucciones *if*), *Triple* gestiona su sustitución para el cálculo de la precondición más débil.



3.2.8 Diagrama de Clase de Paquete Debug

El paquete *Debug* tiene como principal objetivo proporcionar al usuario una manera sencilla, intuitivo y guiada por pasos de verificar o depurar su algoritmo.

El esquema general para alcanzar dicho objetivo es crear un proceso para cada árbol generado en el cual el usuario podrá ir decidiendo qué ramas son susceptibles de ser cerradas y cuáles no. Si al finalizar dicho proceso, todas las ramas han sido cerrdas, se cumplirá la prueba formal que generó dicho árbol. En caso contrario, habría que revisar el algoritmo en busca de fallos.

Para el cierre de las ramas, *Debug* cuenta con la ayuda de la clase *BranchCloser.* Dicha clase gestiona las posibles inconsistencias y tuplas de fórmulas atómicas pertenecientes a cada rama, ofreciéndoselas al usuario para que éste, como oráculo, determine si cierran, individualmente o en conjunto, su propia rama.

3.2.9 Diagrama de Clase de Paquete Drawing

Sin duda uno de los paquetes que más líneas de código junto con el FormulaSimplfier ha llevado, ya que mostrar la solución de una manera intuitiva y sencilla para el usuario era uno de los objetivos primordiales de la herramienta.

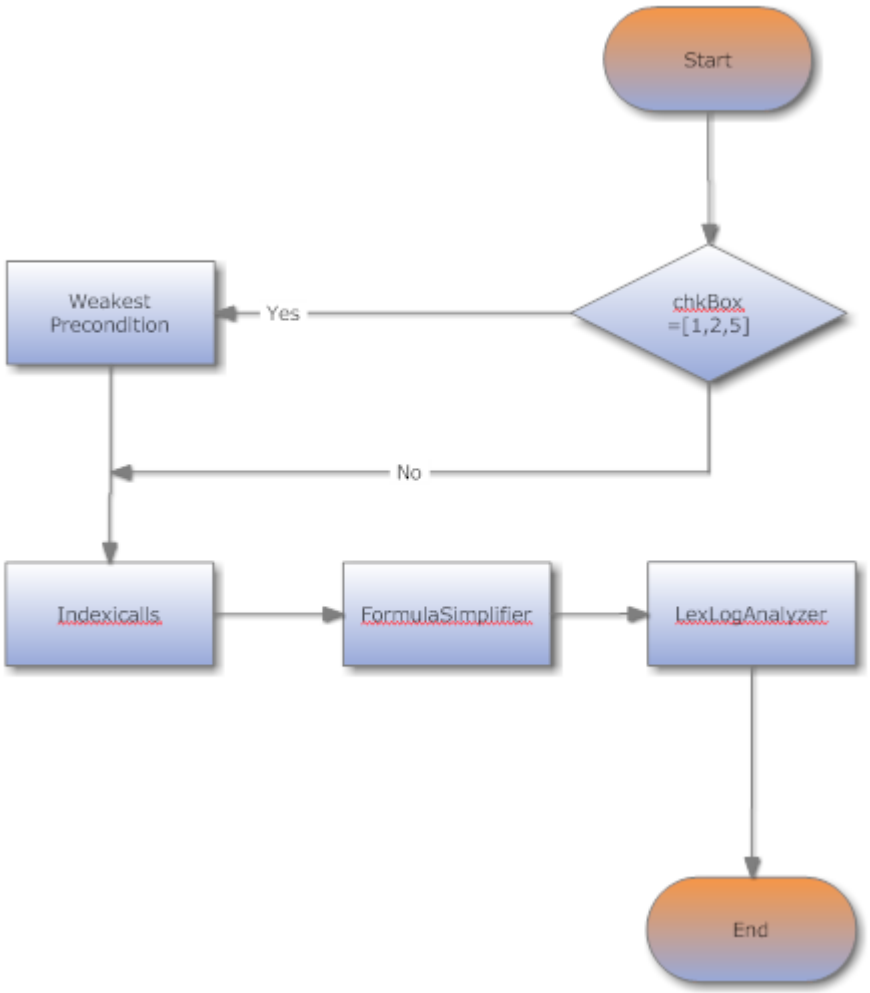
Drawing se compone de la clase Draw, cuya equivalencia más comprensible sería la de un motor de cálculo, MyCanvas, que es un extend de la clase de Java Canvas y es la utilizada para dibujar los árboles en las coordenadas dadas y por último PixelDPaint, StringDPaint y LineDPaint que son clases auxiliares donde se almacena la información necesaria para pintar las líneas, puntos y Strings requeridos para la visualización correcta del árbol.

Draw hace una llamada al método drawingCalculation, que mediante el uso de métodos auxiliares como alfaPaint y BetaPaint calcula todas las coordenadas en función de una estructura TreeSet obtenida a partir del HashMap donde previamente se separaron las cadenas. Este estructura TreeSet tiene un orden en función del atributo que se le dé, en nuestro caso Key para dibujar los árboles en un orden lógico correcto y acorde con las cadenas, ya que el HashMap no tiene orden alguno.

Una vez que drawingCalculation termina, y mediante unas sencillas llamadas a métodos de ajuste del árbol en el canvas, se hace la llamada a la clase MyCanvas, encargada de coger los datos de drawingCalculation, es decir, una estructura de datos que contiene las coordenadas de líneas, puntos y Strings a ser dibujados para mostrarlo correctamente en una nueva pantalla para el usuario.

3.2.10 Otros diagramas

El siguiente diagrama muestra el camino de ejecución que recorre la fórmula desde que se selecciona el árbol a generar.



3.3 - PLATAFORMAS Y TECNOLOGÍAS UTILIZADAS

Para el desarrollo de nuestra aplicación, hemos utilizado especialmente dos herramientas. Para llevar a cabo la implementación hemos usado el programa Netbeans IDE 6.9.1. El almacenamiento de los archivos durante el proceso de desarrollo de la herramienta se ha realizado usando un cliente de subversion: Tortoise SVN.

Netbeans IDE 6.9.1 es un reconocido entorno de desarrollo integrado disponible para numerosos sistemas operativos, tales como Windows, Linux, Mac y Solaris. Es un producto libre y gratuito que permite usarlo sin restricciones y permite a los desarrolladores crear rápidamente aplicaciones utilizando la plataforma Java, así como PHP, JavaFX, JavaScript y Ajax, Ruby, Groovy and Grails y C/C++.

En nuestra aplicación utilizamos la plataforma Java y JavaFx . Uno de los motivos por los que nos decidimos a utilizar Netbeans frente a otros conocidos entornos de desarrollo como por ejemplo Eclipse, fue que nos facilitaba el trabajo a la hora de desarrollar las interfaces gráficas del proyecto. Además estábamos familiarizados con su utilización debido a las diferentes prácticas realizadas durante la carrera.

TortoiseSVN es un sistema gratuito de código abierto para el control de versiones. Tortoise almacena ficheros en un repositorio central. Podemos crear subdirectorios para mejorar su accesibilidad. La ventaja de su uso radica en que no solo se guarda la última versión del fichero, sino todos los cambios realizados y las versiones anteriores, para así poder recuperar otras versiones del proyecto o poder consultar cómo, cuándo o quién realizó ciertos cambios.

El uso del SVN nos ha facilitado mucho el trabajo, ya que teníamos accesible el código actualizado en cualquier momento. Cuando alguien del equipo desarrollaba algo nuevo, lo subía al SVN para que todos los demás tuviesen una versión actualizada para trabajar y así evitar solapamientos o conflictos con las versiones. Además en caso de necesitar consultar versiones anteriores las teníamos disponibles en cualquier momento. Además nuestro director de proyecto podía consultar cuando lo deseara los cambios realizados o probar la funcionalidad de la versión más actual de la práctica.

3.4 - PARTE INTERACTIVA Y PANELES GRÁFICOS

A la hora de pintar el árbol uno de los principales métodos es drawingCalculation, al que le entra un hashMap relleno previamente por lexLogAnalyzer con las cadenas separadas; un treeSet que contiene el hashMap ordenado por Key, para poder dibujar el árbol en el orden correcto; un entero con la altura del canvas y otro con la anchura. DrawingCalculation devuelve un arrayList de ArrayList con los puntos, rallas y strings que se pintarán en el canvas. Este método crea un iterador para recorrer el treeSet. En función de si en cada momento la información del treeSet es una fórmula alpha o beta, llamará a los métodos auxiliares alfaPaint y BetaPaint, que calcularán las coordenadas de los siguientes puntos a pintar.

```
public ArrayList drawingCalculation(HashMap<Key, Data> h, TreeSet t,
int a, int b) {

    ArrayList<ArrayList> resp = new ArrayList<ArrayList>();

    resp.add(new ArrayList<PointDPaint>());

    resp.add(new ArrayList<LineDPaint>());

    resp.add(new ArrayList<StringDPaint>());

    Iterator it = t.iterator();

    Key k = new Key();   Data d;

    ArrayList<PixelInfo> openBranches = new ArrayList();

    while (it.hasNext()) {

        String s = (String) it.next();

        k = k.toKey(s);

        k = this.getKeyMemDirection(h, k);
```

```
            d = h.get(k);
            if (s.contains("a")) {
                resp = alphaPaint(d, k, openBranches, resp, a, b);
            } else if (s.contains("b")) {
                resp = betaPaint(d, k, openBranches, resp, a, b);
            }
        }
        OB = openBranches;
        return resp;
}
```

Una vez que tenemos calculados todos los puntos, líneas y strings que utilizaremos para pintar los árboles, se llama a la clase myCanvas, que mediante el método paint, dibujará el árbol que el usuario visualizará. En este método mediante un bucle se dibujará un punto, una raya y el string correspondiente para cada componente del array.

```
private ArrayList<ArrayList> draw;
 private DrawedTableau tab;
 private ArrayList<PixelInfo> tags;
public void paint(Graphics g) {
        ArrayList auxal;
        g.setColor(Color.RED);
        for (int i = 0; i < draw.size(); i++) {
            auxal = draw.get(i);
            if (i == 0) { for (int j = 0; j < auxal.size(); j++) {//points
```

```
                g.fillOval(((PointDPaint) auxal.get(j)).getX(), ((PointDPaint)
auxal.get(j)).getY(), 7, 7);
            }
        } else if (i == 1) {
 for (int j = 0; j < auxal.size(); j++) {//lines
                g.drawLine(((LineDPaint) auxal.get(j)).getX1(), ((LineDPaint)
auxal.get(j)).getY1(),
                        ((LineDPaint) auxal.get(j)).getX2(), ((LineDPaint)
auxal.get(j)).getY2());
            }
        } else {
            int n = 1;
            for (int j = 0; j < auxal.size(); j++) {//strings
                g.setColor(Color.BLUE);
                String aux = ((StringDPaint) auxal.get(j)).getS();
                aux = replaceSymbols(aux);
                g.drawString(aux, ((StringDPaint) auxal.get(j)).getX(),
                        ((StringDPaint) auxal.get(j)).getY());
                for (int k = 0; k < tags.size(); k++) {
                    String string1 = ((StringDPaint) auxal.get(j)).getS();
                    int k1 = ((StringDPaint) auxal.get(j)).getK();
                    String string2 = ((PixelInfo) tags.get(k)).getF();
                    int k2 = Integer.parseInt(((PixelInfo) tags.get(k)).getS());
                    if (string1.equals(string2) && k1 == k2) {
                        int x = ((StringDPaint) auxal.get(j)).getX();
                        int y = ((StringDPaint) auxal.get(j)).getY() + 20;
                        g.drawString("[" + ((Integer) n).toString() + "]", x, y);
```

```
                    n++;
                    break;
                }
            }
        }
    }
}
```

CAPÍTULO 4: VERIFICACIÓN Y DEPURACIÓN DE PROGRAMAS CON TVT

4.1 - EJEMPLO DE VERIFICACIÓN

A continuación mostraremos paso a paso la verificación de un programa iterativo usando nuestra herramienta TVT:

Precondición: (a>=0)/\(b>0)

Postcondición: ((a=b*c+r)/\(r>=0))/\(r<b)

Invariante: ((a=b*c+r)/\(r>=0))/\(b>0)

Cota: r

Programa:

```
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
```

Para la verificación del programa habrá que comprobar que se cumplen 5 condiciones:

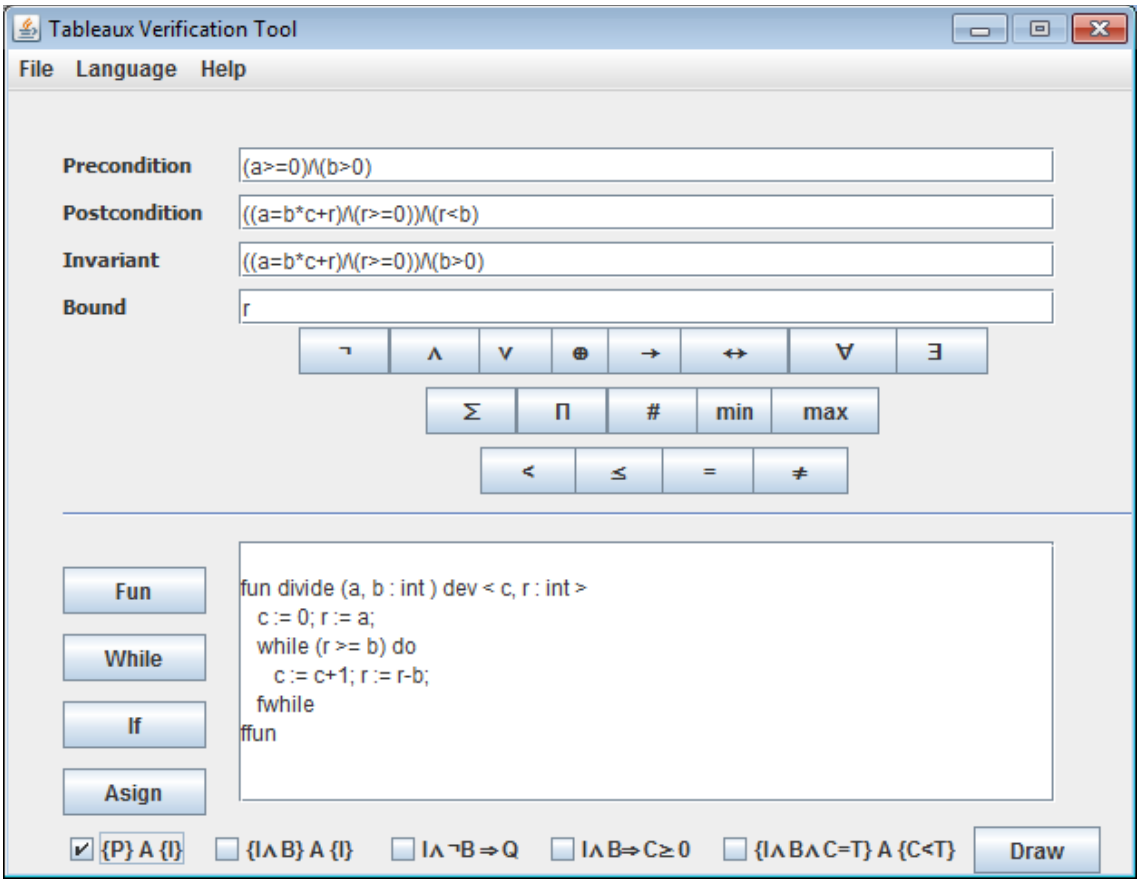
1.El invariante se cumple antes de la primera iteración del bucle.
2.Mientras se ejecuta el cuerpo del bucle A, el invariante se mantiene.
3.El invariante se sigue cumpliendo al salir del bucle.

4.C será una función dependiente de las variables del bucle que garantice que este termina. En este caso mientras se cumpla la condición B, C será mayor que cero.
5.C decrecerá al ejecutarse el cuerpo del bucle.

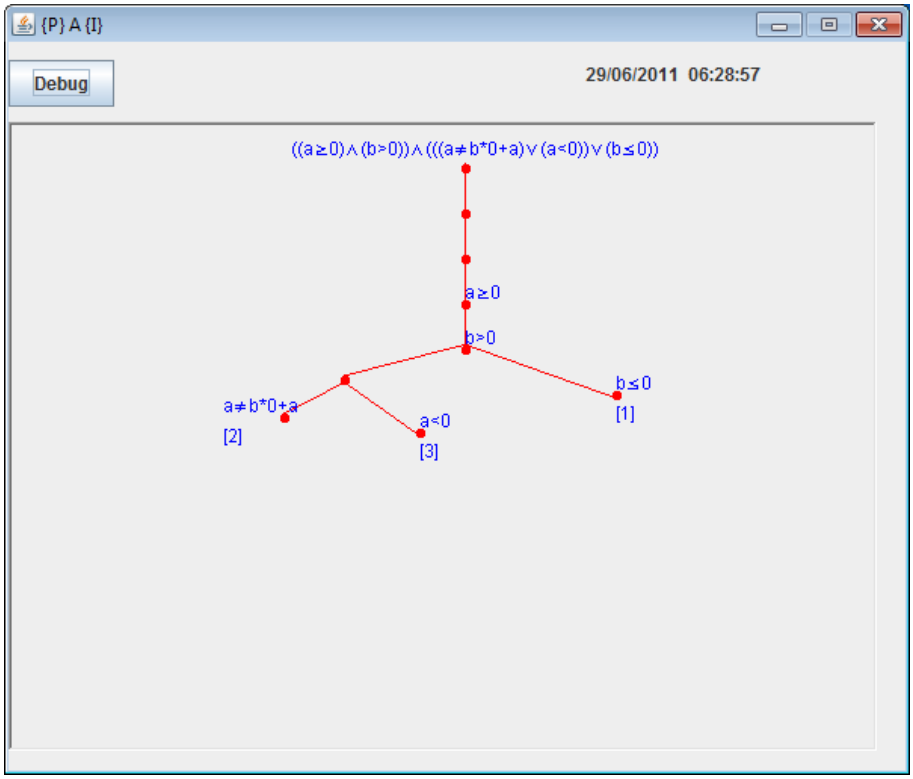
Cada una de estas pruebas formales las seleccionaremos mediante un check box de la herramienta.

A continuación, verificaremos la primera prueba formal:

{P} A {I}

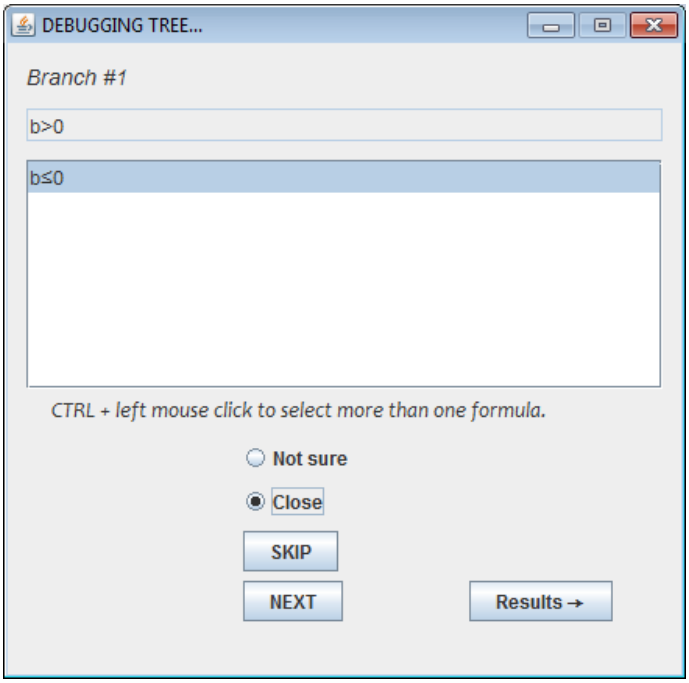


Pulsamos el botón draw para ver el árbol generado:

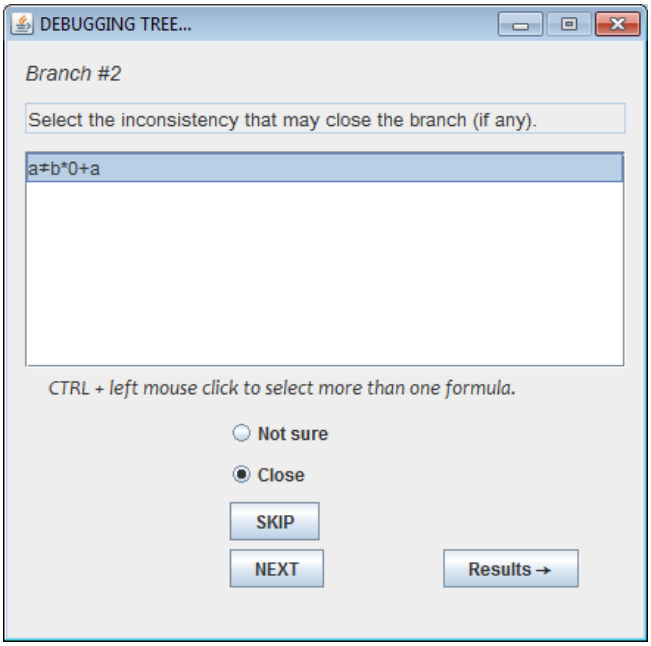


Se puede ver la fórmula resultante para {P}A{I} y su dibujo en forma de árbol, ramificándose las fórmulas beta. A continuación pulsamos el botón *Debug* para comprobar la corrección del algoritmo, iniciándose la fase de depuración:

Se nos mostrará una de las fórmulas del árbol junto con las fórmulas candidatas para cerrar esa rama. Para cerrar la rama, la fórmula ofrecida en el text box deberá ser la opuesta a la fórmula principal mostrada.

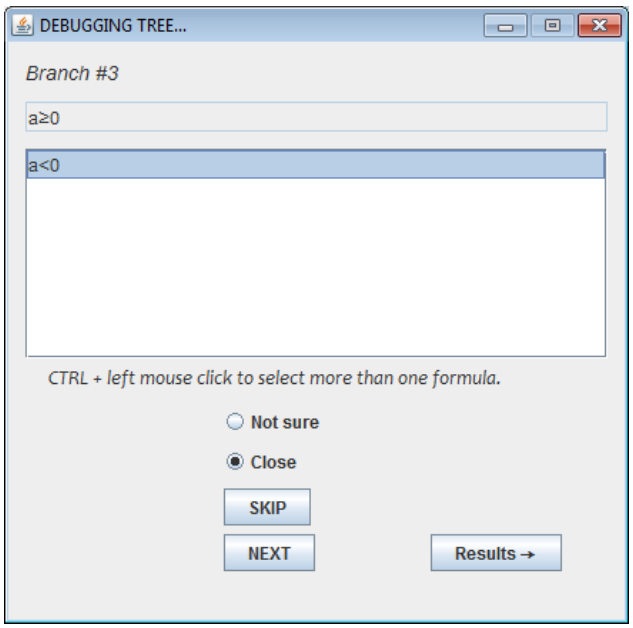


Se puede ver claramente que las fórmulas mostradas son opuestas por lo que seleccionamos la fórmula, marcamos *Close* y pulsamos *Next*.

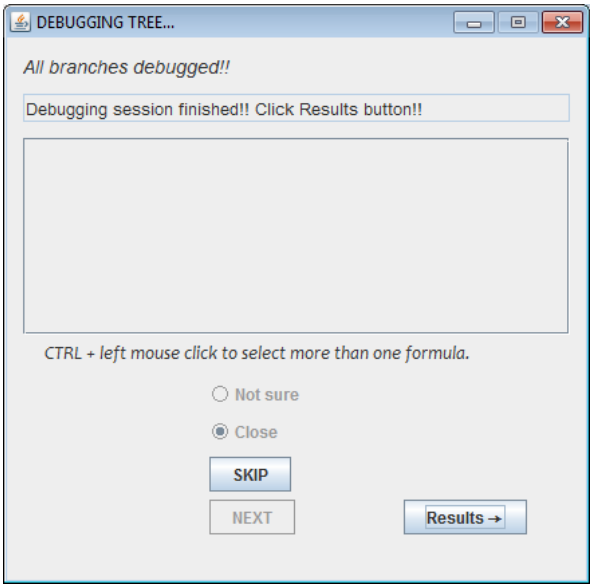


La siguiente opción mostrada nos indica que es una inconsistencia, que quiere decir que no ha encontrado ninguna fórmula opuesta para mostrar. En este caso si no estamos seguros de la falsedad

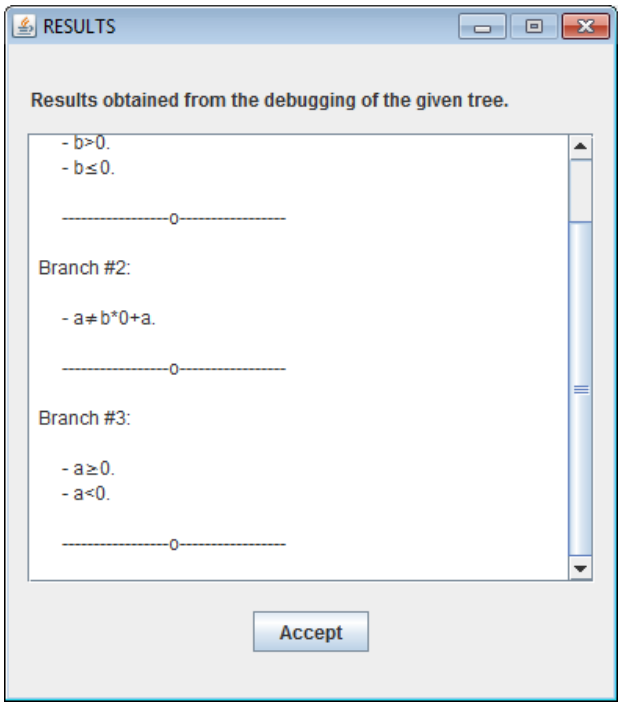
de la fórmula mostrada pulsaríamos *Not Sure* pero se puede ver fácilmente que debido a la multiplicación por cero la fórmula mostrada se simplifica como a ≠a . Al ser falsa la afirmación seleccionamos la opción *Close* y pulsamos en *Next*.



Podemos asegurar que la fórmula principal y la fórmula candidata son opuestas, por lo que marcaremos la opción *Close* y pulsaremos en *Next*.



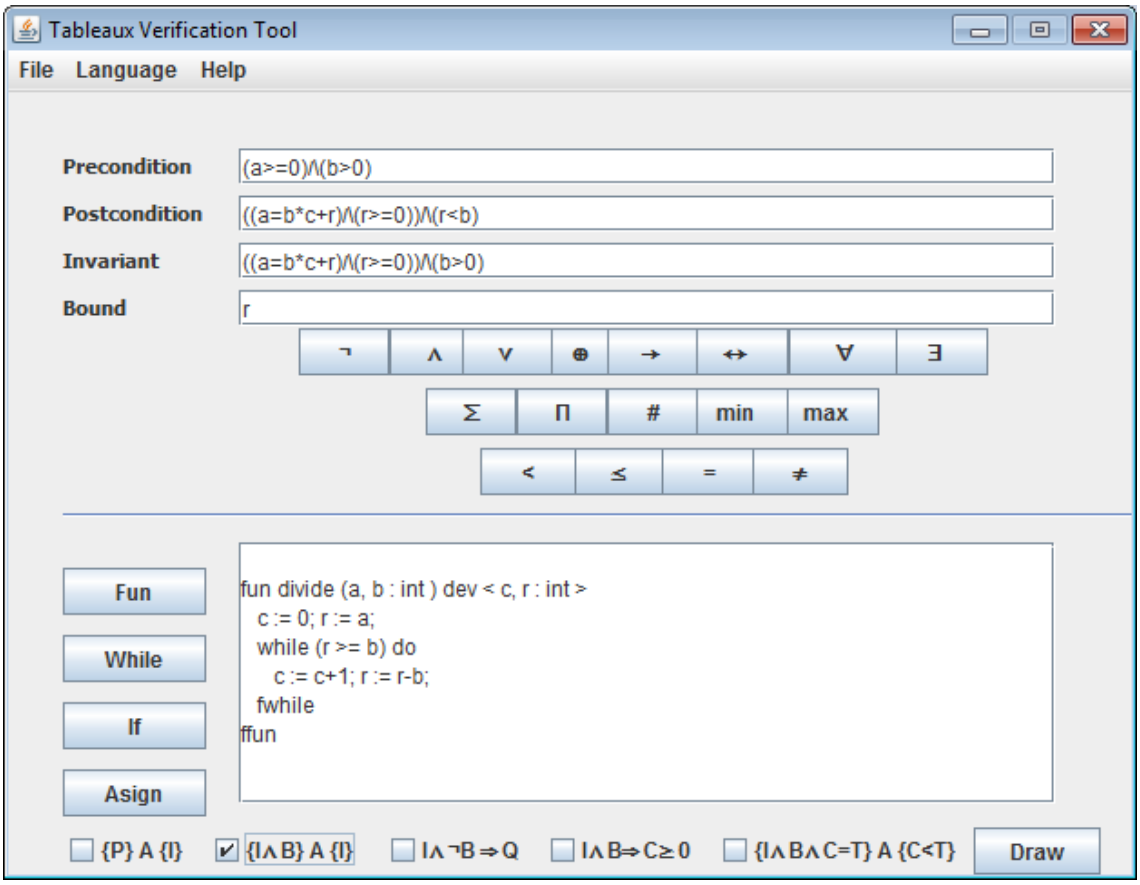
La herramienta nos informa de que recorrido todas las ramas. A continuación pulsaremos el botón *Results* para comprobar los resultados.



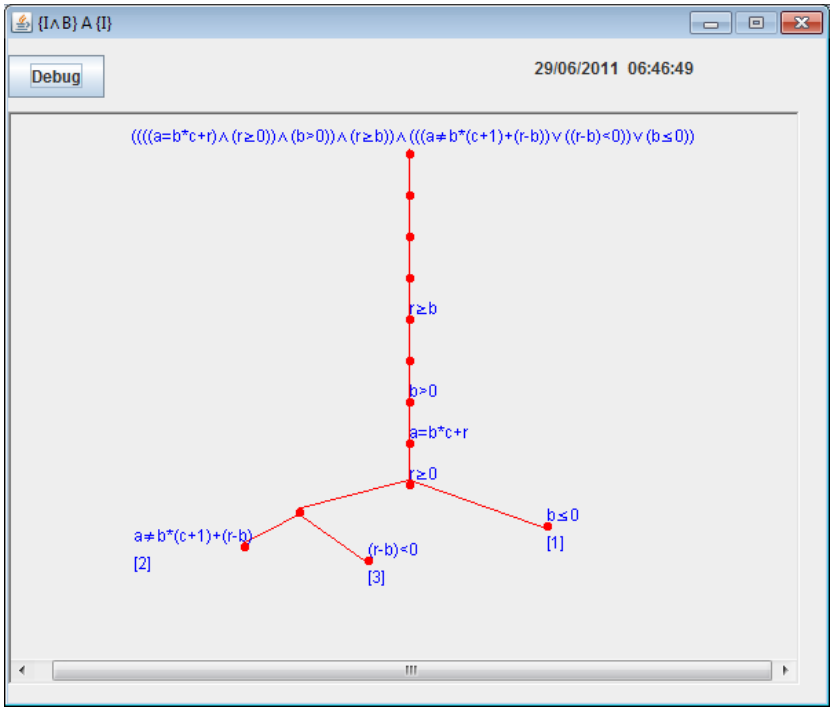
Se nos muestran todas las ramas con las fórmulas opuestas que verifican la corrección del cierre de todas las ramas, en caso de que alguna de las ramas no se hubiese cerrado la rama se mostraría como abierta.

A continuación, verificaremos la segunda prueba formal:

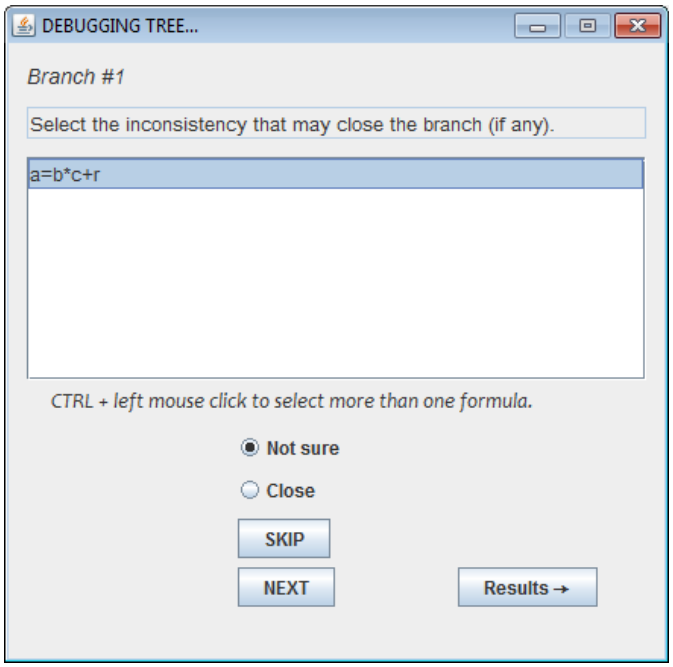
$$\{I \wedge b\}\ A\ \{I\}$$



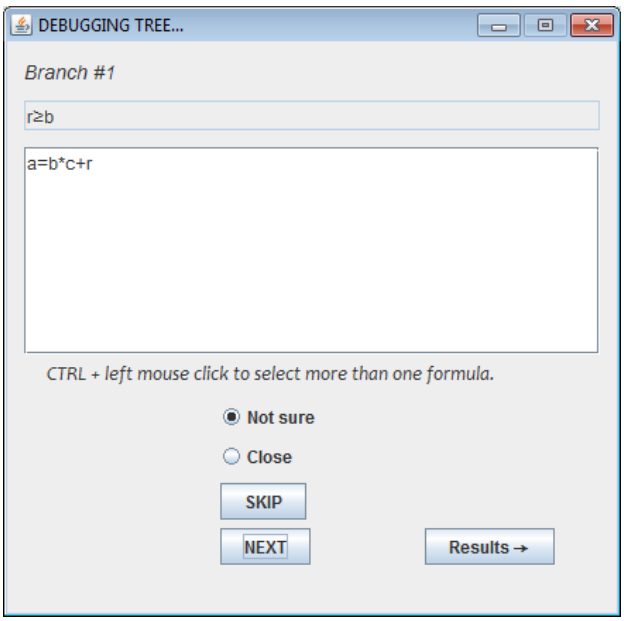
Tras seleccionar la opción deseada volvemos a pulsar el botón *Draw*.



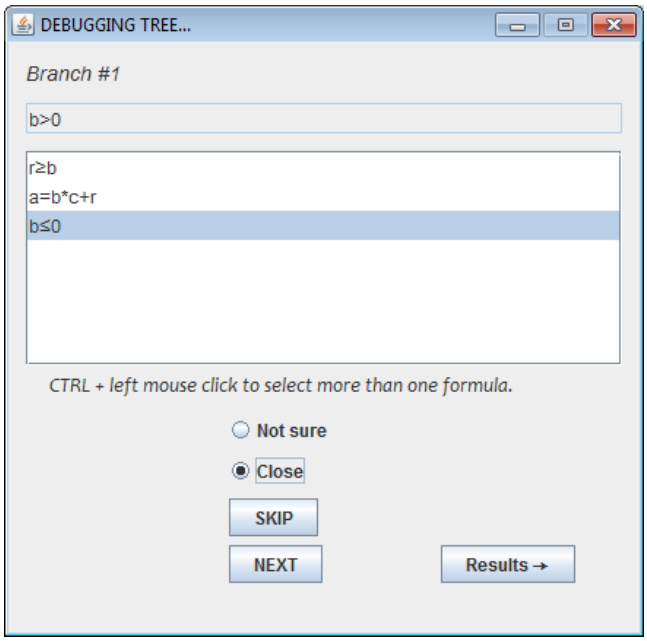
Para comenzar a depurar el árbol mostrado pulsaremos el botón *Debug*.



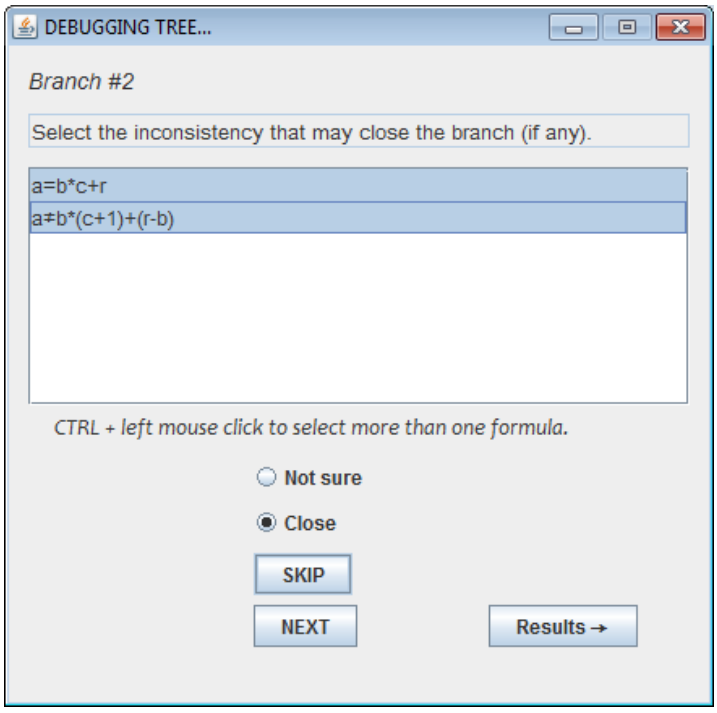
Nos facilita la primera fórmula a analizar, que es una inconsistencia, ya que viendo solo la fórmula ofrecida no podemos estar seguros de su corrección. Seleccionaremos la fórmula , marcaremos la opción *Not Sure* y pulsaremos el botón *Next.*



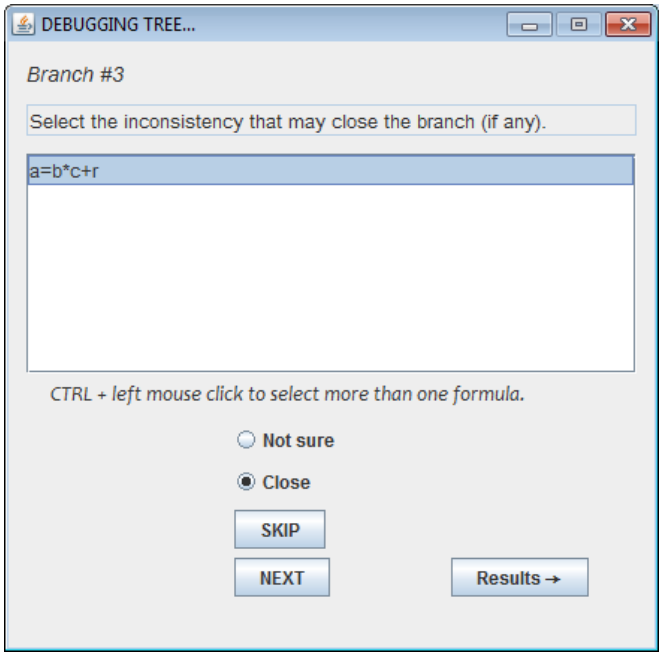
La siguiente fórmula ofrecida es r>=b y nos ofrecen como candidata para cerrar la rama a=b*c+r que no nos permite cerrar la rama ya que no podemos saber si ambas fórmulas son opuestas, por ello seleccionamos la fórmula, la opción *Not Sure* y pulsamos *Next.*



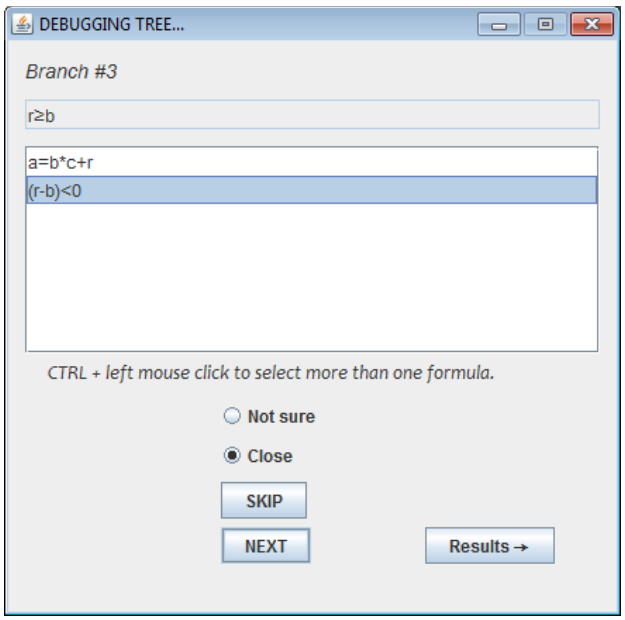
La fórmula principal es b>0 y se puede ver claramente que entre las candidatas ofrecidas su opuesta es b <=0, por lo que seleccionamos esta, la opción *Close* y pulsamos el botón *Next*.



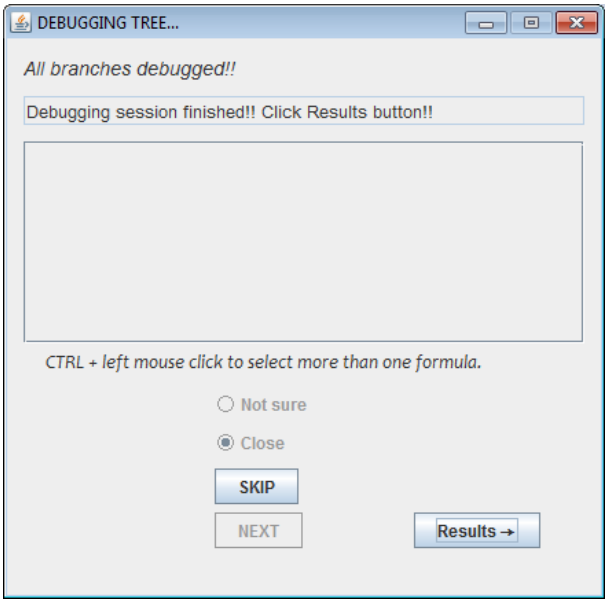
Las fórmulas mostradas son dos inconsistencias, pero si las comparamos se ve que son entre sí opuestas. a≠b*(c+1)+(r-b) se puede desarrollar como a≠b*c +b+ r-b, que se puede simplificar como a≠b*c+r , que es opuesta a la primera ofrecida, por lo que marcamos las dos, seleccionamos la opción *Close* y pulsamos en *Next*.



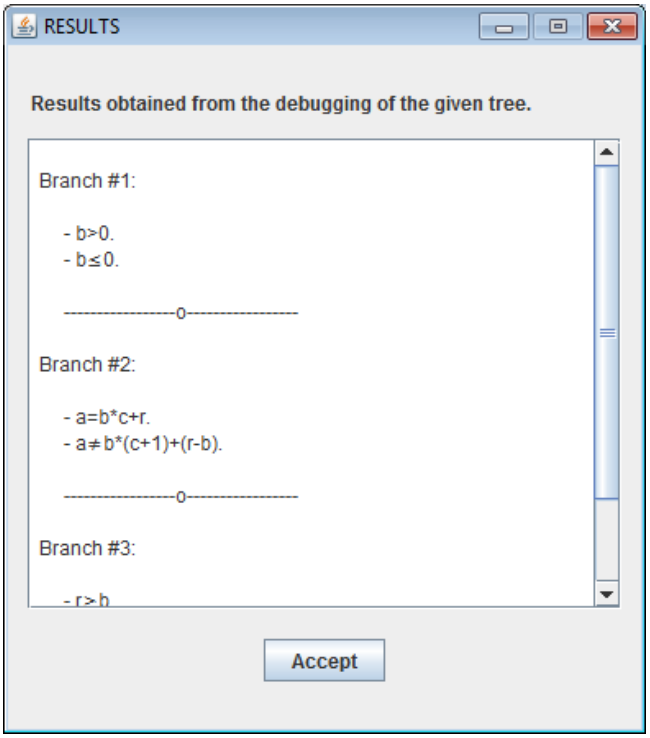
Nos vuelve a ofrecer una inconsistencia con la que no sabemos si se puede cerrar la rama o no. Como en casos anteriores, la seleccionamos, marcamos la opción *Not Sure* y pulsamos en *Next.*



Se ve fácilmente que la fórmula principal es la opuesta a (r-b)<0 por lo que la seleccionamos, marcamos la opción *Close* y pulsamos en *Next.*



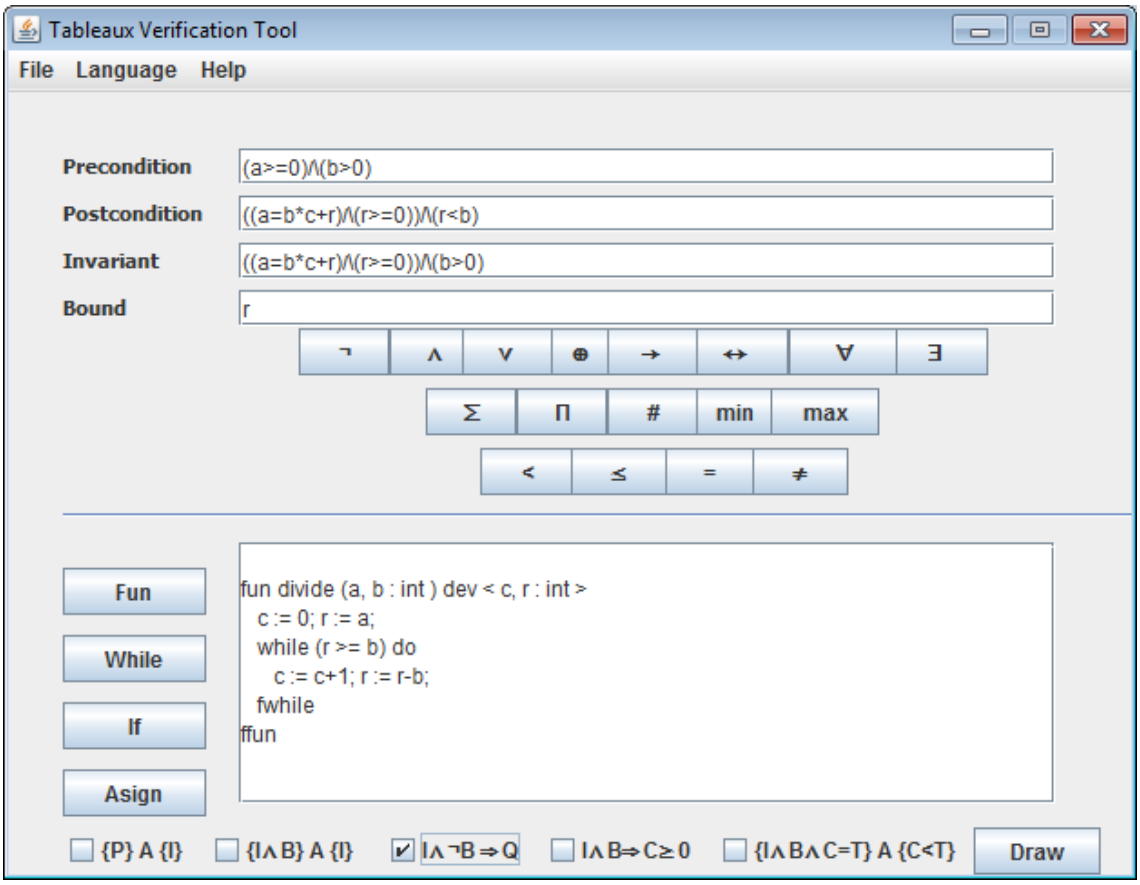
Nos indica que hemos terminado el proceso y a continuación pulsaremos el botón *Results* para ver los resultados obtenidos



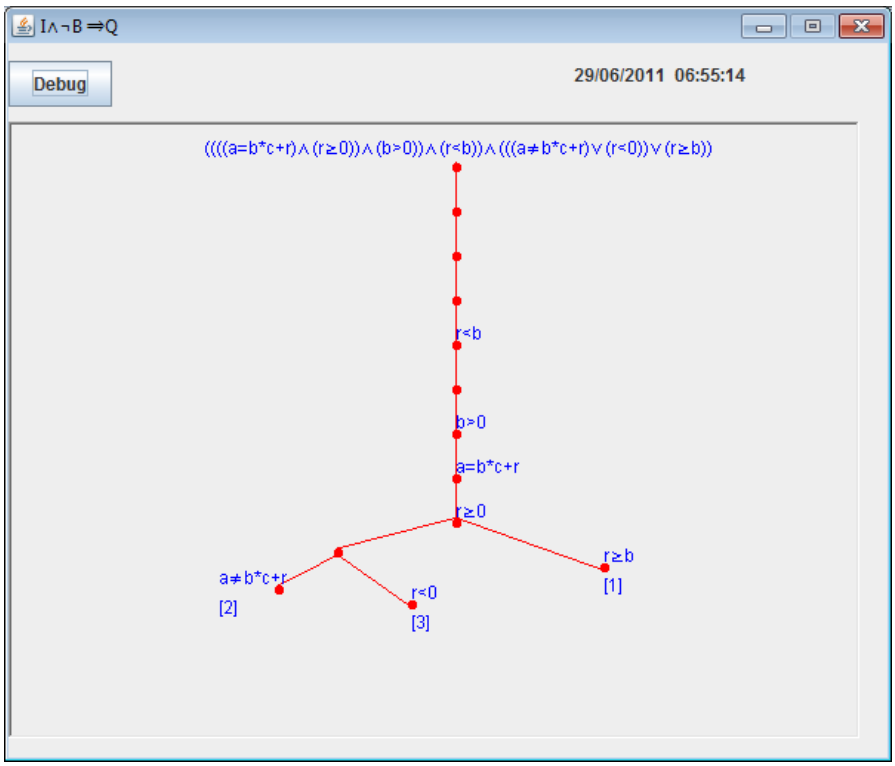
Como se puede observar todas las ramas se han cerrado y las fórmulas con las que se ha hecho.

A continuación, verificaremos la tercera prueba formal:

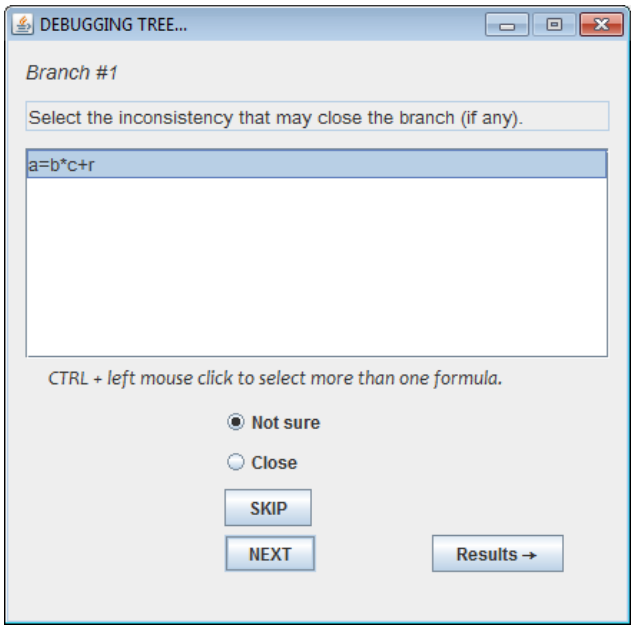
$$I \wedge \neg B \Rightarrow Q$$



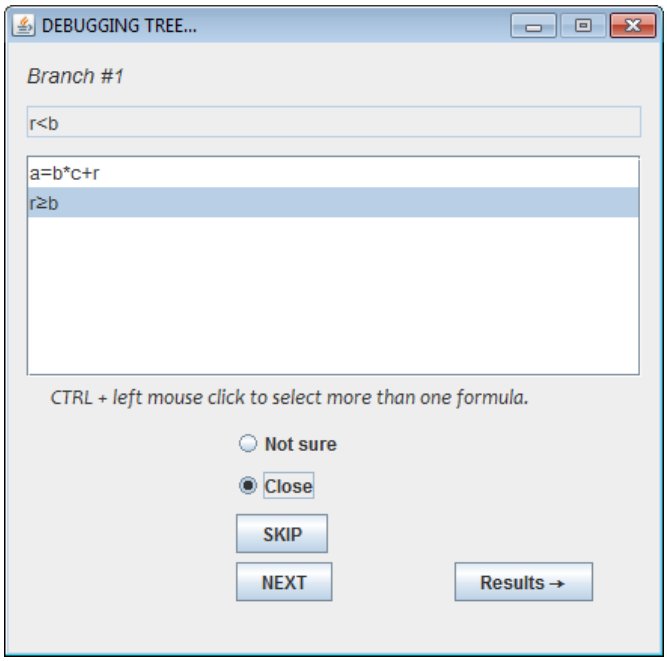
Pulsamos el botón *Draw* para ver el árbol correspondiente a nuestra selección.



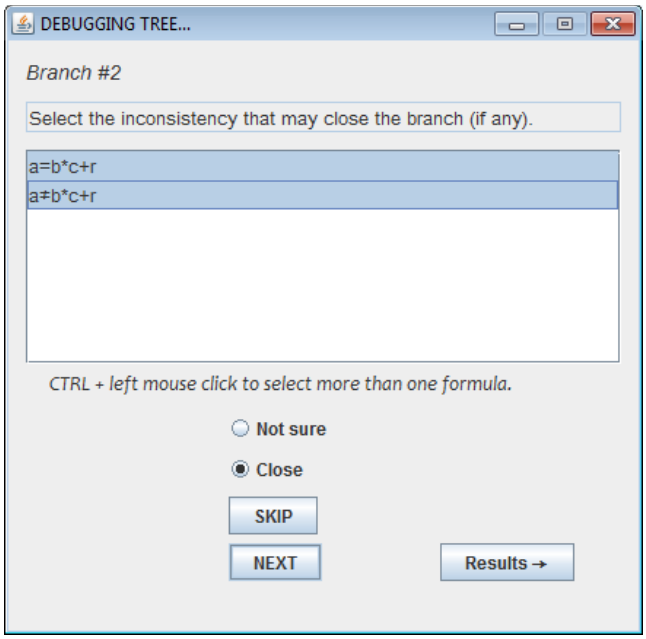
Pulsaremos el botón *Debug* para comenzar con la sesión de cierre de las ramas del árbol.



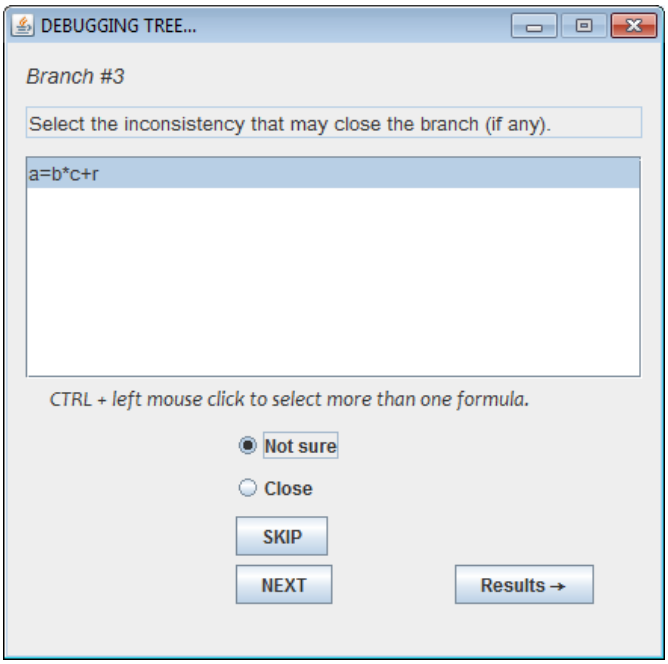
La primera fórmula es una inconsistencia de la que no podemos estar seguros de su corrección, debido a esto la seleccionamos, marcamos la opción *Not Sure* y pulsamos el botón *Next.*



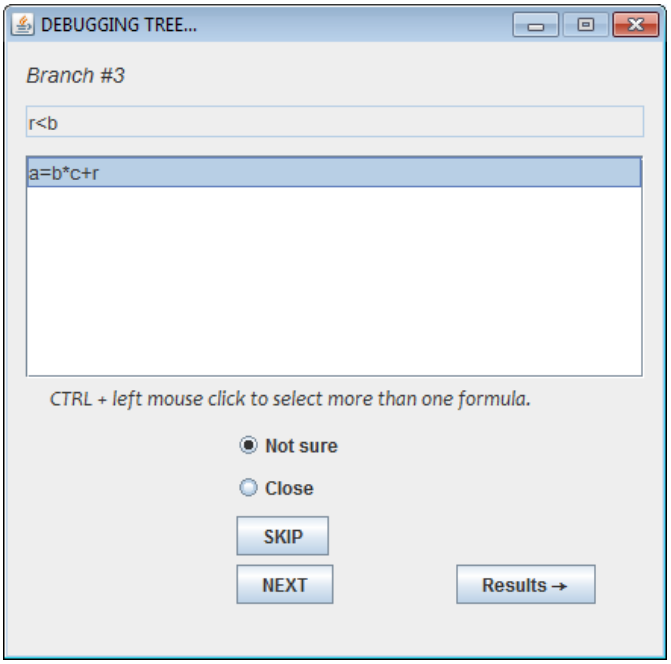
La siguiente formula mostrada se ve que tiene su opuesto en la fórmula candidata r≥b, por lo que la marcamos, seleccionamos *Close* y pulsamos en *Next.*



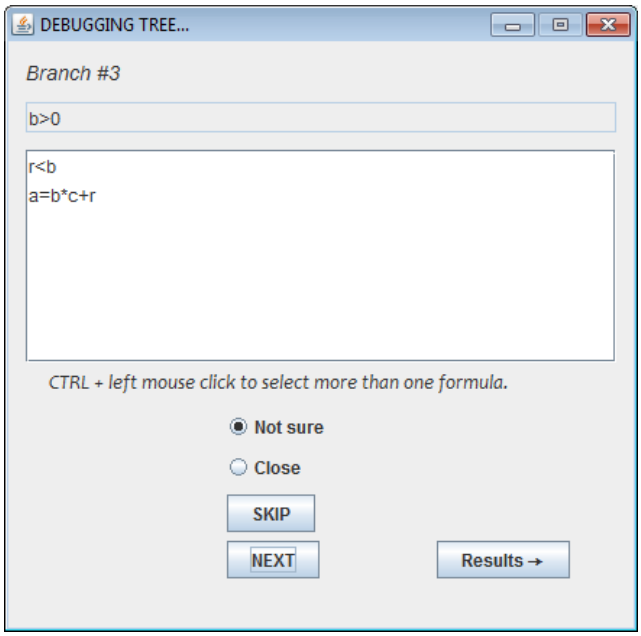
A continuación se nos presentan dos inconsistencias que claramente son opuestas, por lo que marcamos las dos, seleccionamos la opción *Close* y pulsamos en *Next*.



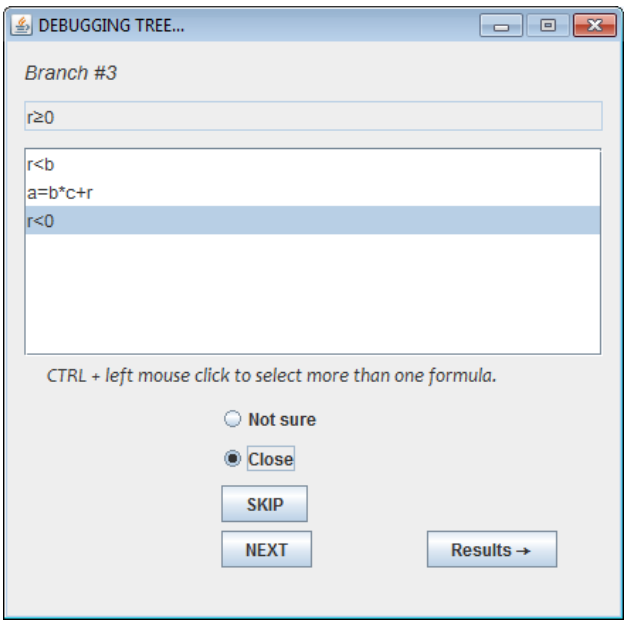
La herramienta de nuevo nos muestra otra inconsistencia de la que no podemos estar seguros de su falsedad, por lo que seleccionamos la opción *Not Sure* y pulsamos en *Next.*



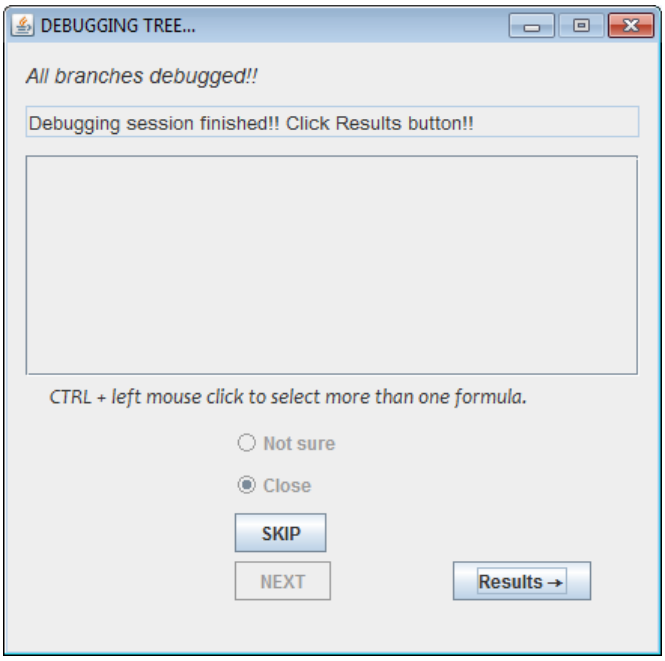
La fórmula candidata mostrada para cerrar la rama no nos da la información necesaria para cerrar la rama, por lo que pulsamos *Not Sure* y el botón *Next.*



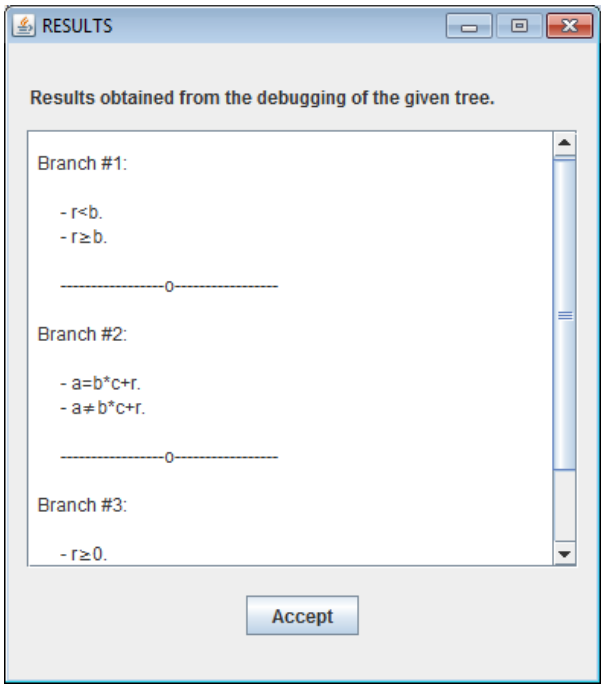
Al igual que en el caso anterior las fórmulas ofrecidas no nos dan toda la información necesaria para cerrar la rama, por lo que volvemos a seleccionar *Not Sure.*



Claramente podemos observar que la fórmula mostrada es la opuesta de la candidata seleccionada, por lo que marcamos la opción *Close* y pulsamos *Next*.



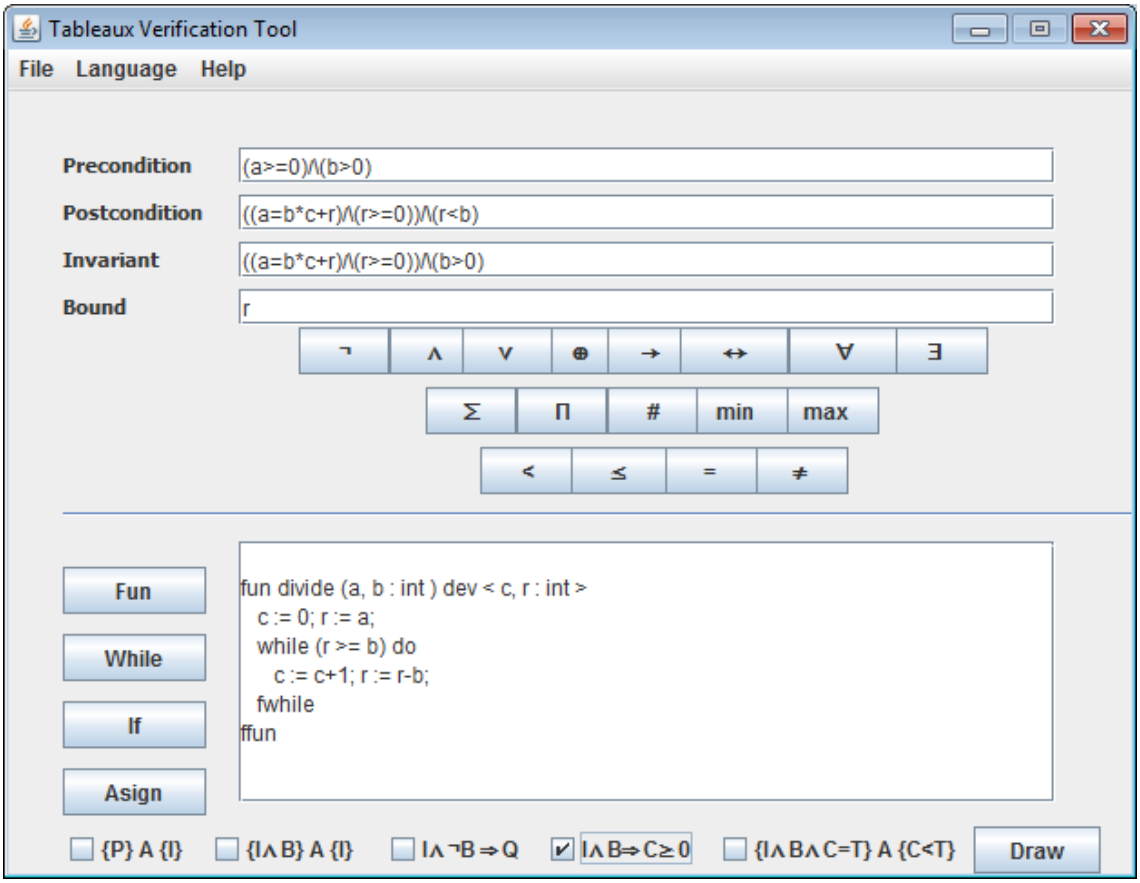
Hemos terminado el proceso y para ver los resultados obtenidos pulsamos en *Results*.



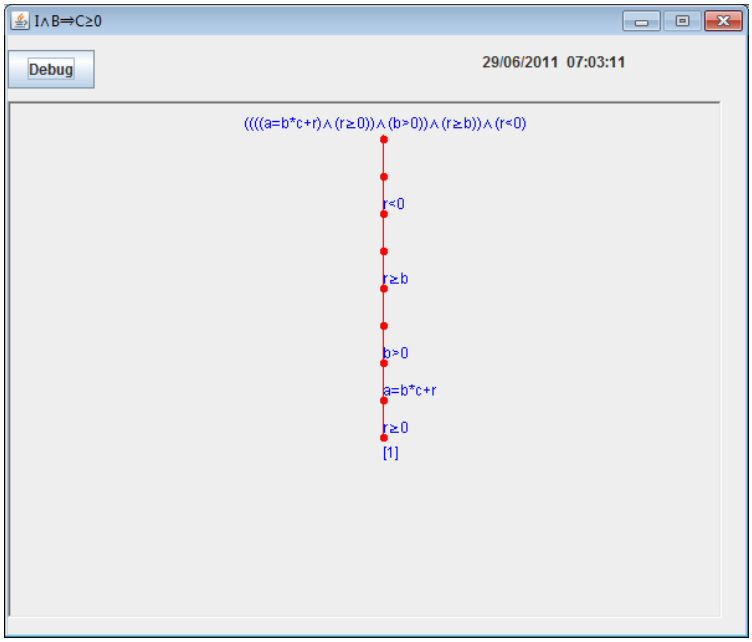
Podemos observar como todas las ramas se han cerrado y las fórmulas involucradas en el cierre de todas las ramas.

A continuación, verificaremos la cuarta prueba formal:

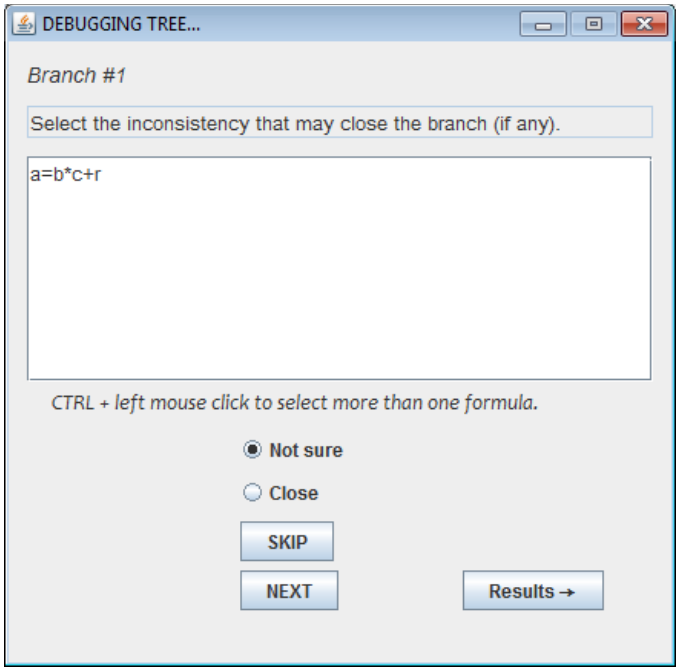
$$I \wedge B \Rightarrow C \geq 0$$



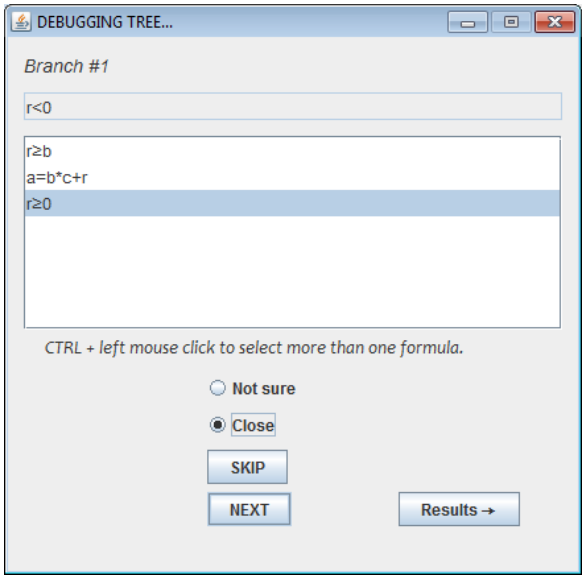
Pulsamos el botón *Draw* para ver el árbol resultante de nuestra elección.



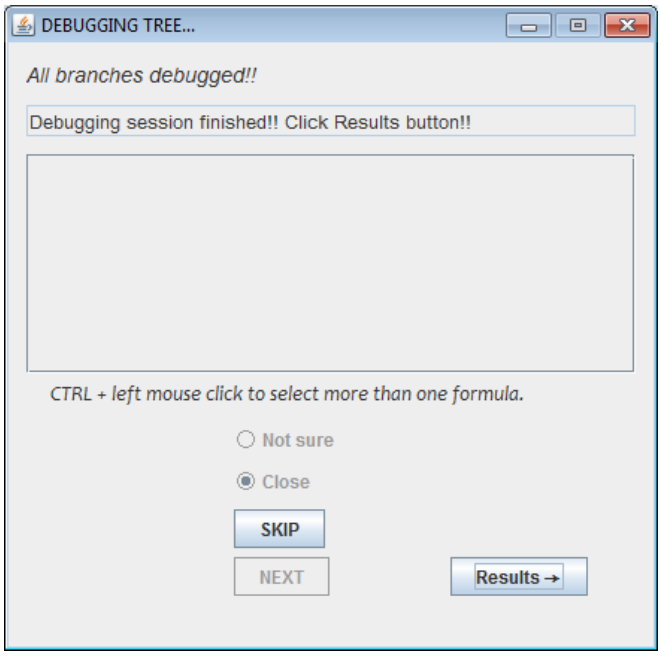
El árbol resultante es muy sencillo y con una única rama. Comenzamos a comprobar su corrección pulsando el botón *Debug*.



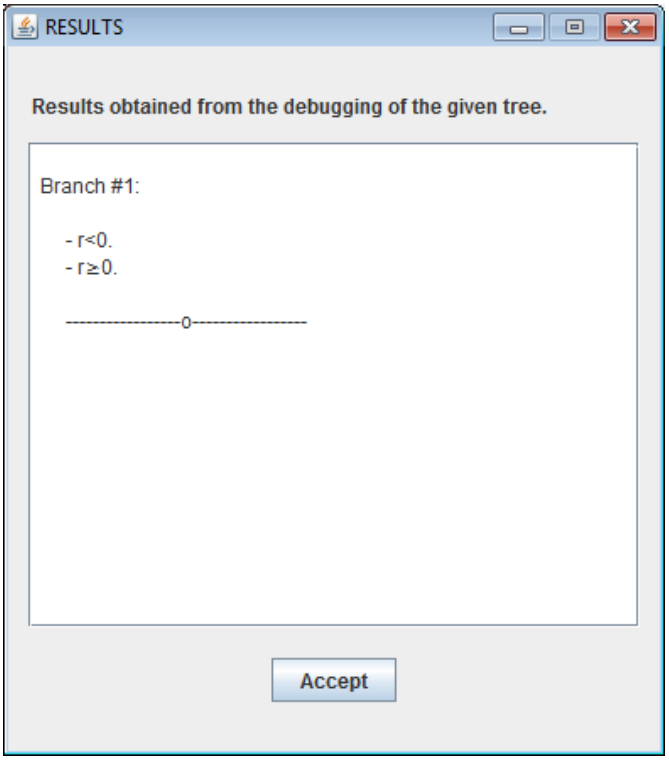
Seleccionamos la inconsistencia, al no estar seguros de su corrección marcamos la opción *Not Sure* y pulsamos en *Next*.



Se ve claramente que la fórmula ofrecida es la opuesta de la candidata r≥0 por lo que marcamos esta, seleccionamos *Close* y pulsamos en *Next.*



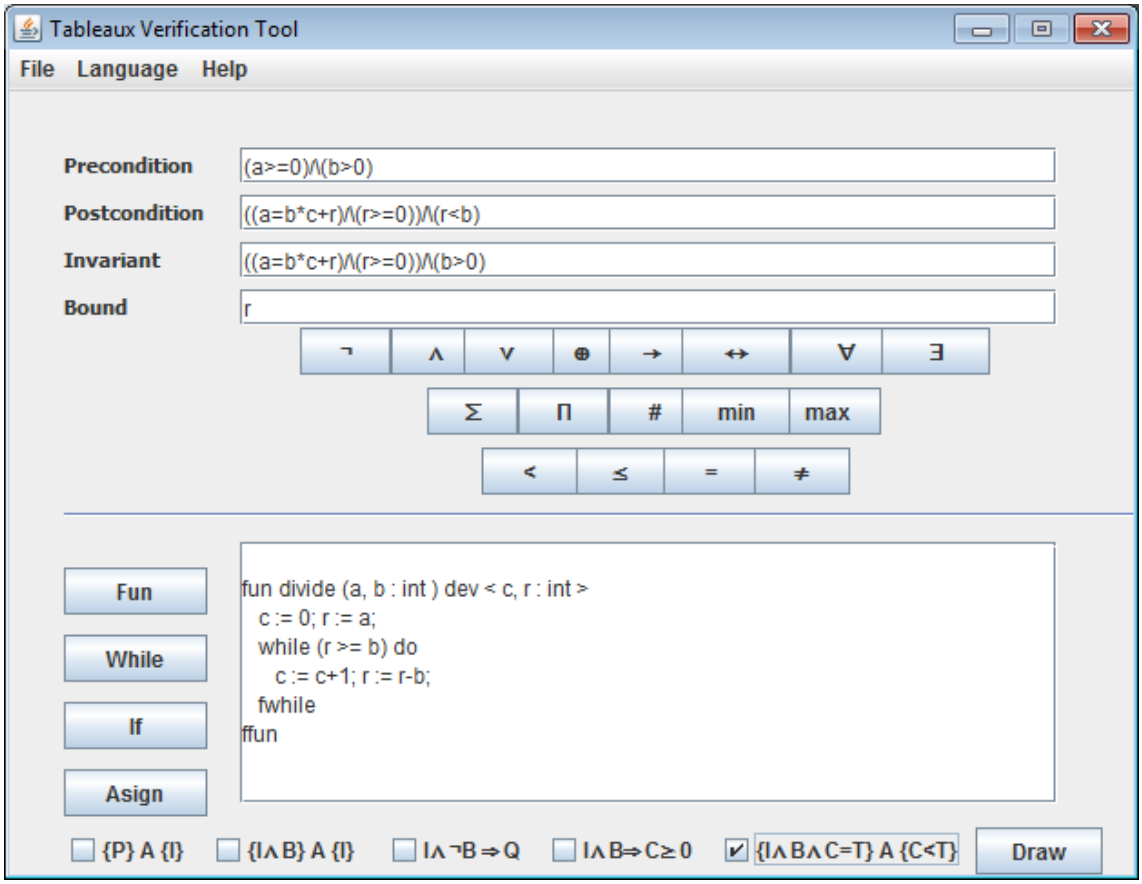
Debido a que este árbol solo tenía una rama ya hemos terminado con la depuración y para ver los resultados pulsaremos en *Results.*



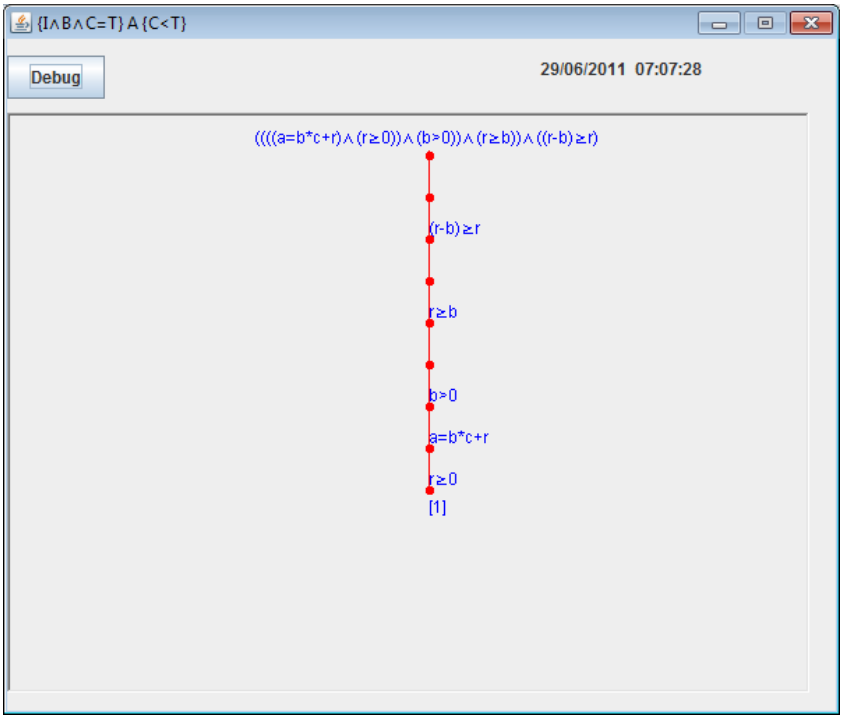
Nos muestra la única rama que formaba el árbol y las fórmulas que la cierran. Hemos verificado su corrección.

Por último, solo falta verificar la prueba formal:

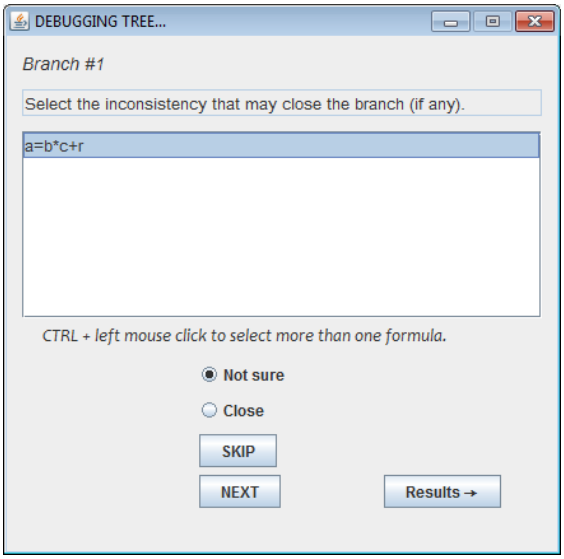
{I^B^C=T} A {C<T}



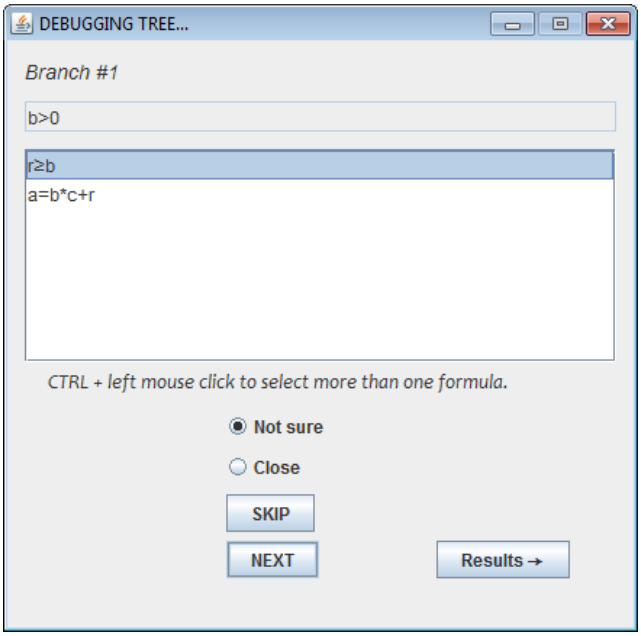
Pulsamos el botón *Draw* para ver el árbol generado.



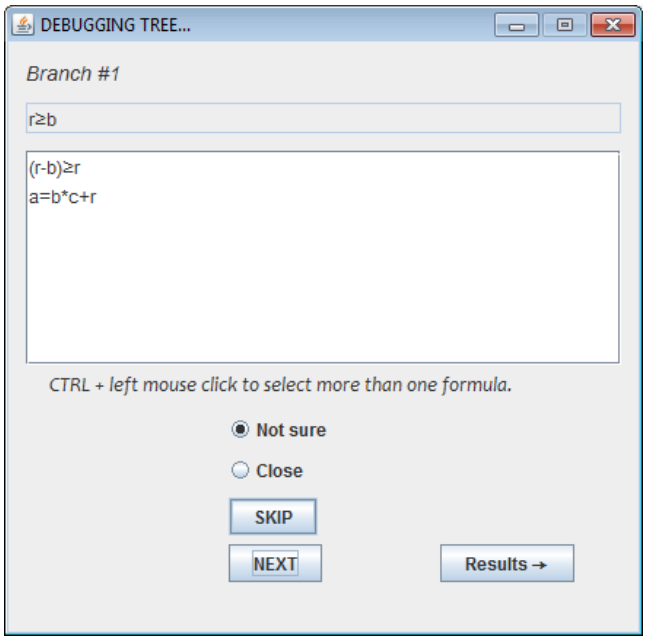
El árbol contiene una única rama . pulsaremos el botón *Debug* para comenzar con su verificación.



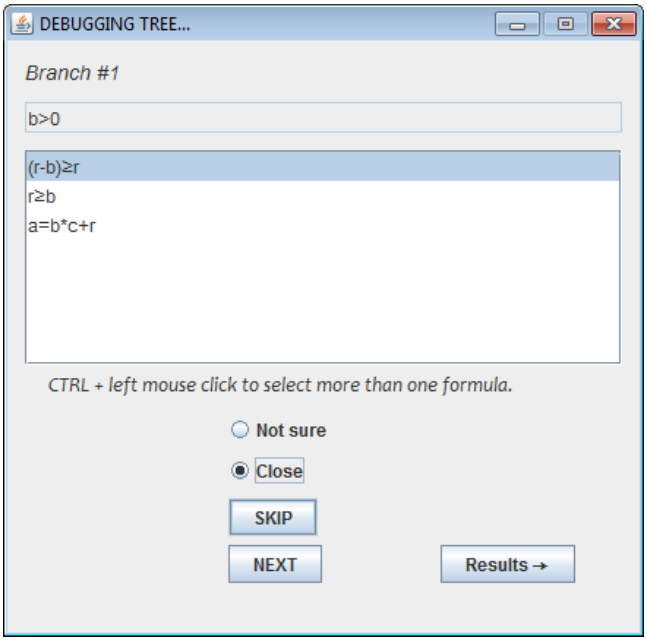
La primera fórmula que se nos muestra es una inconsistencia. Al no estar seguros de llegar a un absurdo con ella, seleccionaremos la opción *Not Sure* y pulsaremos en *Next.*



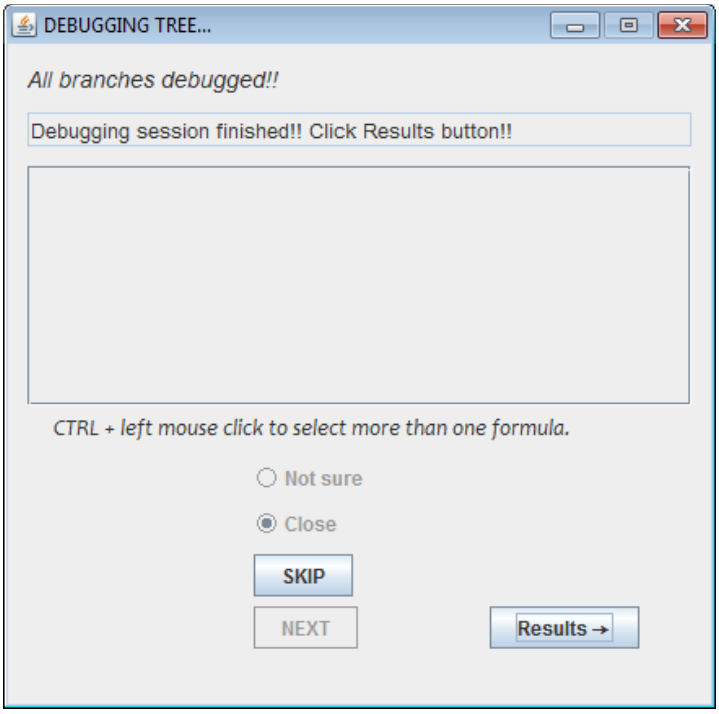
Como no estamos seguros de poder cerrar la rama con las fórmulas ofrecidas mastacamos *Not Sure* de nuevo y pulsamos *Next.*



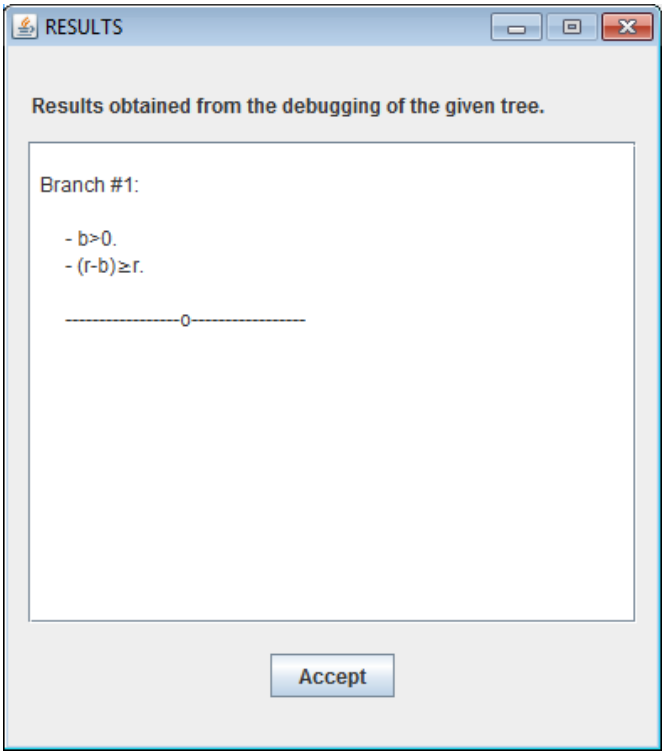
Igual que en el caso anterior no podemos estar seguros de si la rama cerraría o no, por lo que marcamos *Not Sure* y pulsamos en *Next.*



Simplificando la fórmula candidata (r-b)≥r llegamos a la conclusión de que es la opuesta de la ofrecida b>0 por lo que la seleccionamos, marcamos *Close* y pulsamos en *Next.*



Nos informa de que hemos finalizado de recorrer las ramas. Pulsamos el botón *Results* para ver los resultados obtenidos.

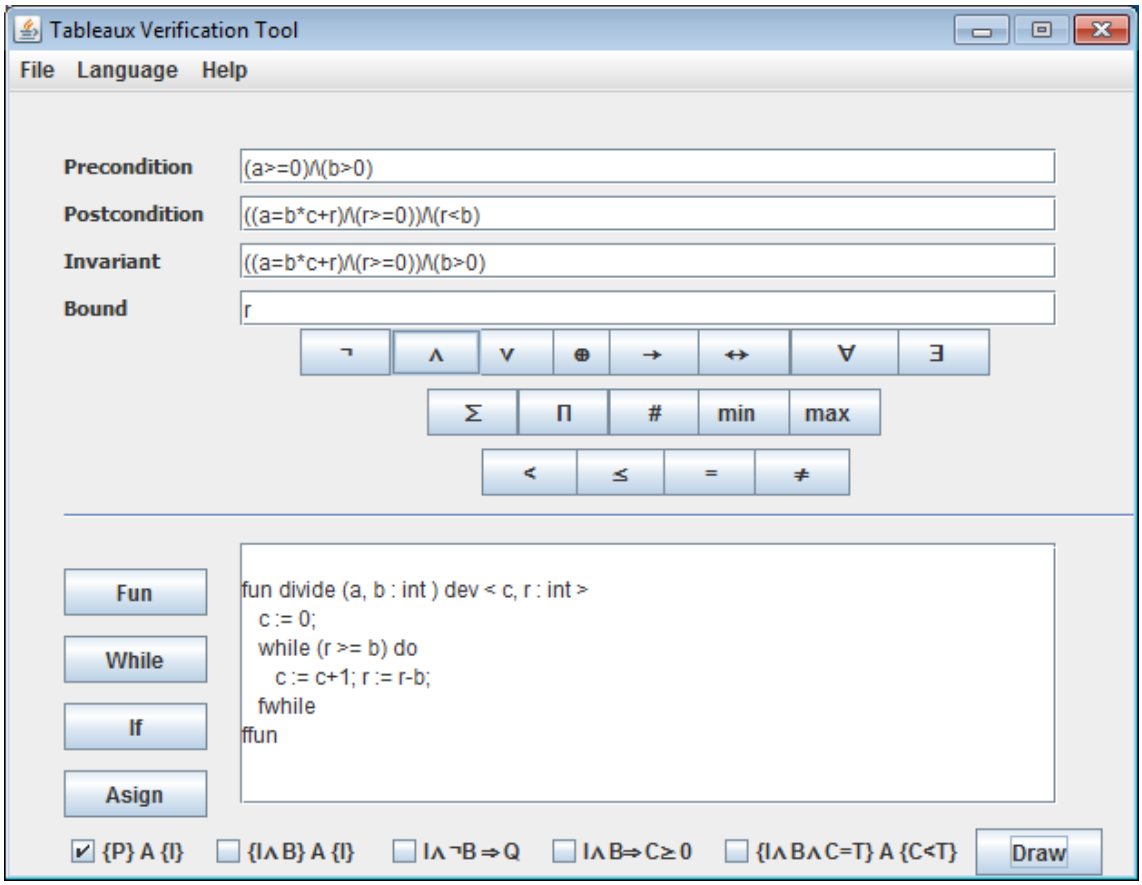


Podemos ver como se cierra la rama con las fórmulas b>0 y (r-b)≥r por lo que esta condición también se ha verificado.

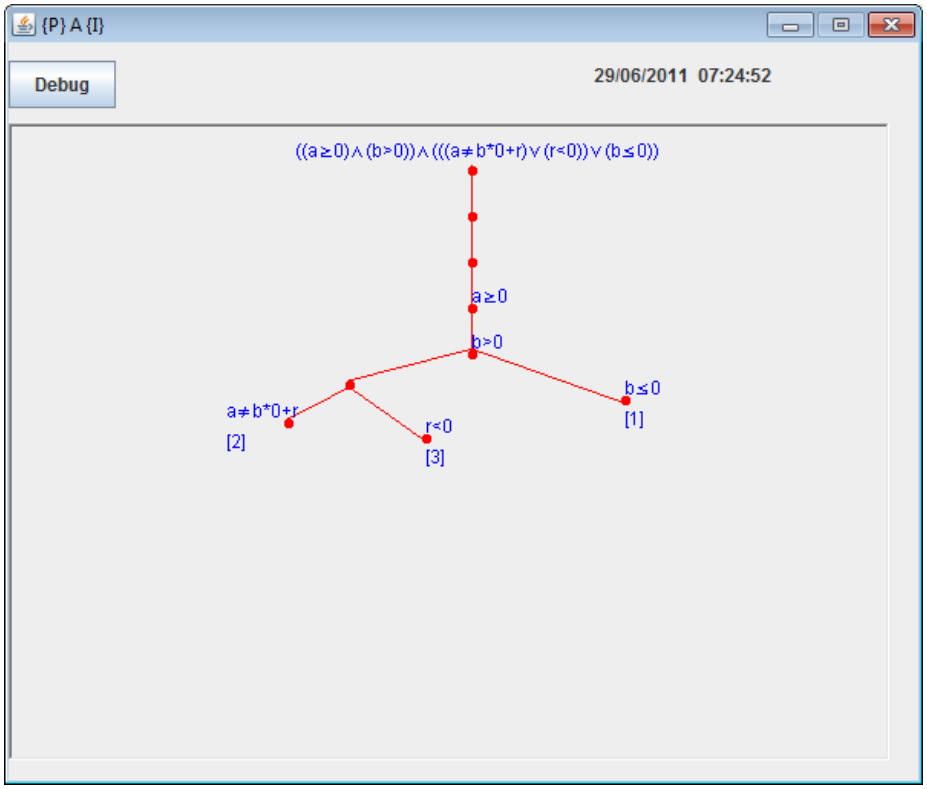
Al haber verificado las cinco pruebas formales, podemos concluir que el algoritmo verificado es correcto.

4.2 - EJEMPLO DE DEPURACIÓN

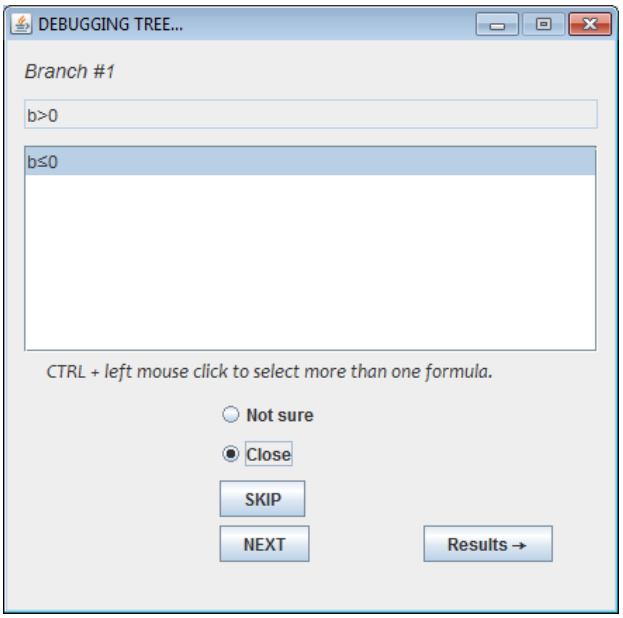
Con nuestra herramienta además de comprobar la corrección de un programa también nos ayuda a encontrar errores en el código en caso de haberlos o a generar código que falte y sea necesario. Por ejemplo probaremos el programa de la división que acabamos de verificar pero eliminando una de sus inicializaciones r:=a y veremos cómo nuestra herramienta nos ayuda a deducir que falta código y cuál deberíamos introducir.



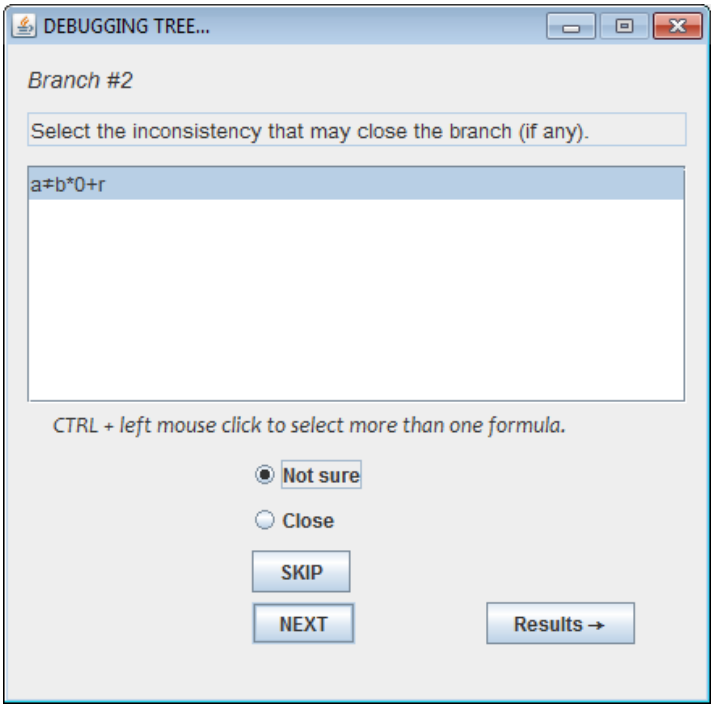
Generamos el árbol correspondiente con el botón *Draw.*



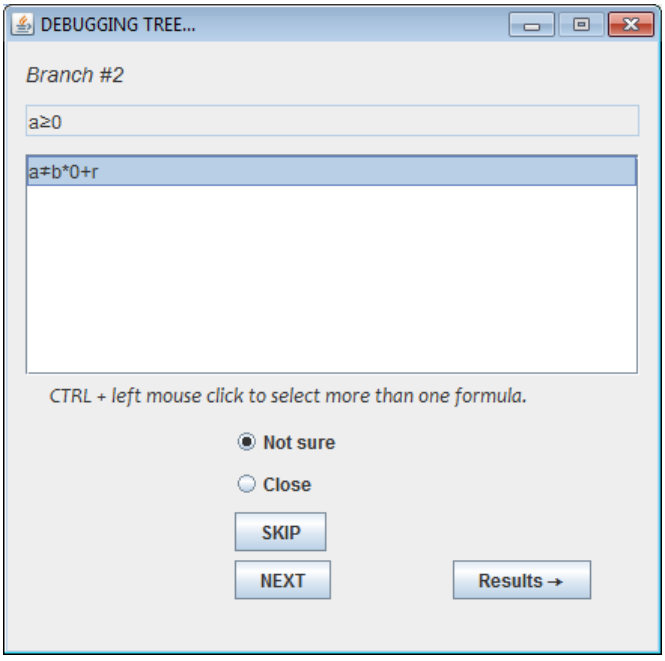
Como se puede observar el árbol ha cambiado respecto al mostrado para esta opción cuando la inicialización estaba completa. Comenzaremos el proceso de depuración pulsando el botón *Debug.*



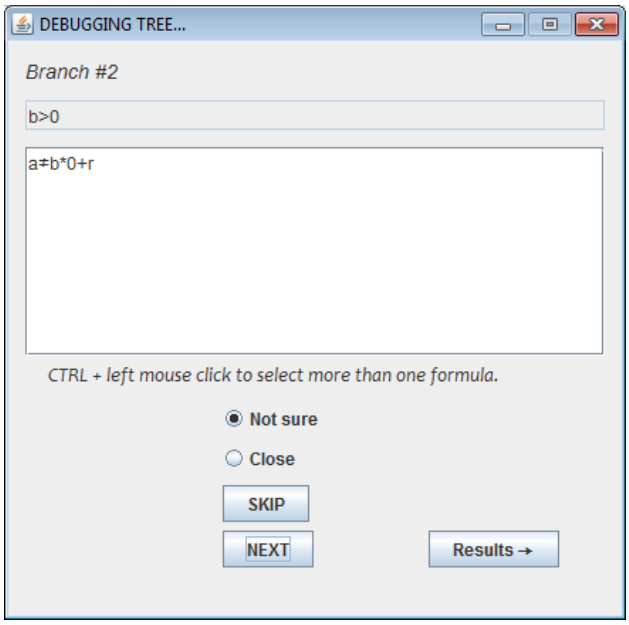
Las dos fórmulas que nos muestra son opuestas por lo que seleccionamos la ofrecida , marcamos *Close* y hacemos click en *Next.*



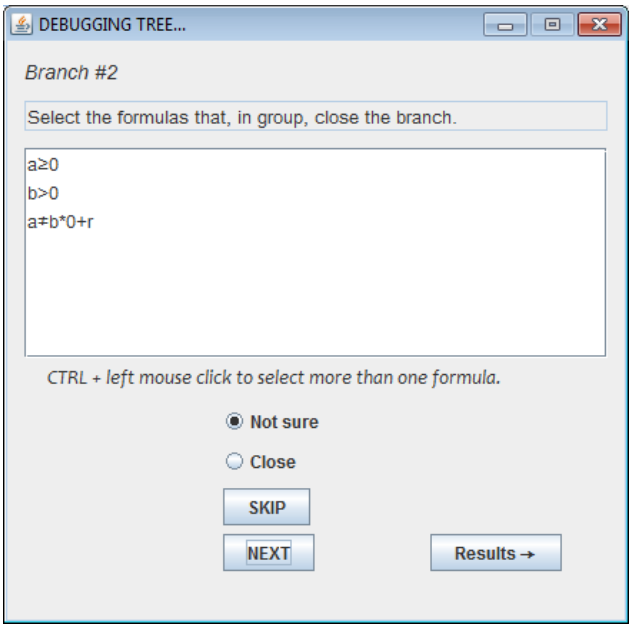
Lo siguiente que nos ofrece es una inconsistencia y como no estamos seguros de poder llegar a un absurdo ya que eso depende del valor de la "r", marcamos *Not Sure* y pulsamos *Next.*



Las fórmulas mostradas tampoco podemos asegurar que sean opuestas, por lo que marcaremos de nuevo la opción *Not Sure* y pulsaremos en *Next.*

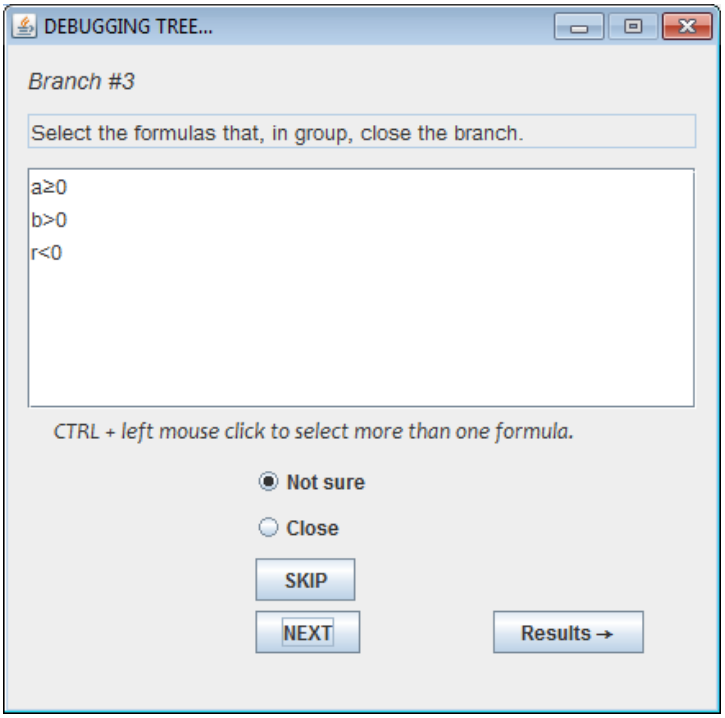


Las fórmulas ofrecidas tampoco nos aportan la información necesaria para cerrar la rama por lo que volveremos a marcar *Not Sure* y pulsaremos de nuevo *Next.*

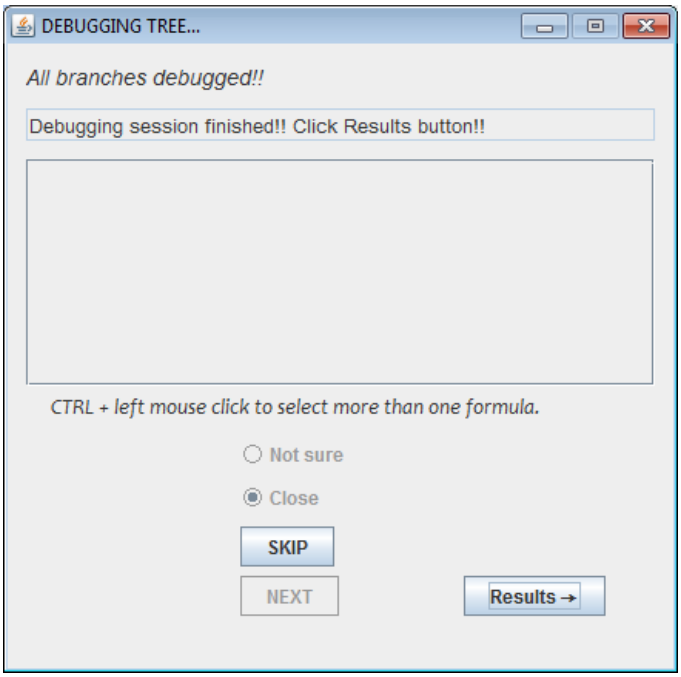


Aunque seleccionemos varias de las fórmulas ofrecidas, no podremos cerrar la rama ya que necesitamos algo que nos dé

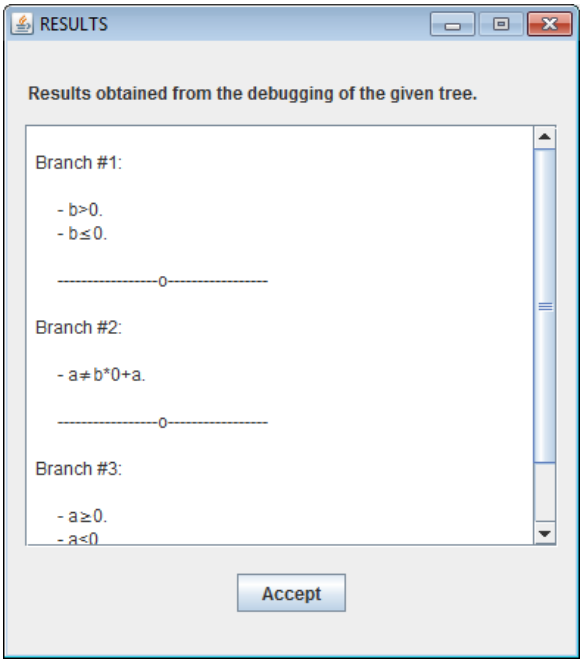
información sobre cómo se relacionan “a” y “ r”, por lo que volvemos a marcar *Not Sure* y hacemos click en *Next*.



Las nuevas fórmulas que nos ofrece tampoco nos ayudan a cerrar la rama puesto que ninguna es la opuesta de la otra, por lo que volvemos a marcar *Not Sure* y pulsamos el botón *Next*.

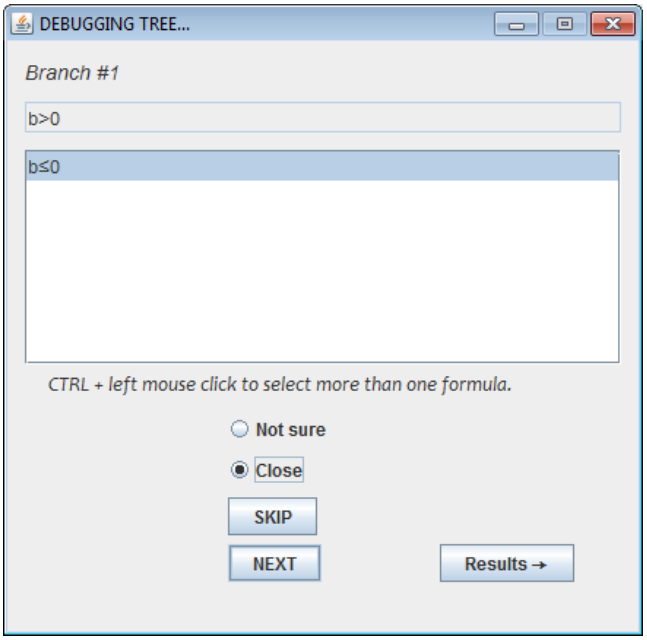


Nos indica que ya hemos terminado de probar todas las fórmulas, y para ver los resultados pulsamos en *Results.*

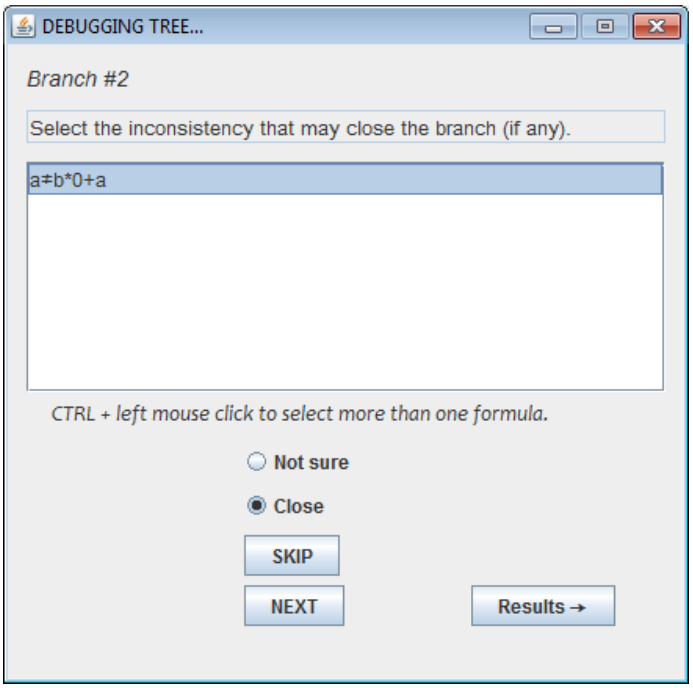


Como podemos ver las ramas 2 y 3 se han quedado abiertas, lo que nos indica que o bien hay algún fallo en el código, o bien falta algo en él, además por el paso de verificación en el que estamos {P}A{I} sabemos que este fallo estará en las inicializaciones y como nos dimos cuenta anteriormente una opción a tener en cuenta será relacionar de alguna manera "r" y "a". Con toda esta información ejecutamos de nuevo la herramienta, poniendo especial cuidado en la depuración de las ramas que ahora se nos han quedado abiertas, y teniendo en cuenta las conclusiones anteriores.

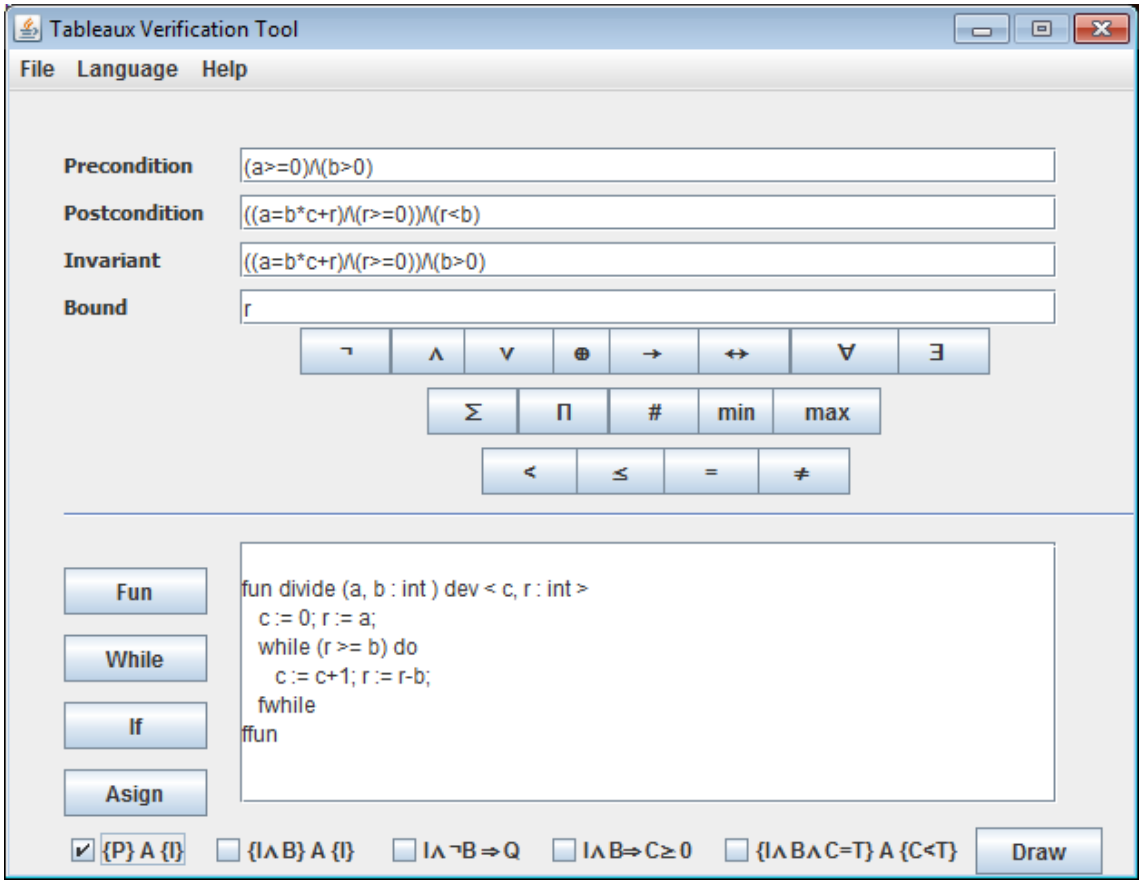
Iniciamos de nuevo el proceso de depuración.



Como ya habíamos hecho antes, cerramos esta rama puesto que las dos fórmulas ofrecidas son opuestas, por lo que marcamos la opción *Close* y pulsamos en *Next*.



Nos aparece de nuevo la inconsistencia, y nos fijamos en ella sabemos que para cerrar la rama necesitamos relacionar de alguna manera “r” y “a ” y la única forma de que la inconsistencia se haga falsa es si r=a para que al simplificar la fórmula quedase a≠a , que es falso y podríamos cerrar la rama, por lo que nos damos cuenta de que en la inicialización falta esa asignación: r:=a . Para añadirla al código, pulsamos en *Skip* y modificamos el código.



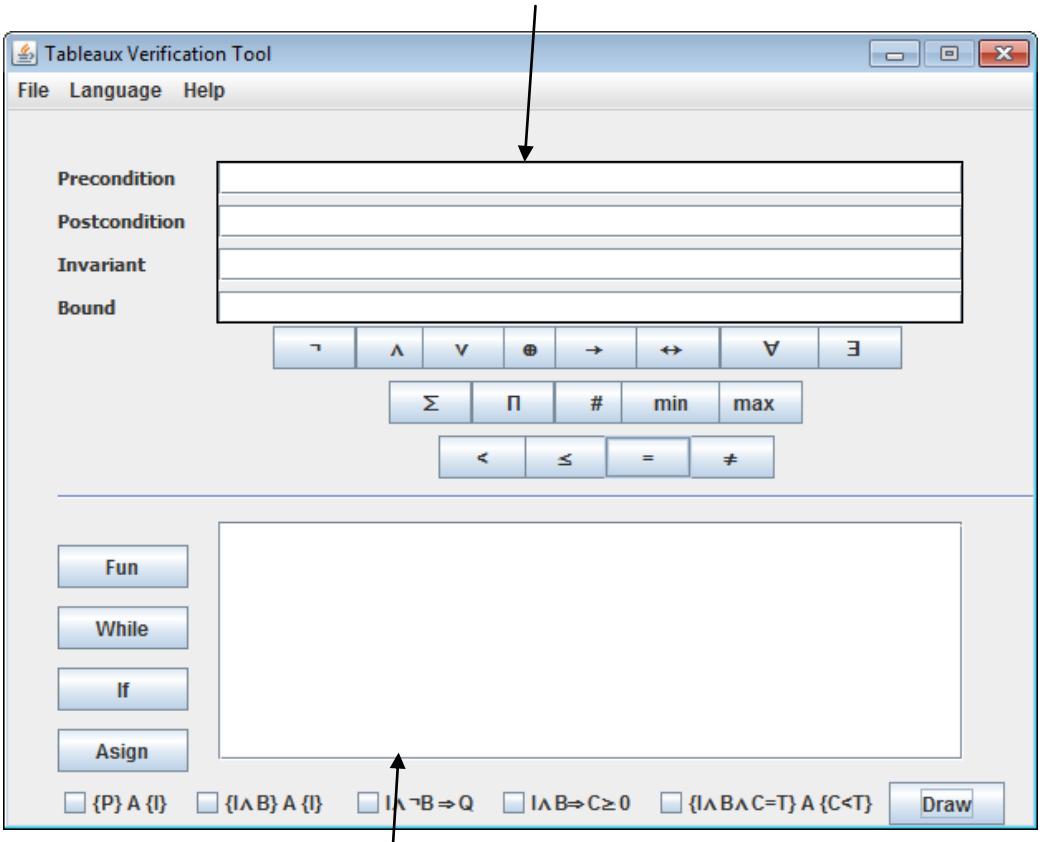
Si volvemos a generar el árbol y hacemos el proceso de depuración veremos como con el nuevo código insertado se cierran todas las ramas por lo que el código, tras la inserción, es correcto. Nuestra herramienta nos ha ayudado a generar el código que faltaba.

CAPÍTULO 5: MANUAL DE USUARIO

La siguiente guía de usuario mostrará las funcionalidades de la interfaz así como un ejemplo guiado, facilitando así sucesivos usos de la herramienta TVT.

5.1 - INTERFAZ

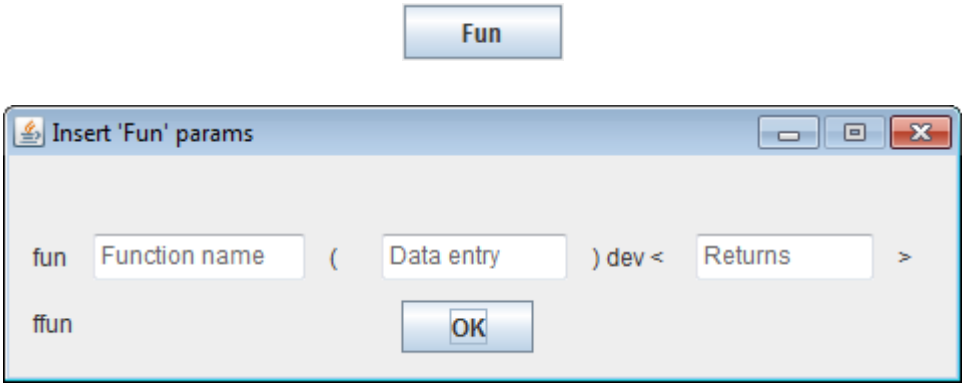
Campos para introducir la precondición, postcondición, invariante y función de cota del algoritmo a tratar. Las expresiones deberán ir completamente parentizadas. Dichos paréntesis separarán las fórmulas entre sí y se utilizarán para dar prioridad a unas operaciones sobre otras como se muestra más adelante en el ejemplo. Para facilitar la escritura se han añadido una serie de botones que comentaremos a continuación.



Campo en el que se escribirá el algoritmo que deseamos verificar o depurar. Para facilitar su escritura se han añadido una serie de botones que nos servirán de guía para introducir el código.

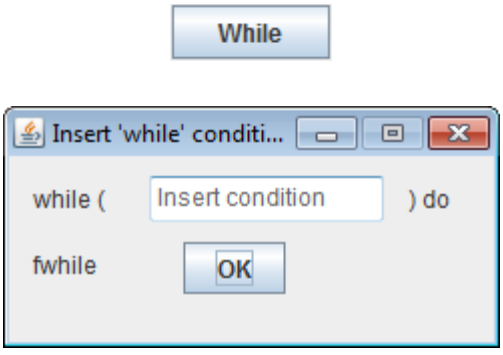
5.2 – BOTONES PARA LA EDICIÓN DE CÓDIGO

5.2.1 FUN:



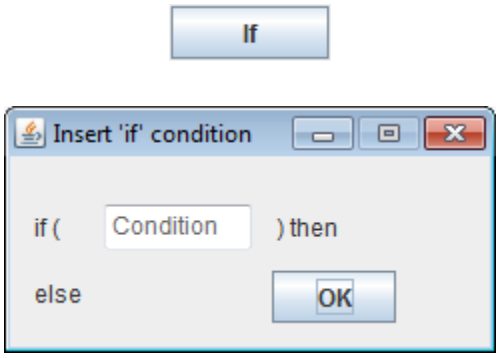
Como los propios campos indican en *"Function name"*, deberemos escribir el nombre que tendrá la función que deseamos tratar; en *"Data entry"*, deberemos introducir las variables que utilizará la función, con formato: "nombreVar : tipo". En caso de disponer de varias variables deberemos separarlas por comas. En el campo "*Returns"*, se escribirá la variable o variables de salida, con formato : "nombreVar : tipo". Análogamente a *"Data entry"*, en caso de escribir varias variables, deberemos separarlas por comas.

5.2.2 WHILE:



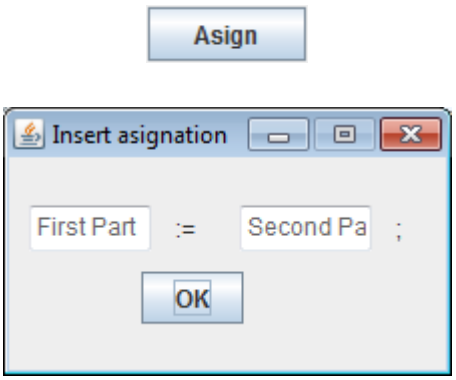
En el campo "*Insert* C*ondition*", se deberá insertar la condición de continuidad del bucle, que será de tipo booleano. En el caso de que esa condición se haga falsa no continuará la ejecución del bucle.

5.2.3 IF



En el campo "*Condition",* se escribirá una expresión booleana. En caso de ser cierta, el usuario deberá escribir tras el "then" la instrucción que quiera ejecutar. Si, por el contrario, la condición se evalúa a falso, el usuario deberá escribir la instrucción que quiere que se ejecute entonces. Si no desea ejecutar ninguna instrucción en caso de que la condición se evalúe a falso, deberá eliminar la palabra "else" del cuerpo del bucle.

5.2.4 ASIGN



Insertará una asignación. En el campo "*First Part*", el usuario deberá introducir la variable a la que desea asignar un valor. En el campo "*Second Part*", se deberá introducir el valor que deseamos darle a la primera variable, pudiendo ser este un número, una expresión matemática u otra variable.

Todas las expresiones anteriores se insertarán en el campo que contiene el cuerpo del bucle, en la posición en la que se encuentre el cursor en ese momento.

5.3 - BOTONES PARA LA PRECONDICIÓN, POSTCONDICIÓN, INVARIANTE Y COTA

Cada uno de estos botones introducirá el carácter o caracteres correspondientes en la posición en la que se encuentre el cursor en el momento de su activación.

Introduce la negación.

Introduce el símbolo AND, debe ser introducido entre dos variables o expresiones.

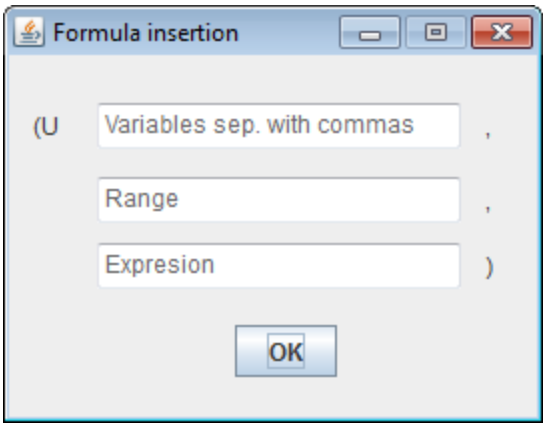
Introduce el símbolo OR, análogamente a lo que ocurría con la and debe ser introducido entre dos variables o expresiones.

Introduce el símbolo XOR, entre dos variables o expresiones.

Implicación.

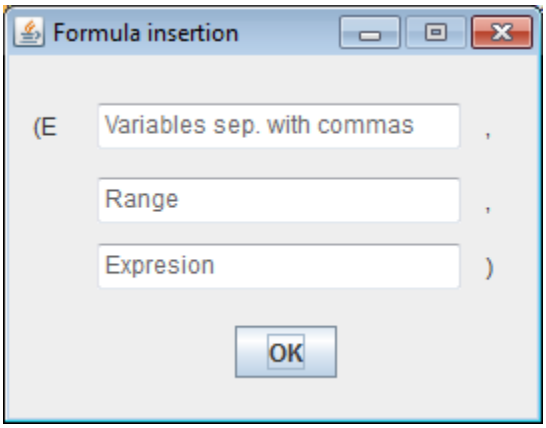
Doble implicación.

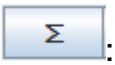
Universal ∀: Tras pulsarlo aparecerá el siguiente formulario:



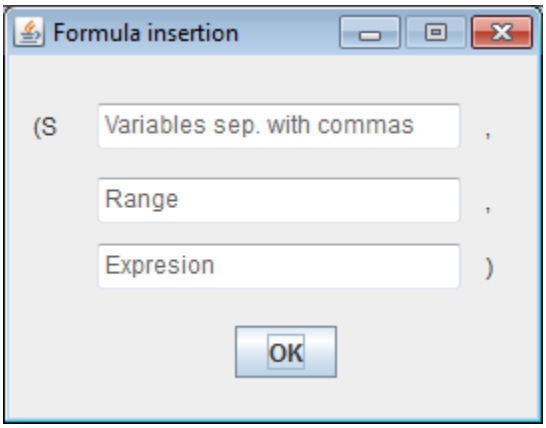
En el primer campo, el usuario introducirá las variables ligadas del universal, separadas por comas. En el segundo campo, deberá introducir el rango entre el que estarán estas variables, pudiendo ser de la forma: $x_1<x_2<x_3<\ldots<x_n$. En el tercer campo, introduciremos la propiedad que queremos que se cumpla para el cuantificador universal, pudiendo haber propiedades anidadas, como otro cuantificador.

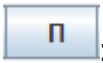
Existencial ∃: Análogo al cuantificador universal. Los campos se rellenarán de igual forma.



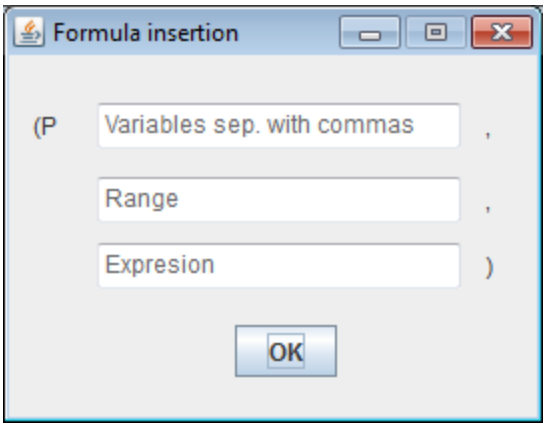
Sumatorio 

: Análogo al cuantificador universal. Los campos se rellenarán de igual forma.



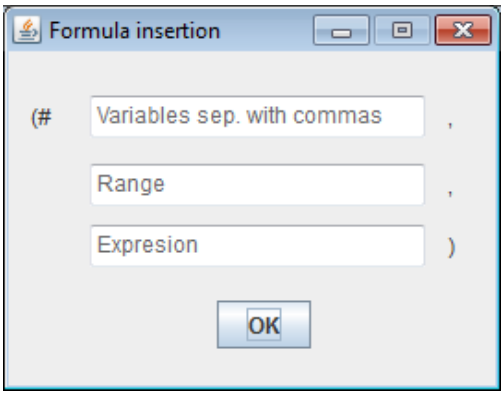
Producto 

: Análogo al cuantificador universal. Los campos se rellenarán de igual forma.



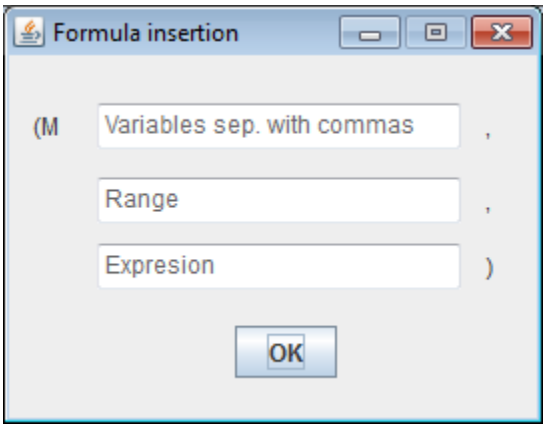
Contador 

: Análogo al cuantificador universal. Los campos se rellenarán de igual forma.



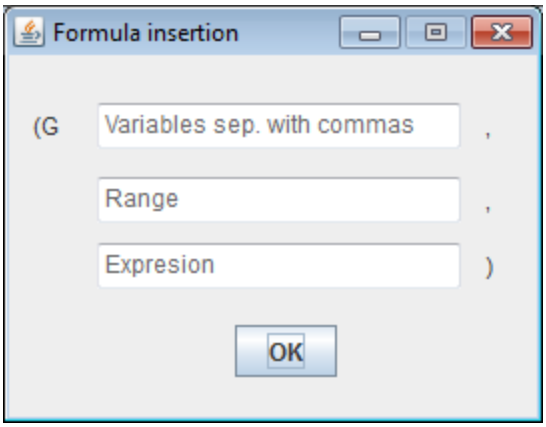
Mínimo 

: Análogo al cuantificador universal. Los campos se rellenarán de igual forma.



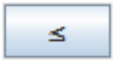
Máximo 

: Análogo al cuantificador universal. Los campos se rellenarán de igual forma.

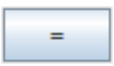


Menor 

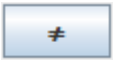
: introduce el símbolo menor en el campo donde se halle el cursor.

Menor o igual 

: introduce el símbolo menor o igual.

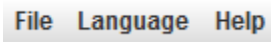
Igual 

: introduce el símbolo igual.

Símbolo de distinto 

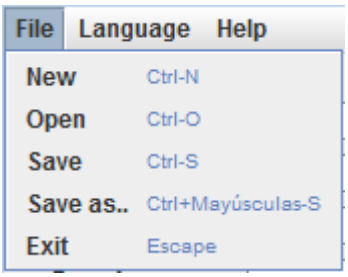
: introduce el símbolo distinto.

5.4 - BARRA DE HERRAMIENTAS



Encontramos los desplegables:

5.4.1 FILE

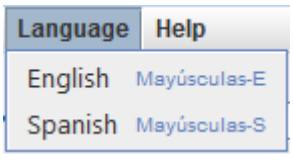


Contiene las siguientes opciones:

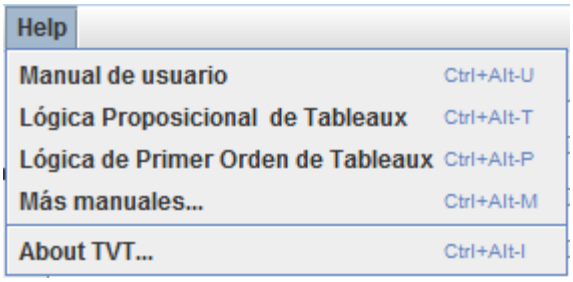
- ***New***: Limpia los campos del formulario para que el usuario pueda reescribir su algoritmo.
- ***Open***: Permite cargar un algoritmo guardado en formato ".txt".
- ***Save***: Permite guardar un algoritmo que hemos cargado previamente.
- ***Save as***: Permite guardar un algoritmo que el usuario ha escrito con extensión ".txt"
- ***Exit:*** Cierra la aplicación.

5.4.2 LANGUAGE

Nos permitirá seleccionar el idioma en el que deseamos trabajar, español o inglés.



5.4.3 HELP



Varias opciones:

- ***Manual de Usuario***: Abre este manual de usuario.
- ***Lógica Proposicional de Tableaux:*** Abre un sencillo manual sobre las principales fórmulas lógicas proposicionales empleadas en los Tableaux.
- ***Lógica de Primer Orden de Tableaux***: Abre un sencillo manual sobre las principales fórmulas lógicas de primer orden empleadas en los Tableaux.
- ***Más manuales…*** **:** Abre la carpeta donde están contenidos todos los manuales.
- ***About TVT…:*** Información de la herramienta, autores, versión …

5.5 - CHECK BOX

Para verificar la corrección de un algoritmo iterativo, el invariante deberá cumplir una serie de condiciones, cada check box permite al usuario comprobar la corrección de una de estas condiciones.

☐ $\{P\}\,A\,\{I\}$ El invariante se cumple antes de la primera iteración del bucle.

☐ $\{I \wedge B\}\,A\,\{I\}$ Mientras se ejecuta el cuerpo del bucle A, el invariante se mantiene.

☐ $I \wedge \neg B \Rightarrow Q$ El invariante se sigue cumpliendo al salir del bucle.

☐ $I \wedge B \Rightarrow C \geq 0$ C será una función dependiente de las variables del bucle que garantice que éste termina. En este caso, mientras se cumpla la condición B, C será mayor que cero.

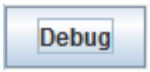
☐ $\{I \wedge B \wedge C=T\}\,A\,\{C<T\}$ C decrecerá al ejecutarse el cuerpo del bucle.

5.6 - BOTON DRAW

Tras rellenar todos los campos del formulario al presionar este botón se dibujará el árbol correspondiente a la opción seleccionada en el check box para comprobar posteriormente su corrección. El usuario puede dibujar otro árbol sin necesidad de cerrar los anteriormente dibujados.

En el nuevo formulario que se genera aparecerá, además del árbol, la fecha y la hora del momento de la generación, como un nuevo botón *Debug*, que servirá al usuario para entrar en modo depuración e ir cerrando ramas del árbol para comprobar la corrección de su algoritmo.

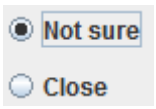
5.7 - BOTON DEBUG



Se creará un nuevo formulario para la depuración de las ramas. Nos aparecerá en negrita la formula a analizar y a continuación en un cuadro de texto las diferentes opciones que el usuario puede seleccionar para cerrar esa rama. Para seleccionar una opción se deberá hacer click en ella. En caso de querer seleccionar varias, podrá hacerse pulsando el botón control de manera simultánea al botón izquierdo del ratón.

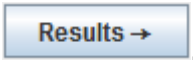
Si el usuario está seguro de la corrección de su elección marcará la opción *Close* y pulsará el botón *Next* 

. En caso de dudar, podrá seleccionar la opción *Not Sure* y a continuación podrá pulsar el botón *Next*. Tras pulsarlo aparecerá la siguiente fórmula a analizar.



En caso de querer detener la depuración o si se desea modificar el código a depurar el usuario deberá pulsar el botón *Skip* 

.

En cualquier momento el usuario podrá pulsar el botón *Results* 

, donde se le mostrará en una nueva ventana los resultados obtenidos hasta el momento, indicando las ramas que no están cerradas. Tras depurar todas las ramas, en la pantalla de depuración el usuario podrá leer un mensaje en el que se le informará de que todas las ramas han sido depuradas.

En caso de haber podido cerrar todas las ramas, el algoritmo analizado será correcto.

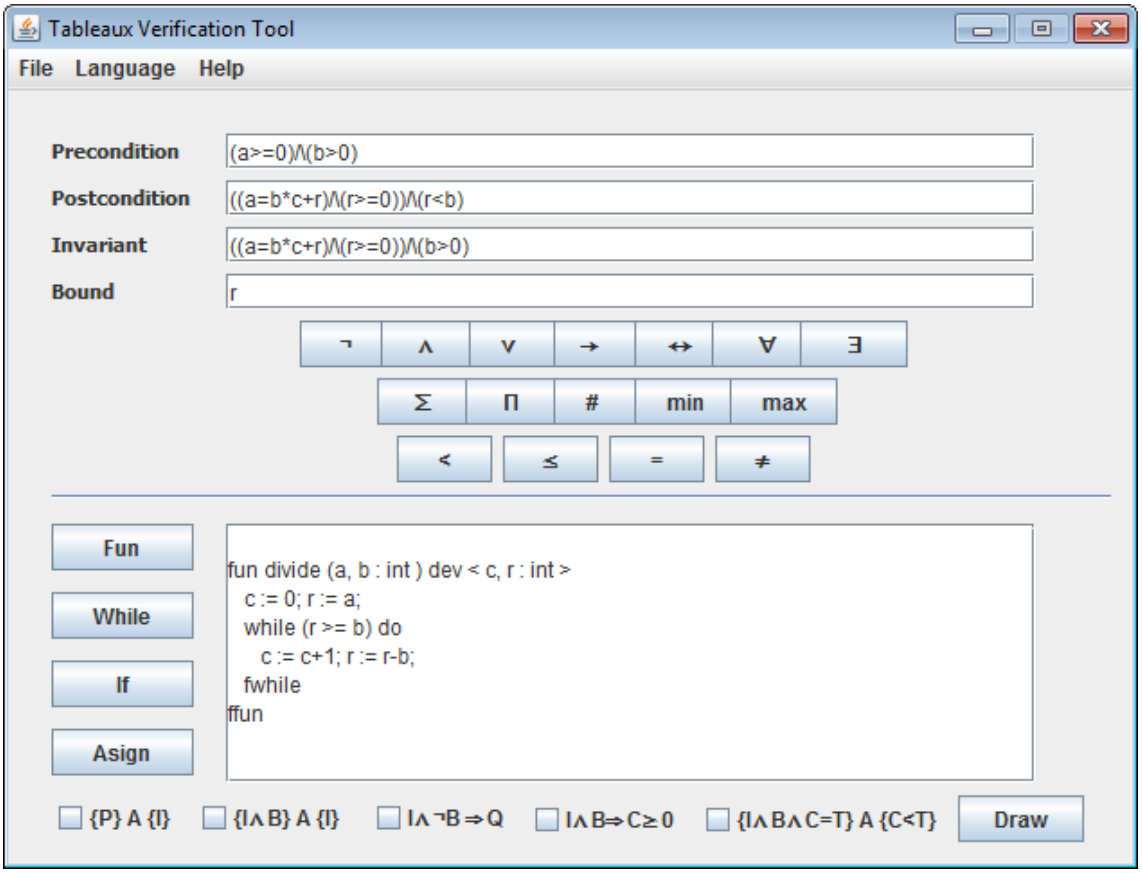
5.8 - EJEMPLO

Deseamos probar la corrección del siguiente algoritmo:

Pre{P :(a>=0)/\(b>0)}
Pos{Q :((a=b*c+r)/\(r>=0))/\(r<b)}
Inv{I :((a=b*c+r)/\(r>=0))/\(b>0)}
Cot{C :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
```

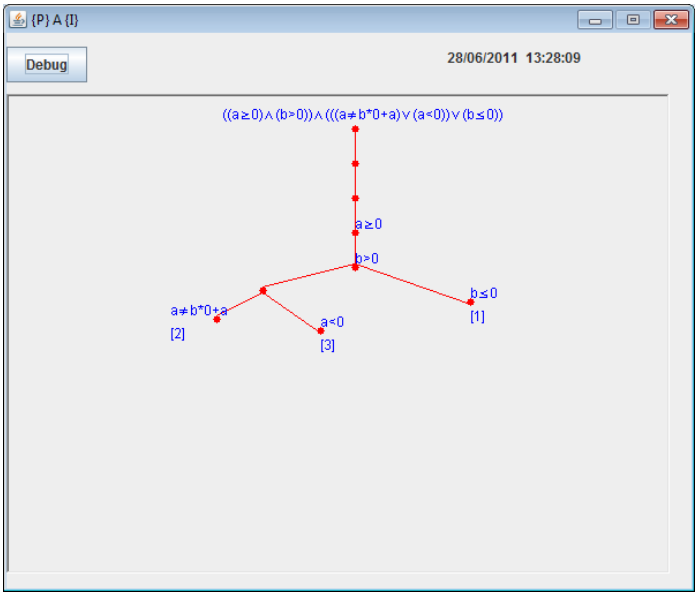
Para comenzar a utilizar nuestra herramienta podemos guardar el código con formato .txt y cargarlo mediante la opción *Load* de la barra de herramientas o rellenar cada uno de los campos del formulario:



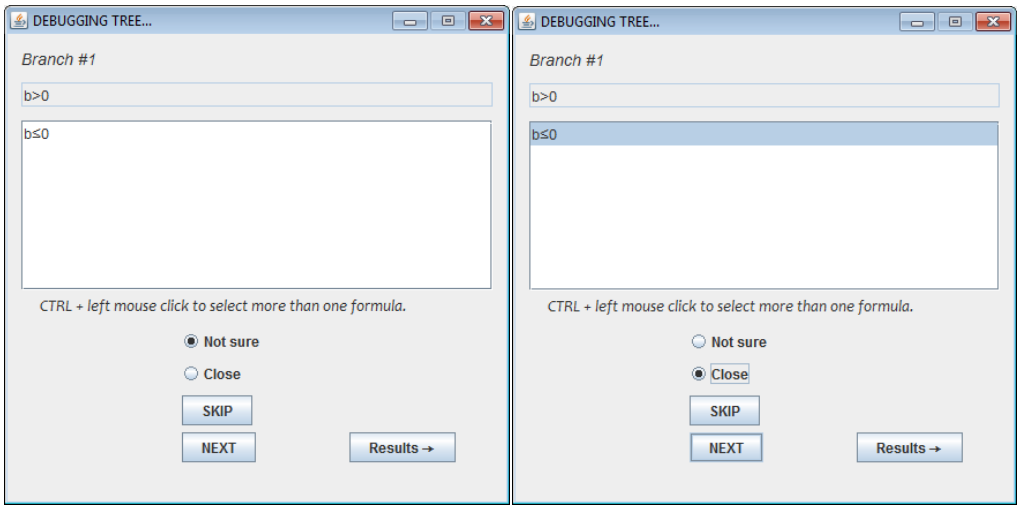
Como se puede observar, todas las expresiones están parentizadas, estando cada fórmula que forma parte de una conjunción, disyunción, disyunción exclusiva, implicación o implicación doble, entre paréntesis. Éstos también servirán para dar prioridad a las operaciones. La prioridad de las operaciones siempre debe quedar definida mediante los paréntesis.

Tras seleccionar la condición a comprobar el usuario deberá pulsar el botón *Draw* para comenzar el proceso de verificación.

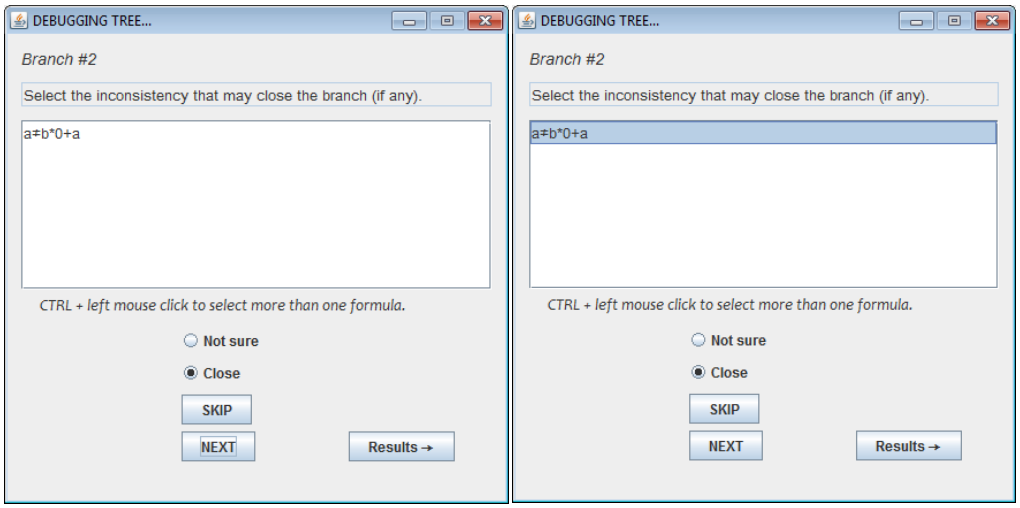
Por ejemplo tras seleccionar {P} A {I}, obtendremos:



El usuario, a continuación, deberá pulsar el botón *Debug* para comenzar a depurar su algoritmo, generando la siguiente pantalla en la que podemos ver la rama por la que va el usuario en el proceso de depuración, la fórmula a analizar y, en el textbox, las fórmulas candidatas para cerrar la rama (en este caso solo nos aparece una). El usuario deberá seleccionarla haciendo click con el botón izquierdo del ratón sobre ella. Si el usuario está seguro de que la fórmula seleccionada es opuesta a la fórmula "llamadora", cerrando así su rama, deberá pulsar el botón *Next* habiendo marcado previamente la opción *Close*.

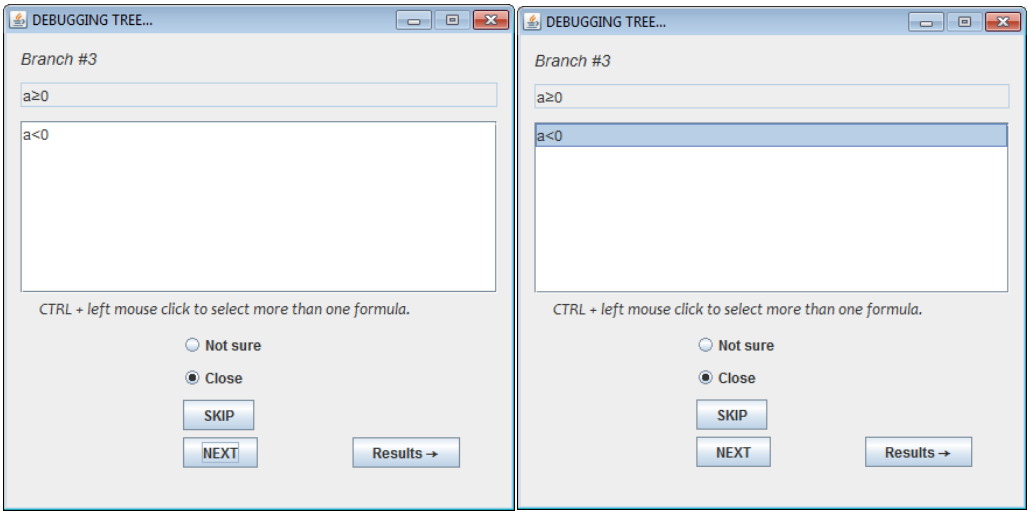


Tras hacer esto se mostrará la fórmula a analizar correspondiente a la siguiente rama:



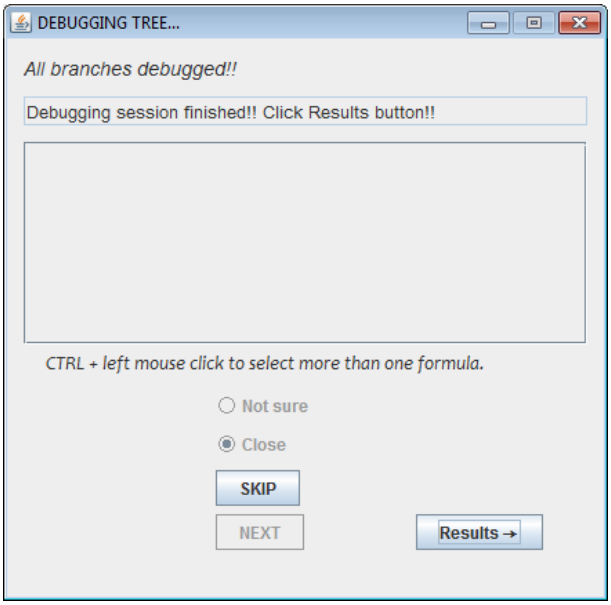
El usuario decidirá si la rama cierra o no, en caso de que considere que la fórmula seleccionada es opuesta su fórmula "llamadora", cerrando así su rama, volverá a pulsar el botón *Next*. En este caso, hay una inconsistencia: *a!=b*0+a,* que se puede simplificar como a!=a, lo que nos lleva a un absurdo, por lo que la rama se cierra.

Tras pulsar *Next*, nos aparecerá el formulario relativo a la tercera rama:

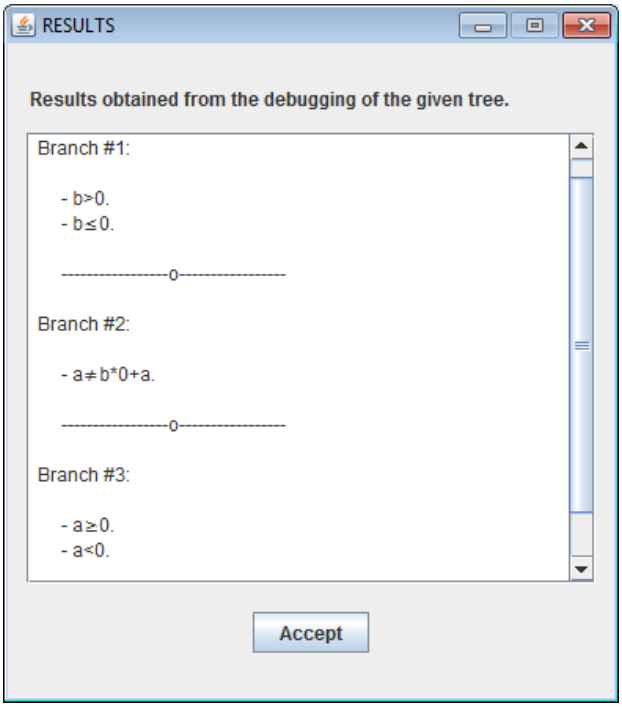


Análogamente a lo explicado para las ramas anteriores, se selecciona la fórmula candidata a cerrar la rama actual y en caso de que el usuario crea que la seleccionada es la indicada, pulsará el botón *Next* de nuevo.

Como ya hemos analizado todas las ramas del árbol, nos aparece el siguiente mensaje, indicando que todas las ramas se han cerrado:

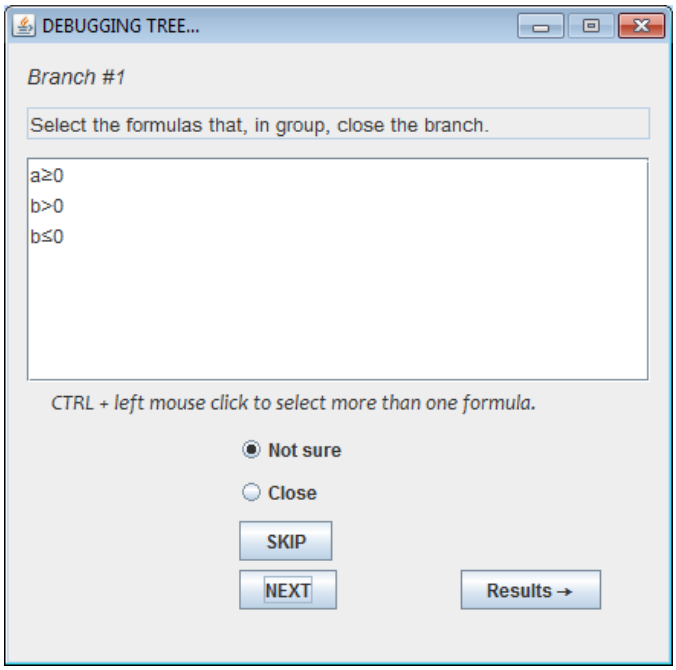


A continuación el usuario deberá pulsar el botón *Results* para ver los resultados obtenidos, apareciendo el siguiente formulario:

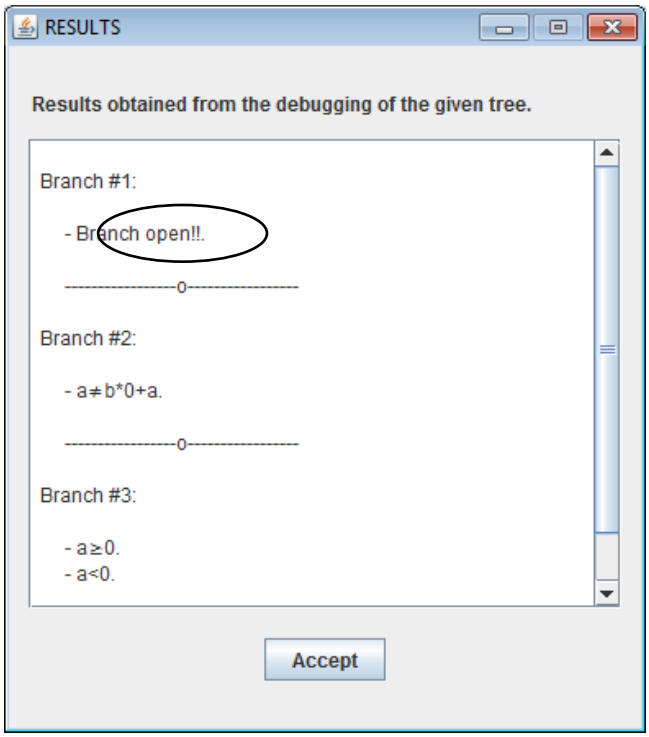


Aquí se puede ver cada rama con las fórmulas que la cierran.

En caso de que alguna rama no se hubiese cerrado, o si se hubiese marcado la opción *Not Sure* para todas las fórmulas, se mostrarían al usuario todas las fórmulas pertenecientes a esa rama en conjunto.



En caso de no estar seguro de qué fórmulas seleccionar para cerrar la rama, si el usuario vuelve a pulsar *Not Sure*, en el formulario de resultados el usuario podrá ver:



CAPÍTULO 6: CONCLUSIONES Y TRABAJO FUTURO

6.1 - CONCLUSIONES DEL TRABAJO

El objetivo principal cuando comenzamos a desarrollar TVT fue intentar crear una nueva herramienta para los alumnos basada en la lógica, que permitiese de manera sencilla, accesible e intuitiva la verificación formal y la depuración de algoritmos iterativos. Creemos que la herramienta que hemos desarrollado no solo cumple este objetivo, sino que será de gran utilidad para ayudar a los alumnos a comprender cómo realizar esta verificación y depuración de algoritmos.

Nuestra herramienta multiplataforma creemos que será muy útil para complementar los conocimientos que van adquiriendo los alumnos en la asignatura de EDI. Además podrán poner en práctica los conocimientos de asignaturas ya cursadas como lógica.

Nuestra herramienta permite la verificación y depuración de algoritmos iterativos, tanto de un bucle como con doble bucle introduciendo cada fragmento de código por separado. Además esta metodología también es aplicable para la verificación y depuración de algoritmos recursivos, lo cual se deja como una de las posibles extensiones de la herramienta explicadas más adelante.

En el apartado de ayuda de la herramienta podemos encontrar además del manual de la herramienta una breve introducción a los tableaux semánticos por si el usuario necesitase refrescar sus conocimientos. Pero creemos que TVT es muy sencillo de usar, lo que amplía el rango de personas que pueden utilizarla, ya que para cerrar cada una de las ramas del árbol sólo será necesario encontrar entre las fórmulas candidatas una que sea opuesta a la principal. En caso de no encontrarla la rama quedará abierta, indicando algún problema en el código verificado. Gracias a la facilidad nos da TVT para

comprobar si las ramas han sido cerradas el usuario ve rápidamente si su programa es correcto o no y como solucionarlo.

En conclusión TVT es una herramienta muy útil para la formación académica de los alumnos de FDI, ya que permite poner en práctica los conocimientos obtenidos y ayuda a la comprensión de los nuevos que se están adquiriendo. Además es muy sencilla e intuitiva. Permite verificar algoritmos rápidamente y depurarlos, encontrando los errores que estos pudiesen contener y guiando al usuario para su solución. Además admite múltiples extensiones como se explica más adelante y su instalación en diversos sistemas operativos. Todo esto convierte a TVT en una herramienta que aportará grandes beneficios a los alumnos y profesores de FDI y que podrá ser mejorada en años posteriores.

6.2 - VALORACIÓN DEL TRABAJO

Creemos que aunque hemos tenido que invertir muchas horas semanales en este proyecto desde el inicio del curso, el resultado final ha merecido la pena, ya que hemos logrado una herramienta que facilitará mucho la comprensión y la realización de la verificación y depuración de algoritmos iterativos a los alumnos. Además hemos podido ratificar está idea con la publicación en dos congresos internacionales de sendos artículos sobre TVT. Estos congresos son FECS'11 - The 2011 International Conference on Frontiers in Education: Computer Science and Computer Engineering y TICTTL´11 – Tools for teaching logic III junio 2011.

Tuvimos la suerte de poder ir e incluso impartir la conferencia en TICTTL. Tras la exposición, muchos de los asistentes, profesores de otras universidades del mundo, nos trasladaron su interés por nuestra herramienta y nos pidieron una web para descargarla, ya que la encontraron de gran interés y utilidad. Todo esto nos motivó y nos reafirmó en la idea de que tanto trabajo había merecido la pena. Además hace escasos días nos llegó un email de otro de los ponentes en el congreso, desde la universidad de West Bohemia en Pilsen ,Czech Republic. En este email nos comentaba que varios de sus compañeros le habían hablado también muy bien de nuestra herramienta y que nos invitaba a un Taller de aprendizaje que se celebrará en su universidad en febrero de 2012, para que presentemos nuestra herramienta.

Todos estos logros obtenidos así como haber alcanzado todos los objetivos propuestos nos hace pensar que TVT es una herramienta muy útil y valorada en el ámbito académico y por lo tanto, que todo el trabajo y esfuerzo invertido en su desarrollo ha merecido la pena.

6.3 - TRABAJO FUTURO

Actualmente en la sesión de depuración utilizamos al usuario como oráculo para que decida qué fórmulas son opuestas y cuáles no, y así poder ir cerrando las ramas del tableau. Una posible extensión sería añadir a nuestra herramienta un resolutor de restricciones para automatizar la sesión de depuración, sin que el usuario tenga que intervenir en el cierre de las ramas y que ésta se produzca de forma autónoma. El único inconveniente a este sistema será que actualmente el usuario al realizar la depuración ve donde tiene el error en caso de que éste exista. Debido a esto, la extensión de automatización debería indicar al usuario en caso de que queden ramas abiertas, qué ramas son y qué necesitaría encontrar para cerrarlas. El objetivo es que con esta información el usuario pueda corregir errores en su código o generar partes ausentes de éste. Para esta extensión, se ha incluido una clase denominada *HomogeneousMaker*, cuya finalidad es la de fijar un standard común a todas las fórmulas con únicamente los siguientes operadores comparacionales: $>, \geq, =, \neq$. Al final de su declaración se adjuntan consejos sobre posibles algoritmos necesarios.

Otra posible extensión, será crear una nueva pestaña en la interfaz para facilitar el uso de nuestra herramienta en programas recursivos. Esta nueva pestaña deberá incluir en lugar de las cinco pruebas formales de los algoritmos iterativos las siguientes:

- $P \Rightarrow a < b \vee a \geq b$
- $P \wedge a < b \Rightarrow Q_{c,r}^{0,a}$
- $P \wedge a \geq b \Rightarrow P_{a,b}^{a-b,b}$
- $P \wedge a \geq b \wedge Q_{a,b}^{a-b,b} \Rightarrow Q_{c,r}^{c+1,r}$
- $P \Rightarrow C \geq 0$
- $P \wedge a \geq b \Rightarrow C_{a,b}^{a-b,b} < C$

Actualmente si queremos verificar un algoritmo iterativo con doble bucle, deberemos escribir cada bucle por separado y verificarlos también de manera independiente. Otra posible mejora será que en el caso de querer verificar estos algoritmos se pueda escribir el algoritmo completo y que sea la herramienta la que separe ambos bucles en el proceso de verificación.

Análogamente en el caso de la recursión múltiple, la mejora sería que la herramienta verifique el algoritmo recursivo sin que el usuario tenga que probar cada fragmento recursivo por separado.

APÉNDICES

APÉNDICE A: CÓDIGO DE LA HERRAMIENTA

A.1 - PAQUETE GUI
Clase PrincipalW
Método jButtonDibujoVActionPerformed

Este método es el encargado de subdividir una cadena usando indexicall, FormulaSimplfier, WeakesPrecondition y LexLogAnalyzer según el árbol seleccionado para la verificación.

```
private void jButtonDibujoVActionPerformed(java.awt.event.ActionEvent evt) {
    ArrayList<String> fieldsText = new ArrayList();
    String prec = "", post = "", inv = "", bound = "";
    prec = whiteRemover(jTextFieldPrecondicionV.getText());
    post = whiteRemover(jTextFieldPostcondicionV.getText());
    inv = whiteRemover(jTextFieldInvarianteV.getText());
    bound = whiteRemover(jTextFieldCotaV.getText());
    fieldsText.add(prec);
    fieldsText.add(post);
    fieldsText.add(inv);
    fieldsText.add(bound);
    String err = checkSintax(fieldsText);
    if (err == null) {
       String toTreat = "";
       boolean error = false;
       HashMap HMresult = new HashMap();
       String bCond = whiteRemover(getBucleCondition());
       String program = jTextAreaProgramaV.getText().replace(" ", "");
       String WP = "";
       if (jCheckT1.isSelected()) {
          WeakestPrecondition WpModule = new WeakestPrecondition();
          WP = WpModule.getWP(inv, program, "fun", "while");
          toTreat = "(" + prec + ")/\\" + "¬(" + WP + ")";
          toTreat = coder(toTreat);
       } else if (jCheckT2.isSelected()) {
```

```
            WeakestPrecondition WpModule = new WeakestPrecondition();
            WP = WpModule.getWP(inv, program, "while", "fwhile");
            toTreat = "((" + inv + ")/\\(" + bCond + "))/\\" + "¬(" + WP + ")";
            toTreat = coder(toTreat);
        } else if (jCheckT3.isSelected()) {
            toTreat = "((" + inv + ")/\\¬(" + bCond + "))/\\¬(" + post + ")";
            toTreat = coder(toTreat);
        } else if (jCheckT4.isSelected()) {
            toTreat = "((" + inv + ")/\\(" + bCond + "))/\\¬(" + bound + ">=0" + ")";
            toTreat = coder(toTreat);
        } else if (jCheckT5.isSelected()) {
            WeakestPrecondition WpModule = new WeakestPrecondition();
            WP = WpModule.getWP(bound + "<ᵃᵃᵃ", program, "while", "fwhile");
            WP = WP.replace("ᵃᵃᵃ", bound); // Initial bound T is replaced here with its original value
to avoid wrong evaluation of finite loops.
            toTreat = "(((" + inv + ")/\\(" + bCond + "))" + ")/\\" + "¬(" + WP + ")";
            toTreat = coder(toTreat);
        } else {    error = true;
            JOptionPane.showMessageDialog(this, errorSeleccion, "ERROR",
JOptionPane.ERROR_MESSAGE);
        }
        if (!error) {
            toTreat = new Indexicals().doIndexical(toTreat);
            toTreat = new FormulaSimplifier().Simplifier(toTreat);
            HMresult = new LexLogAnalyzer().analyzeLog(toTreat);
            HMresult = recoder(HMresult, cod);
            Enumeration e = buttonGroup1.getElements();
            String name = "";
            JCheckBox elem;
            while (e.hasMoreElements()) {
                elem = (JCheckBox) e.nextElement();
                if (elem.isSelected()) {
                    name = elem.getText();
                    break;
                }
            }
            new Draw(HMresult, name, lang).run();
        }
    } else {    err = errCoding(err);
        JOptionPane.showMessageDialog(this, errorEntrada + ": " + err, "ERROR",
JOptionPane.ERROR_MESSAGE);
    }
```

```
}
```

A.2 - PAQUETE Indexical
Clase Indexicals

Método indexicalizer

```
/**
```

* In this method the String will be cutted and indexicalized.

* @param begin The begin cutting position of the String.

* @param end The end cutting position of the String.

* @param s The full String.

* @return The String with the indexical between begin and end positions subdivided in atomic formulas.

```
*/
private DataPair indexicalizer(int begin, int end, String s) {
    DataPair info = new DataPair(s, end);
    String p1 = "", p2 = "", sol = "";
    boolean simboloGuardado = false;
    int i = 0, nLog = 0, simbolPos = 0, loopCont = 0;
    p2 = s.substring(begin + 1, end - 1);
    for (int j = 0; j < numLogFormulas(p2); j++) {
        while ((nLog <= 1) && (i < p2.length())) {
            //this loop obtains an atomic formula
            if ((p2.charAt(i) == '<') || (p2.charAt(i) == '>')) {
                if (!simboloGuardado) {
                    simboloGuardado = true;
                    simbolPos = i;
                }
                nLog++;
            }
            if (nLog <= 1) {
                p1 = p1 + p2.charAt(i);
                i++;
            }
        }
        simboloGuardado = false;
        sol = sol + "(" + p1 + ")/\\";
```

```
        //add the atomic formula to the final solution
        i = simbolPos + 1;
        if (p2.charAt(i) == '=') {
            i++;
        }
        if (loopCont > 0) {
            sol = sol.substring(0, sol.length() - 2);
            if (!(loopCont + 1 == numLogFormulas(p2))) {
                sol = "(" + sol + ")/\\";
                //if necessary, stocks 2 atomic formulas between brackets
            }
        }
        p1 = "";
        nLog = 0;
        loopCont++;
    }
    p2 = s.substring(0, begin + 1) + sol + s.substring(end - 1);
    int newPos = s.substring(0, begin + 1).length() + sol.length();
    //updates the main String and saves the new position to continue checking for indexicals
    info.setSt(p2);
    info.setNewpos(newPos);
    return info;
}
```

A.3 - PAQUETE FormulaSimplifier
Clase FormulaSimplifier

Método simplifier

```
/**
 * A simplified formula is returned by this method.
 * @param s It contains the formula to be simplified.
 * @return the new simplified formula.
 */

public String Simplifier(String s) {
    s = removeBrackets(s);
    if ((s.contains("E")) || (s.contains("U"))) {
        s = quantSearchAndTreat(s);
    }
    if (s.contains("¬(¬(")) {
        s = doubleNegTreatment(s);
    }

    s = s.replace("¬¬", "");
    if (s.contains("¬((")) {
        s = getSubClausesNeg(s);
    }

    boolean notFinished = (s.contains("¬(") || s.contains("->") || s.contains("<->") || s.contains("!\\/"));
    int cut = 0;

    while (notFinished) {                    // It's been implemented using "if-else if" so to set
priority while simplifying.

        if (s.contains("¬(¬(")) {
            s = doubleNegTreatment(s);
        }
        if (s.contains("¬((")) {
            s = getSubClausesNeg(s);
        }
        if (s.contains("<->")) {                // p<->q.
```

```
        cut = s.indexOf("<->");                //It returns the position of "<->".

        boolean n = false;
        this.opC = 1;
        int opB = getOB(s, cut);
        String sPrev = "";
        if ((s.charAt(opB) == '(') && (opB != 0)) {
           opB++;
           n = true;              // We have a negated clause.
           sPrev = s.substring(0, opB);
        }
        String sAux1 = s.substring(opB, cut);// Clause p.



        this.opC = 1;
        opB = getCB(s, cut);
        if ((s.charAt(opB) == ')') && (n)) {
           opB--;
        }
        String sAux2 = s.substring(cut + 3, opB + 1);    // Clause q. "+1" includes char in last
position of the cut as substring method doesn't.

        String s1 = new String("(" + sAux1 + "->" + sAux2 + ")");   // (p->q).
        String s2 = new String("(" + sAux2 + "->" + sAux1 + ")");   // (q->p).

        if (n) {
           s2 = s2 + ")";
        }

        s = sPrev + s1 + "/\\" + s2;                        // (p->q) /\ (q->p).
     } else if (s.contains("->")) {                          // p->q.
        cut = s.indexOf("->");
        this.opC = 1;
        int opB = getOB(s, cut);                       // Position of the opening bracket.

        String sAux3 = s.substring(opB, cut);                    // Clause p.
        sAux3 = new StringBuffer(sAux3).insert(0, "¬").toString();      // It negates the first clause
from p to ¬p.
        cut = s.indexOf("->");                              //It returns the position of "->".
        s = s.substring(0, opB) + sAux3 + "\\/" + s.substring(cut + 2);         // ¬p\/q.
     } else if (s.contains("!\\/")) {                  // p!\/q. (exclusive-OR).
        cut = s.indexOf("!\\/");                //It returns the position of "!\/".
```

```
            this.opC = 1;
            int opB = getOB(s, cut);

            String sPrev = s.substring(0,opB);
            String sAux1 = s.substring(opB, cut);// Clause p.

            this.opC = 1;
            opB = getCB(s, cut);
            opB--;

            if (opB == s.length()-2) {
               if (s.charAt(opB) == ')') opB++;
            }

            String sAux2 = s.substring(cut + 3, opB/* + 1*/);    // Clause q. "+1" includes char in last position of the cut as substring method doesn't.

            String s1 = new String("(" + sAux1 + "\\/" + sAux2 + ")");   // (p\/q).
            String s2 = new String("(" + sAux1 + "/\\" + sAux2 + ")");   // (q/\p).

            s = sPrev + s1 + "/\\¬" + s2 + s.substring(opB);                    // (p\/q) /\ ¬(q/\p).
         } else if (s.contains("¬(")) {
            cut = s.indexOf("¬(");                  // cut has the position of the first occurrence of "¬(".
            String sTreated = s.substring(0, cut);          // Subsequence already treated.
            this.opC = 1;
            int i = getCB(s, cut + 2);
            String sToTreat = s.substring(cut + 2, i);          // Subsequence to treat (including char in last position of the cut but not '¬').

            String sNegT = negTreatment(sToTreat, 0);                  // It treats the negation of the treating clause.
            s = sTreated + "(" + sNegT + s.substring(i);                // It puts together all the string subsequences: the treated and the untreated ones.
         }
         s = s.replace("¬¬", "");
         notFinished = (s.contains("¬(") || s.contains("->") || s.contains("<->") || s.contains("!\\/"));
      }
      return removeBrackets(s);
   }
```

A.4 - PAQUETE WeakestPrecondition
Clase WeakestPrecondition

Metodo getWP

```
/**
 * WeakestPrecondition main method.
 * @param s In case of steps I and II, s will represent the Invariant.
 * In case of step V, s will represent the Bound.
 * @param program String that will be treated by
getStack(java.lang.String, java.lang.String, java.lang.String) getStack} .
 * @param begin Initial string.
 * @param end Final string.
 * @return String s whose identifiers has been replace by its value.
 */

   public String getWP(String s, String program,String begin, String end){
      // In case of steps I and II, s will represent the Invariant.
      // In case of step V, s will represent the Bound. The string s returned by this method will be
added this way: "s < Bound".

      Stack k = getStack(program,begin,end);
      String auxValue = "";
      while (!k.empty()){
         Triple t = (Triple) k.pop();          // The Triple object at the top of the stack is removed from
it and stored in t.
         ArrayList id = t.getId();
         Iterator itId = id.iterator();
         ArrayList value = t.getValue();
         Iterator itValue = value.iterator();
         if (t.getCond().contentEquals("TRUE")){
            auxValue = (String)itValue.next();
            if (auxValue.length() > 1) auxValue = "("+auxValue+")";
            s = s.replace((String)itId.next(),auxValue);   // Replaces each substring that matches the
exact id-string with the value-string.
         }
```

```
        else {
            auxValue = (String)itValue.next();
            if (auxValue.length() > 1) auxValue = "("+auxValue+")";
            String first = s.replace((String)itId.next(),auxValue);
            first = t.getCond()+"/\\"+"("+first+")";
            Triple t2 = (Triple) k.pop();                // The other Triple of the IF-THEN-ELSE statement
is removed from the stack here and stored in t2.
            ArrayList id2 = t2.getId();
            Iterator itId2 = id2.iterator();
            ArrayList value2 = t2.getValue();
            Iterator itValue2 = value2.iterator();
            auxValue = (String)itValue2.next();
            if (auxValue.length() > 1) auxValue = "("+auxValue+")";
            String second = s.replace((String)itId2.next(),auxValue);
            second = t2.getCond()+"/\\"+"("+second+")";
            s = "("+first+")\\/("+second+")";
        }
    }
    return s;
  }
```

A.5 - PAQUETE LexLogAnalyzer
Clase LexLogAnalyzer

Método analyzeLog

```
/**
    * The atomic formulas of the given String returned by the
    * { FormulaSimplifier} class is contained in the HashMap which is returned.
    * @param s is the String to be treated.
    * @return a {java.util.HashMap} with atomic formulas.
    */

  public HashMap analyzeLog(String s) {
    boolean isBinForm = false;
    HashMap<Key,Data> result = new HashMap<Key,Data>();
    Stack pila = new Stack();
    int opCount, cut, offset = 0;
    boolean p = false, neg = false;
    String aux, aux1 = "", aux2 = "";
    String f = " ";
    String k = "1";
    pila.push(k);
    pila.add(s);

    // Division of the Formula (String s)
    do {
      aux = (String) pila.pop();
      opCount = cut = 0;
      neg = p = false;

      // bracket counter and operator type
      for (int i = 0; i < aux.length(); i++) {
        if (aux.charAt(i) == '(') {
          p = true;
          opCount++;
        } else if (aux.charAt(i) == ')') {
```

```
            opCount--;
        }
        if (((aux.charAt(i) == '¬') && (opCount == 0) && (aux.length() == 2))
                || ((aux.charAt(i) == '¬') && (opCount == 0) && ((aux.length() > 2)
                && (aux.contains(">") || aux.contains("<") || aux.contains(">=") || aux.contains("<=")
                || aux.contains("=") || aux.contains("!=")))) {
            neg = true;
            break;
        }
        if (opCount == 0) {
            cut = i;
            break;
        }
    }

    //Evaluation of the formula cutting position
    if (p) {
        offset = 1;
    } else {
        offset = 0;
        cut = cutSearch(aux);
    }

    //analyzing binary operators
    if ((cut != -1) & !neg) {
        if (aux.charAt(cut + offset) == '/') {       // ALFA OPERATION /\
            if (aux.charAt(cut + 1 + offset) == '\\') {
                isBinForm = true;
                f = "a";
                aux1 = aux.substring(offset, cut);
                if (p) {
                    aux2 = aux.substring(cut + 3 + offset, aux.length() - 1);
                } else {
                    aux2 = aux.substring(cut + 2, aux.length());
                }
            }
        } else if (aux.charAt(cut + offset) == '\\') { //BETA OPERATION \/
            if (aux.charAt(cut + 1 + offset) == '/') {
                isBinForm = true;
                f = "b";
                aux1 = aux.substring(offset, cut);
```

```
                if (p) {
                    aux2 = aux.substring(cut + 3 + offset, aux.length() - 1);

                } else {
                    aux2 = aux.substring(cut + 2, aux.length());
                }
            }
        } else {
            isBinForm = false;
            pila.pop();
        }
        if (isBinForm) {
            Data data = new Data(aux1, aux2);
            k = (String) pila.pop();
            Key key = new Key(f, Integer.parseInt(k), false);
            result.put(key, data);
            pila.push(k + "2"); // Right Branch
            pila.push(aux2);
            pila.push(k + "1"); // Left Branch
            pila.push(aux1);
        }
    } else {
        pila.pop();
    }
  } while (!pila.empty());
  return result;
}
```

A.6 - PAQUETE Draw
Clase Drawing

Método Drawing calculation

/** This method returns an ArrayList of ArrayLists of points, lines and strings.

* @param h is a hashmap filled by { LexLogAnalyzer}

* @param t is a treemap with hashmap keys setted in order.

* @param a width of canvas.

* @param b height of canvas.

* @return an ArrayList of ArrayLists of points,lines and strings

*/

```
public ArrayList drawingCalculation(HashMap<Key, Data> h, TreeSet t, int a, int b) {
    //resp initialitation
    ArrayList<ArrayList> resp = new ArrayList<ArrayList>();
    resp.add(new ArrayList<PointDPaint>());
    resp.add(new ArrayList<LineDPaint>());
    resp.add(new ArrayList<StringDPaint>());
    Iterator it = t.iterator();
    Key k = new Key();
    Data d;
    ArrayList<PixelInfo> openBranches = new ArrayList();
    while (it.hasNext()) {
        String s = (String) it.next();
        k = k.toKey(s);
        k = this.getKeyMemDirection(h, k);
        d = h.get(k);
        if (s.contains("a")) {
            resp = alphaPaint(d, k, openBranches, resp, a, b);
        } else if (s.contains("b")) {
            resp = betaPaint(d, k, openBranches, resp, a, b);
        }
```

```
        }
        OB = openBranches;
        return resp;
    }
```

Método alphaPaint

/ /**

* @param d is a Data Object that contains the Strings of the painted formula.

* @param k is a Key Object that contains the type of the painted formula.

* @param openBranches is the ArrayList of branches whose painting coordinates have been calculated.

* @param a is the ArrayList of the actual coordinates of the predecessor points, lines and Strings

* @param w width needed in case of be the first formula to be painted

* @param desp is the number of pixeles that the formula need to be moved to the right.

* @return an ArrayList of ArrayLists of points,lines and strings

*/

```
private ArrayList alphaPaint(Data d, Key k, ArrayList openBranches, ArrayList<ArrayList> a, int w, int desp) {
        Iterator itBranches;
        ArrayList temporalOB = new ArrayList();
        String aux = "(" + d.getS1() + ")/\\(" + d.getS2() + ")";
        String aux2, aux3;
        if (k.getNum() == 1) {
            int x1 = w / 2 + desp;
            ArrayList sl = new ArrayList();
            sl.add(aux);
            PixelInfo root = new PixelInfo(x1, 25, "1", aux, null, null, sl, null);
            PointDPaint pointD = new PointDPaint(x1 - 3, 25);
```

```
a.get(0).add(pointD);
LineDPaint lineD = new LineDPaint(x1, 25, x1, 55);
a.get(1).add(lineD);
StringDPaint stringD = new StringDPaint(aux, 1, x1 - aux.length() * 5 / 2, 20);
a.get(2).add(stringD);

//s1 añadir 1
aux2 = d.getS1();
ArrayList sl1 = new ArrayList();
sl1.add(aux);
sl1.add(aux2);
ArrayList a1 = new ArrayList();
a1.add(1);
a1.add(11);
PointDPaint pointD2 = new PointDPaint(x1 - 3, 55);
a.get(0).add(pointD2);
LineDPaint lineD2 = new LineDPaint(x1, 55, x1, 85);
a.get(1).add(lineD2);
if (isAtomic(aux2)) {
   StringDPaint stringD2 = new StringDPaint(aux2, 11, x1, 55);
   a.get(2).add(stringD2);
}
PixelInfo p1 = new PixelInfo(x1, 115, "11", aux2, k, a1, sl1, root);
p1.setDraw(true);

//s2 añadir 2
aux3 = d.getS2();
ArrayList a2 = new ArrayList();
a2.add(1);
a2.add(11);
a2.add(12);
ArrayList sl2 = new ArrayList();
sl2.add(aux);
sl2.add(aux2);
sl2.add(aux3);
PointDPaint pointD3 = new PointDPaint(x1 - 3, 85);
a.get(0).add(pointD3);
if (isAtomic(aux3)) {
   StringDPaint stringD3 = new StringDPaint(aux3, 12, x1, 85);
   a.get(2).add(stringD3);
}
```

```
        PixelInfo p2 = new PixelInfo(x1, 115, "12", aux3, k, a2, sl2, root);

        openBranches.add(p1);
        openBranches.add(p2);
      } else {
        //caso general
        itBranches = openBranches.iterator();
        while (itBranches.hasNext()) {
          PixelInfo coord = (PixelInfo) itBranches.next();
          if (!coord.getDraw() && isSon(coord, k)) {
            LineDPaint lineD = new LineDPaint(coord.getCoordx(), coord.getCoordy() - 30,
coord.getCoordx(), coord.getCoordy());
            a.get(1).add(lineD);

            //s1 añadir 1
            aux2 = d.getS1();
            PointDPaint pointD = new PointDPaint(coord.getCoordx() - 3, coord.getCoordy());
            a.get(0).add(pointD);
            LineDPaint lineD2 = new LineDPaint(coord.getCoordx(), coord.getCoordy(),
coord.getCoordx(), coord.getCoordy() + 30);
            a.get(1).add(lineD2);
            if (isAtomic(aux2)) {
              StringDPaint stringD = new StringDPaint(aux2, (k.getNum() * 10) + 1,
coord.getCoordx(), coord.getCoordy());
              a.get(2).add(stringD);
            }
            ArrayList a1 = new ArrayList();
            a1.addAll(coord.getPreviousPixelList());
            a1.add((k.getNum() * 10) + 1);
            ArrayList sl3 = new ArrayList();
            sl3.addAll(coord.getPreviousStringList());
            sl3.add(aux2);
            PixelInfo p1 = new PixelInfo(coord.getCoordx(), coord.getCoordy() + 60,
String.valueOf(k.getNum()) + "1", aux2, k, a1, sl3, coord);
            p1.setDraw(true);

            //s2 añadir 2
            aux3 = d.getS2();
            PointDPaint pointD2 = new PointDPaint(coord.getCoordx() - 3, coord.getCoordy() +
30);
            a.get(0).add(pointD2);
            if (isAtomic(aux3)) {
              StringDPaint stringD2 = new StringDPaint(aux3, (k.getNum() * 10) + 2,
coord.getCoordx(), coord.getCoordy() + 30);
```

```
                a.get(2).add(stringD2);
            }

            ArrayList a2 = new ArrayList();
            a2.addAll(coord.getPreviousPixelList());
            a2.add((k.getNum() * 10) + 1);
            a2.add((k.getNum() * 10) + 2);
            ArrayList sl4 = new ArrayList();
            sl4.addAll(coord.getPreviousStringList());
            sl4.add(aux2);
            sl4.add(aux3);
            PixelInfo p2 = new PixelInfo(coord.getCoordx(), coord.getCoordy() + 60,
String.valueOf(k.getNum()) + "2", aux3, k, a2, sl4, p1);
            coord.setDraw(true);

            temporalOB.add(p1);
            temporalOB.add(p2);
          }
        }
      }
      for (int i = 0; i < temporalOB.size(); i++) {
        if (!openBranches.contains(temporalOB.get(i))) {
          openBranches.add(temporalOB.get(i));
        }
      }
      temporalOB.clear();
      openBranches = updateOB(openBranches, k);
      return a;
    }
```

Método betaPaint

```
  /**
```

* @param d is a Data Object that contains the Strings of the painted formula.

* @param k is a Key Object that contains the type of the painted formula.

* @param openBranches is the ArrayList of branches whose painting coordinates have been calculated.

* @param a is the ArrayList of the actual coordinates of the predecessor points, lines and Strings

* @param w width needed in case of be the first formula to be painted

* @param desp is the number of pixeles that the formula need to be moved to the right.

* @return an ArrayList of ArrayLists of points,lines and strings

*/

```
private ArrayList betaPaint(Data d, Key k, ArrayList openBranches, ArrayList<ArrayList> a, int w, int desp) {
    ArrayList temporalOB = new ArrayList();
    Iterator itBranches;
    String aux = "(" + d.getS1() + ")\\/(" + d.getS2() + ")";
    String aux2, aux3;
    itBranches = openBranches.iterator();
    PixelInfo coord = null;
    if (k.getNum() == 1) {
        int x1 = mc.getWidth() / 2;
        ArrayList sl = new ArrayList();
        sl.add(aux);
        PixelInfo root = new PixelInfo(x1, 25, "1", aux, null, null, sl, null);
        PointDPaint pointD = new PointDPaint(x1 - 3, 25);
        a.get(0).add(pointD);
        LineDPaint lineD = new LineDPaint(x1, 25, x1 - 50, 55);
        a.get(1).add(lineD);
        LineDPaint lineD2 = new LineDPaint(x1, 25, x1 + 50, 55);
        a.get(1).add(lineD2);
        StringDPaint stringD = new StringDPaint(aux, 1, x1 - aux.length() * 5 / 2, 20);
        a.get(2).add(stringD);

        //s1
        aux2 = d.getS1();
        PointDPaint pointD2 = new PointDPaint(x1 - 53, 55);
        a.get(0).add(pointD2);
        if (isAtomic(aux2)) {
            StringDPaint stringD2 = new StringDPaint(aux2, 11, x1 - 50 - aux2.length() * 5, 55);
```

```
            a.get(2).add(stringD2);
        }
        ArrayList a1 = new ArrayList();
        a1.add(1);
        a1.add(11);
        ArrayList sl1 = new ArrayList();
        sl1.add(aux);
        sl1.add(aux2);
        PixelInfo p1 = new PixelInfo(x1 - 50, 85, "11", aux2, k, a1, sl1, root);

        //s2
        aux3 = d.getS2();
        PointDPaint pointD3 = new PointDPaint(x1 + 47, 55);
        a.get(0).add(pointD3);
        if (isAtomic(aux3)) {
            StringDPaint stringD3 = new StringDPaint(aux3, 12, x1 + 50, 55);
            a.get(2).add(stringD3);
        }
        ArrayList a2 = new ArrayList();
        a2.add(1);
        a2.add(12);
        ArrayList sl2 = new ArrayList();
        sl2.add(aux);
        sl2.add(aux3);
        PixelInfo p2 = new PixelInfo(x1 + 50, 85, "12", aux3, k, a2, sl2, root);

        openBranches.add(p1);
        openBranches.add(p2);
    } else {
        //caso general
        while (itBranches.hasNext()) {
            coord = (PixelInfo) itBranches.next();
            if (!coord.getDraw() && isSon(coord, k)) {

                coord.setDraw(true);
                LineDPaint lineD = new LineDPaint(coord.getCoordx(), coord.getCoordy() - 30,
coord.getCoordx() - 40 * betas, coord.getCoordy() - 10);
                a.get(1).add(lineD);
                LineDPaint lineD2 = new LineDPaint(coord.getCoordx(), coord.getCoordy() - 30,
coord.getCoordx() + 50 * betas, coord.getCoordy() + 5);
                a.get(1).add(lineD2);
```

```
                //s1
                aux2 = d.getS1();
                PointDPaint pointD = new PointDPaint(coord.getCoordx() - 40 * betas - 3,
coord.getCoordy() - 10);
                a.get(0).add(pointD);
                if (isAtomic(aux2)) {
                   StringDPaint stringD = new StringDPaint(aux2, (k.getNum() * 10) + 1,
coord.getCoordx() - 40 * betas - aux2.length() * 5, coord.getCoordy() - 10);
                   a.get(2).add(stringD);
                }
                ArrayList a1 = new ArrayList();
                a1.addAll(coord.getPreviousPixelList());
                a1.add((k.getNum() * 10) + 1);
                ArrayList sl3 = new ArrayList();
                sl3.addAll(coord.getPreviousStringList());
                sl3.add(aux2);
                PixelInfo p1 = new PixelInfo(coord.getCoordx() - 40 * betas, coord.getCoordy() + 25,
String.valueOf(k.getNum()) + "1", aux2, k, a1, sl3, coord);

                //s2
                aux3 = d.getS2();
                PointDPaint pointD2 = new PointDPaint(coord.getCoordx() + 50 * betas - 3,
coord.getCoordy());
                a.get(0).add(pointD2);
                if (isAtomic(aux3)) {
                   StringDPaint stringD2 = new StringDPaint(aux3, (k.getNum() * 10) + 2,
coord.getCoordx() + 50 * betas, coord.getCoordy());
                   a.get(2).add(stringD2);
                }
                ArrayList sl4 = new ArrayList();
                sl4.addAll(coord.getPreviousStringList());
                sl4.add(aux3);
                ArrayList a2 = new ArrayList();
                a2.addAll(coord.getPreviousPixelList());
                a2.add((k.getNum() * 10) + 2);
                PixelInfo p2 = new PixelInfo(coord.getCoordx() + 50 * betas, coord.getCoordy() + 35,
String.valueOf(k.getNum()) + "2", aux3, k, a2, sl4, coord);

                temporalOB.add(p1);
                temporalOB.add(p2);
             }
          }
       }
       for (int i = 0; i < temporalOB.size(); i++) {
```

```
            if (!openBranches.contains(temporalOB.get(i))) {
                openBranches.add(temporalOB.get(i));
            }
        }
        temporalOB.clear();
        openBranches = updateOB(openBranches, k);
        betas--;
        return a;
    }
```

APÉNDICE B: ARTÍCULOS PUBLICADOS

B.1 - TICTTL 2011

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).

P. Blackburn et al. (Eds.): TICTTL 2011, LNAI 6680, pp. 239–248, 2011.

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.

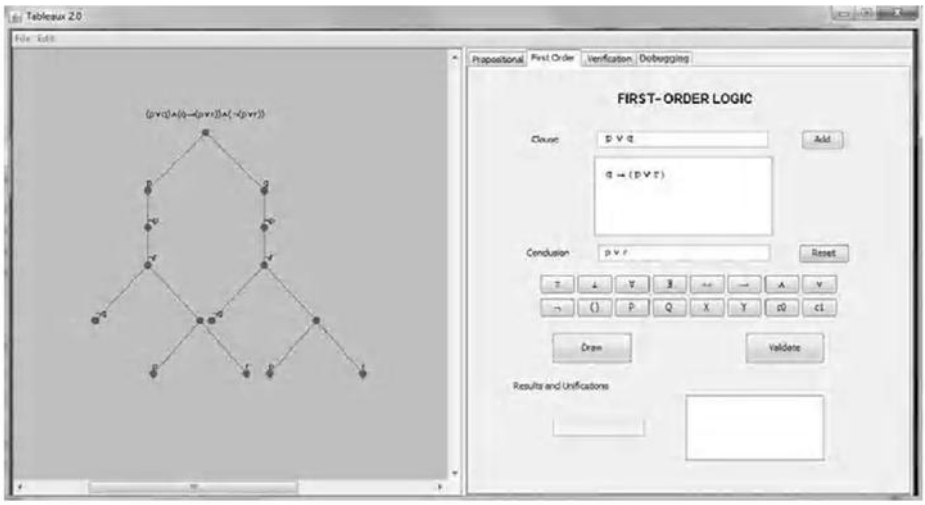


Fig. 1. The logic teaching tool TABLEAUX

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.

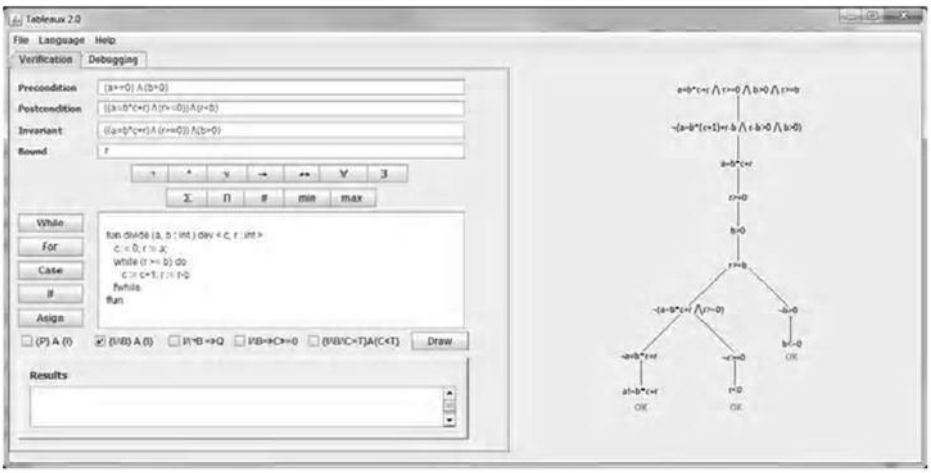


Fig. 2. The logic teaching tool `TABLEAUX` for verification of algorithms

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` $< c, r : int >$
  $c := 0 ; \ \ r := a ;$
  $\{ I \ : \ a = b * c + r \ \wedge \ r \geq 0 \ \wedge \ b > 0 \ , \ C : r \}$
  `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 [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 $R^{e,\ldots}_{x,\ldots}$ 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 $\{I \wedge r \geq b\}\ c := c + 1; r := r - b\ \{I\}$:

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

(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)	$\neg(a = b*(c+1)+r-b \wedge r-b \geq 0 \wedge b > 0)$	$\{\neg(I^{\,c+1,\,r-b}_{c,\,r})\}$

$(\beta, 6)$

(7) $a \neq b*c+r$	(8) $r < b$	(9) $b \leq 0$
✓ $(2,7)$	✓ $(5,8)$	✓ $(4,9)$

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 $I' \wedge r < b \Rightarrow Q$:

(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)	$\neg(a = b*c+r \wedge r \geq 0 \wedge r < b)$	$\{\neg Q\}$

$(\beta, 4)$

(5) $a \neq b*c+r$	(6) $r < 0$	(7) $r \geq b$
✓ $(2,5)$	⇓ $\times$ ⇓	✓ $(3,7)$

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 $\{I'' \wedge r \geq b \wedge C = T\}$ $c := c+1$; $r := r-b$ $\{C < T\}$:

(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)	$b \leq 0$	$(=, 5, 6)$

⇓

$\times$ ⇒ 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** $< c, r : int >$
  $c := 0$; r := 0; ←-- *wrong code*!
  $\{ I \ : \ a = b * c + r \ \wedge \ r \geq 0 \ \wedge \ b > 0 \ , \ C : r \}$
  while $r \geq b$ **do** r := r - b **fwhile** ←-- *missing code*!
ffun
$\{ Q \ : \ a = b * c + r \ \wedge \ r \geq 0 \ \wedge \ r < b \}$

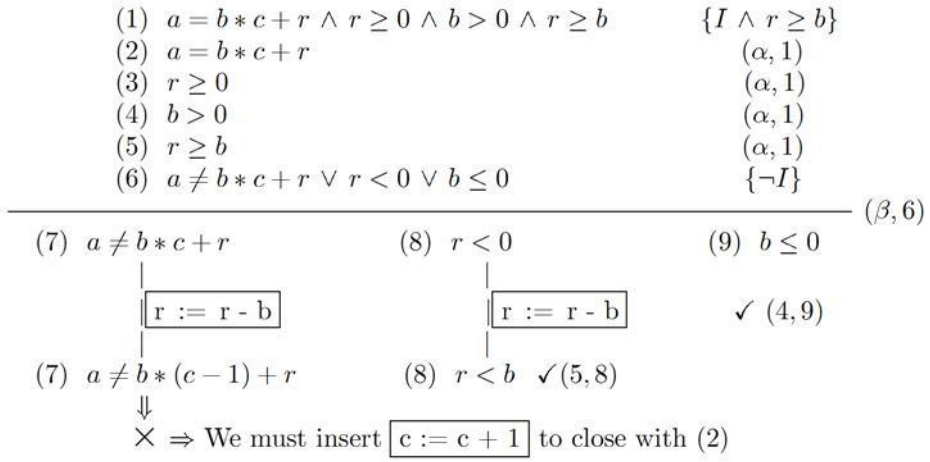
If we try to verify this erroneous algorithm, we can execute again the tool. Now, TABLEAUX displays an open tableau $P \Vdash \neg I$ for debugging $\{P\}\, c := 0; r := 0\, \{I\}$, instead of $P \Vdash \neg(I^{0,0}_{c,r})$. However, the weakest precondition $I^{0,0}_{c,r}$ is built from (5) and (6),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

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

$(\beta, 4)$

(5) $a \neq b * c + r$	(6) $r < 0$	(7) $b \leq 0$
|	|	
| c := 0	| r := 0 (or a)	✓ (3, 7)
|	|	
(5) $a \neq r$	(6) $0 \ (or \ a) < 0$	
|		
| r := 0	✓ *or* (✓(2, 6))	
|		
(5) $a \neq 0$		
⇓		
× ⇒ We must replace r := 0 by 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 $\{I \wedge r \geq b\}\ r := r - b\ \{I\}$:



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)

B.2 - FECS 2011

Debido a que el congreso se realiza en julio, el artículo está por publicar.

BIBLIOGRAFÍA

1. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
2. Kaldewaij, A.: Programming: The Derivation of Algorithms. Prentice-Hall International Series in Computer Science (1990)
3. Naish, L.: A Declarative Debugging Scheme. Journal of Functional and Logic Programming 3 (1997)
4. Shapiro, E.Y.: Algorithmic Program Debugging. MIT Press, Cambridge (1983)
5. Martí Oliet, Narciso, Segura Díaz, Clara María, Verdejo López, José Alberto : Especificación, derivación y análisis de algoritmos, Pearson Prentice Hall (2006)
6. María Teresa Hortalá Gonz: Matemática discreta y lógica matemática, Editorial Complutense (2008)
7. Página oficial de Netbeans: http://netbeans.org/
8. Página oficial de Java: http://www.sun.com/java/

GLOSARIO

Tableaux semántico: método de cálculo lógico que permite demostrar una conclusión a partir de unas premisas dadas, o bien construir un contraejemplo en el caso de que la conclusión no sea consecuencia lógica de las premisas.

Verificación : Comprobación de la corrección de un algoritmo dado.

Depuración: Proceso de encontrar y corregir los errores de programación de un algoritmo.

Algoritmo Iterativo: Aquel cuya característica principal es funcionar en base a un procedimiento cíclico.

Algoritmo Recursivo: Aquel que expresa la solución a un problema en términos de una llamada a sí mismo, conocida como llamada recursiva. La llamada recursiva se deberá definir como un problema de complejidad menor y deberá existir un caso base para evitar una recursividad infinita.

Precondición : conjunto de estados válidos al comienzo del algoritmo.

Postcondición: conjunto de estados alcanzables a la terminación del algoritmo.

Precondición más débil: se denota por pmd (A, Q) y se define como el conjunto de todos los estados tales que una ejecución de A que comience en uno de ellos se garantiza que termina en el estado Q en un tiempo finito.

Invariante: conjunto de estados que deben satisfacerse antes de la primera iteración del bucle, mientras se ejecute el cuerpo del bucle y al salir del bucle.

Función de Cota: número de iteraciones que quedan por realizar en el algoritmo.

Diseño por Contrato: Metodología para el diseño e implementación de aplicaciones. Considera los elementos del diseño como participantes en un contrato de negocios. Debido a esto el diseño se podrá realizar asumiendo que se cumplirán ciertas precondiciones y deberán garantizar ciertas postcondiciones e invariantes. Con este sistema garantizamos menos errores en el código y en caso de haberlos la detección es más sencilla.

Especificación algebraica de un TAD: especificar las propiedades que lo definen. Deberá definir el comportamiento del TAD para cualquier usuario que lo necesite, y deberá ser legible, precisa, no ambigua y general.

Lógica de Hoare: extensión de la lógica de predicados de primer orden para razonar sobre la corrección de programas imperativos construidos sobre una signatura (S, Σ) mediante una serie de reglas de inferencia.

Demostración automática: Realizar demostraciones mediante un programa de ordenador.