Enfoque práctico a la programación basada en invariantes y el editor Socos

28 de noviembre de 2011

Pablo Federico Dobal

Lic. en Cs. de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura (UNR) - Rosario, Argentina

Supervisores

Ralph-Johan Back, Johannes Eriksson y Viorel Preoteasa

Departamento de Tecnologías de la Información

Facultad de Tecnología

Åbo Akademi Universidad

Joukahaisenkatu 3-5 A, 20520 Turku, Finlandia

Resumen

En este trabajo serán presentadas las ideas detrás de una prometedora técnica para la construcción de software, esto es la programación basada en invariantes (IBP). Este trabajo expondrá el marco de trabajo teórico y práctico en el cual IBP es soportado. De la misma manera será presentada la herramienta Socos. Socos permite la construcción de programas utilizando IBP. Finalmente un conjunto de programas serán construidos utilizando IBP y Socos y se probará su corrección utilizando PVS. También serán presentados algunos detalles de implementación de la nueva versión de la herramienta Socos denominada Socos segunda generación.

Índice

1. Introducción **10**

2. Casos de estudio **16**
2.1. Suma 17
2.2. Raíz cuadrada 18
2.3. Exponencial 18
2.4. Factorial 19
2.5. Palíndrome 20
2.6. Partición 20
2.7. Búsqueda lineal 21
2.8. Búsqueda lineal en dos dimensiones 23
2.9. Búsqueda binaria 24
2.10. Insertion sort 25
2.11. Dutch flag problem 27
2.12. Promedio 28
2.13. Encontrar los M elementos más pequeños 30
2.14. Mapeo 31
2.15. Merge 32

3. Socos: Detalles de la implementación de su interfaz gráfica **35**

4. Conclusiones **65**

Índice de figuras

1. Algoritmo para sumar los primeros N números naturales en Python 10
2. Diagrama de flujo del algoritmo para sumar los primeros N números naturales . . . 11
3. Diagrama de invariantes del programa para sumar los primeros N números naturales . . . 12
4. Clases que componen el modelo . . . 37
5. Clases que componen la vista . . . 38
6. Clases que componen al controlador . . . 39
7. Movimiento de un segmento de una transición . . . 40
8. Definición de la clase LabelLocator . . . 41
9. Definición de la clase LabelLocator . . . 42
10. Método ReferencePoint de la clase LabelLocator . . . 42
11. Método que calcula la nueva posición de la etiqueta - Parte 1 . . . 44
12. Método que calcula la nueva posición de la etiqueta - Parte 2 . . . 45
13. Snapshot del la GUI de Socos: La longitud del invariante es más grande que el ancho de la situación. . . . 45
14. Socos GUI agregando el tipo de los objetos a los identificadores de los objetos. . . . 46
15. Vista parcial del código que permite caracteres especiales en el texto de los invariantes. . . . 48
16. Método *getDescendants* que retorna todos los objetos anidados de un objeto de tipo caja. . . . 49
17. El área de trabajo de Socos: Un borde fue agregado a las figuras de las presituaciones y postsituaciones. . . . 49
18. Comportamiento previo y actual de la acción de copy y paste. . . . 50
19. Código agregado al editor. . . . 52
20. Código para definir la acción para llevar a cabo el chequeo del programa . . . 54
21. Código para registrar la acción CheckAction dentro del editor. . . . 55
22. Código utilizado para la comunicación entre la GUI y el compilador de Socos . . . 55
23. Código para definir la acción que abre la teoría PVS asociada al programa . . . 56
24. Código para registrar la acción OpenPVSFileAction en el editor . 57
25. Código para abrir la teoría PVS asociada al programa . . . 57
26. Implementación del método createPartControl . . . 57
27. Implementación del método *setErrorList* . . . 58
28. Ejemplo de la notificación de errores para un simple programa . 59
29. Sintaxis generada por la nueva GUI de Socos para el programa que suma los primeros números naturales . . . 61
30. Ventana de preferencias . . . 64
31. Diagrama de invariantes para calcular la suma de los primeros N números naturales . . . 68

32. Representación textual del programa que suma de los primeros N números naturales 69
33. Teoría PVS del programa que suma de los primeros N números naturales - Parte 1 70
34. Teoría PVS del programa que suma de los primeros N números naturales - Parte 2 71
35. Implementación en Python del programa que suma de los primeros N números naturales. 72
36. Diagrama de invariantes para calcular la raíz cuadrada 73
37. Representación textual del programa para calcular la raíz cuadrada 74
38. Teoría PVS del programa para calcular la raíz cuadrada - Parte 1 75
39. Teoría PVS del programa para calcular la raíz cuadrada - Parte 2 76
40. Implementación en Python del programa para calcular la raíz cuadrada 77
41. Diagrama de invariantes del programa para calcular potencias . . 78
42. Representación textual del programa para calcular potencias - Parte 1 79
43. Representación textual del programa para calcular potencias - Parte 2 80
44. Teoría PVS del programa para calcular potencias - Parte 1 . . . 81
45. Teoría PVS del programa para calcular potencias - Parte 2 . . . 82
46. Implementación en Python del programa para calcular potencias 83
47. Diagrama de invariantes del programa para calcular el factorial de N 84
48. Representación textual del programa para calcular el factorial de N 85
49. Teoría PVS del programa para calcular el factorial de N - Parte 1. 86
50. Teoría PVS del programa para calcular el factorial de N - Parte 2. 87
51. Implementación en Python del programa para calcular el factorial de N 88
52. Diagrama de invariantes del programa para decidir si un arreglo es palíndrome 89
53. Representación textual del programa para decidir si un arreglo es palíndrome 90
54. Teoría PVS del programa para decidir si un arreglo es palíndrome - Parte 1 91
55. Teoría PVS del programa para decidir si un arreglo es palíndrome - Parte 2 92
56. Teoría PVS del programa para decidir si un arreglo es palíndrome - Parte 3 93
57. Implementación en Python del programa para decidir si un arreglo es palíndrome 94
58. Diagrama de invariantes del programa para particionar un arreglo 95
59. Representación textual del programa para particionar un arreglo 96
60. Teoría PVS del programa para particionar un arreglo - Parte 1 . 97
61. Teoría PVS del programa para particionar un arreglo - Parte 2 . 98

62. Teoría PVS del programa para particionar un arreglo - Parte 3 . 99
63. Implementación en Python del programa para particionar un arreglo . . . 100
64. Diagrama de invariantes de la búsqueda lineal . . . 101
65. Representación textual del programa para realizar una búsqueda lineal . . . 102
66. Teoría PVS del programa para realizar una búsqueda lineal - Parte 1 . . . 103
67. Teoría PVS del programa para realizar una búsqueda lineal - Parte 2 . . . 104
68. Implementación en Python del programa para realizar una búsqueda lineal . . . 105
69. Diagrama de invariantes de la búsqueda lineal en dos dimensiones 106
70. Representación textual del programa para realizar una búsqueda lineal en dos dimensiones . . . 107
71. Teoría PVS del programa para realizar una búsqueda lineal en dos dimensiones - Parte 1 . . . 108
72. Teoría PVS del programa para realizar una búsqueda lineal en dos dimensiones - Parte 2 . . . 109
73. Teoría PVS del programa para realizar una búsqueda lineal en dos dimensiones - Parte 3 . . . 110
74. Implementación en Python del programa para realizar una búsqueda lineal en dos dimensiones . . . 111
75. Diagrama de invariantes del programa para realizar una búsqueda binaria . . . 112
76. Representación textual del programa para realizar una búsqueda binaria . . . 113
77. Teoría PVS del programa para realizar una búsqueda binaria - Parte 1 . . . 114
78. Teoría PVS del programa para realizar una búsqueda binaria - Parte 2 . . . 115
79. Teoría PVS del programa para realizar una búsqueda binaria - Parte 3 . . . 116
80. Teoría PVS del programa para realizar una búsqueda binaria - Parte 4 . . . 117
81. Teoría PVS del programa para realizar una búsqueda binaria - Parte 5 . . . 118
82. Implementación en Python del programa para realizar una búsqueda binaria . . . 119
83. Diagrama de invariantes del programa Insertion sort . . . 120
84. Representación textual del programa Insertion sort . . . 121
85. Teoría PVS del programa Insertion sort - Parte 1 . . . 122
86. Teoría PVS del programa Insertion sort - Parte 2 . . . 123
87. Teoría PVS del programa Insertion sort - Parte 3 . . . 124
88. Implementación en Python del programa Insertion sort . . . 125

89. Diagrama de invariantes del programa para resolver el problema de la bandera alemana . . . 126
90. Representación textual del programa para resolver el problema de la bandera alemana . . . 127
91. Teoría PVS del programa para resolver el problema de la bandera alemana - Parte 1 . . . 128
92. Teoría PVS del programa para resolver el problema de la bandera alemana - Parte 2 . . . 129
93. Teoría PVS del programa para resolver el problema de la bandera alemana - Parte 3 . . . 130
94. Implementación en Python del programa para resolver el problema de la bandera alemana . . . 131
95. Diagrama de invariantes del programa que procesa el promedio de un arreglo . . . 132
96. Representación textual del programa que procesa el promedio de un arreglo - Parte 1 . . . 133
97. Representación textual del programa que procesa el promedio de un arreglo - Parte 2 . . . 134
98. Teoría PVS del programa que procesa el promedio de un arreglo - Parte 1 . . . 135
99. Teoría PVS del programa que procesa el promedio de un arreglo - Parte 2 . . . 136
100. Teoría PVS del programa que procesa el promedio de un arreglo - Parte 3 . . . 137
101. Teoría PVS del programa que procesa el promedio de un arreglo - Parte 4 . . . 138
102. Implementación en Python del programa que procesa el promedio de un arreglo . . . 139
103. Diagrama de invariantes del programa para calcular los M elementos más pequeños de un arreglo . . . 140
104. Representación textual del programa para calcular los M elementos más pequeños de un arreglo - Parte 1 . . . 141
105. Representación textual del programa para calcular los M elementos más pequeños de un arreglo - Parte 2 . . . 142
106. Teoría PVS del programa para calcular los M elementos más pequeños de un arreglo - Parte 1 . . . 143
107. Teoría PVS del programa para calcular los M elementos más pequeños de un arreglo - Parte 2 . . . 144
108. Teoría PVS del programa para calcular los M elementos más pequeños de un arreglo - Parte 3 . . . 145
109. Teoría PVS del programa para calcular los M elementos más pequeños de un arreglo - Parte 4 . . . 146
110. Teoría PVS del programa para calcular los M elementos más pequeños de un arreglo - Parte 5 . . . 147
111. Teoría PVS del programa para calcular los M elementos más pequeños de un arreglo - Parte 6 . . . 148

112. Implementación en Python del programa para calcular los M elementos más pequeños de un arreglo 149
113. Diagrama de invariantes del programa para mapear elementos entre dos arreglos 150
114. Representación textual del programa para mapear elementos entre dos arreglos - Parte 1 151
115. Representación textual del programa para mapear elementos entre dos arreglos - Parte 2 152
116. Teoría PVS del programa para mapear elementos entre dos arreglos - Parte 1 153
117. Teoría PVS del programa para mapear elementos entre dos arreglos - Parte 2 154
118. Teoría PVS del programa para mapear elementos entre dos arreglos - Parte 3 155
119. Teoría PVS del programa para mapear elementos entre dos arreglos - Parte 4 156
120. Implementación en Python del programa para mapear elementos entre dos arreglos 157
121. Diagrama de invariantes del programa para realizar el merge ordenado de dos arreglos 158
122. Representación textual del programa para realizar el merge ordenado de dos arreglos - Parte 1 159
123. Representación textual del programa para realizar el merge ordenado de dos arreglos - Parte 2 160
124. Teoría PVS del programa para realizar el merge ordenado de dos arreglos - Parte 1 161
125. Teoría PVS del programa para realizar el merge ordenado de dos arreglos - Parte 2 162
126. Teoría PVS del programa para realizar el merge ordenado de dos arreglos - Parte 3 163
127. Teoría PVS del programa para realizar el merge ordenado de dos arreglos - Parte 4 164
128. Teoría PVS del programa para realizar el merge ordenado de dos arreglos - Parte 5 165
129. Implementación en Python del programa para realizar el merge ordenado de dos arreglos 166

1. Introducción

Los objetivos de este trabajo son estudiar los aspectos prácticos de una nueva y prometedora técnica de programación denominada *programación basada en invariantes(IBP)* así como también brindar un resumen de los aportes realizados al mismo por parte del autor. La primera parte del trabajo introducirá los conceptos teóricos de IBP de manera coloquial y sin detalles formales. Luego se presentarán los aspectos prácticos de IBP mediante la exposición de la metodología de trabajo, las dificultades, problemas y limitaciones. Luego se realizará una presentación de las modificaciones llevadas a cabo sobre la herramienta que soporta IBP. Concretamente las contribuciones de este trabajo son la construcción de un conjunto de programas utilizando IBP y la implementación de determinadas funcionalidades al plug-in para la herramienta que soporta IBP denominada Socos. Finalmente serán presentadas las conclusiones obtenidas en el trabajo.

IBP está embebido en lo que se conoce como métodos formales. Los métodos formales son técnicas y métodos que permiten mediante la utilización de herramientas matemáticas especificar y definir propiedades de un sistema determinado; a su vez también permiten definir nociones como consistencia, completitud y corrección. La programación basada en invariante (IBP) son técnicas desarrolladas desde el año 80 por Ralph Johan Back. IBP permite el desarrollo y pruebas de programas de manera tal que se pueda verificar la consistencia, terminación y vitalidad (liveness) de los programas.

```
def summe(N):
        k=0
        res = 0
        while(k<N):
                res = res + k + 1
                k=k+1
        return res
```

Figura 1: Algoritmo para sumar los primeros N números naturales en Python

A modo de ejemplo tomaremos el programa para sumar los primeros N números naturales. La Figura 1 muestra el programa implementado con Python y la Figura 2 muestra el diagrama de dicho algoritmo en donde los círculos negros son el nodo inicial y el nodo final respectivamente; cada uno tiene asociadas las precondiciones y postcondiciones del programa respectivamente. Las precondiciones deben cumplirse cuando el algoritmo se inicia y las poscondiciones deben satisfacerse cuando el algoritmo finaliza. Pero también varias situaciones intermedias deberían satisfacerse para estar completamente seguros de que el comportamiento del algoritmo es el previsto. Por ejemplo, después de la primera asignación:

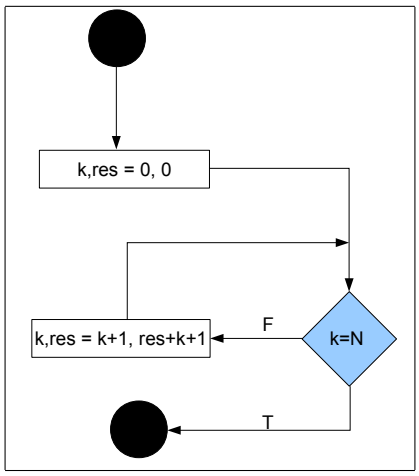

Figura 2: Diagrama de flujo del algoritmo para sumar los primeros N números naturales

$$k, res := 0, 0$$

el algoritmo está en un estado en el cual la variables k y res no pueden tener un valor distinto de cero. De la misma manera, luego de la asignación:

$$k, res := k + 1, res + k$$

k y res deben asumir determinados valores, es decir k y res poseen algunas restricciones que se deben hacer explícitas; en este caso el valor de res debe ser igual a $k(k+1)/2$ y también $k \in \{0..N\}$.

IBP fuerza al programador a hacer explícitas las restricciones de las variables implicadas en el desarrollo del programa para cada posible estado del mismo. Éstas son condiciones que el programador suele pensar mientras está programando sólo que el enfoque tradicional no brinda las herramientas para expresarlo de manera explícita. Para poner en claro lo anterior la Figura 3 muestra el mismo algoritmo utilizando el diagrama de invariantes anidado utilizado en IBP.

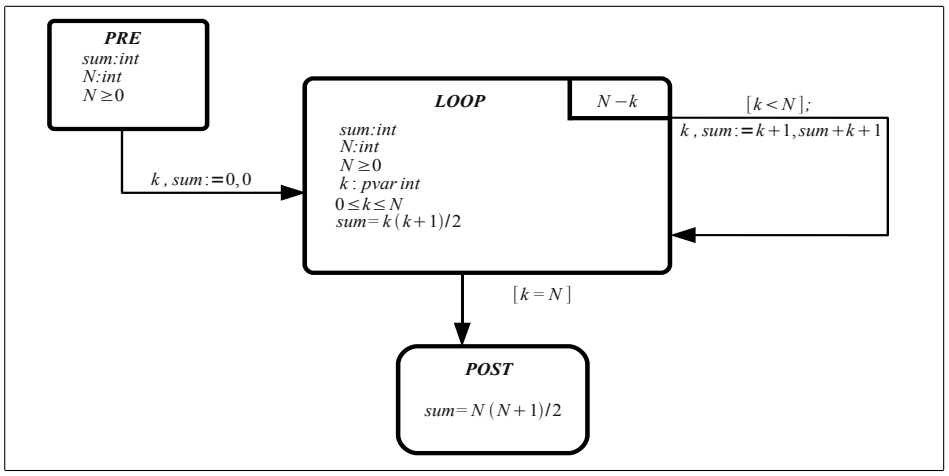

Figura 3: Diagrama de invariantes del programa para sumar los primeros N números naturales

Como se observa las condiciones se expresan fácilmente y el algoritmo no es más complejo de entender que las dos versiones anteriores. También se debe tener en cuenta que finalmente sólo las etiquetas de las transiciones se utilizarán para implementar el algoritmo en un ordenador. Las condiciones escritas dentro de los cuadrados son útiles para demostrar la corrección del programa; de hecho éstas condiciones son los invariantes que permitirán demostrar que el programa es correcto.

Por lo que se observa en el diagrama se podría pensar que IBP es dibujar programas y en cierta manera dicha idea no esta del todo desacertada. IBP se nutre de gráficos que mediante la utilización de un determinado lenguaje lógico permite la identificación de los posibles estados en los cuales un programa puede estar en determinado momento de la ejecución del mismo. Dado dicho gráfico IBP permite, define y genera las condiciones de prueba para asegurar que el programa se comporta como se supone debe funcionar, es decir es consistente, finaliza y verifica la propiedad de vitalidad (liveness). Cada uno de estos conceptos son definidos en [1] de la siguiente manera:

- *El diagrama (programa) es consistente en el sentido de que todas las condiciones de todas sus situaciones son preservadas por las transiciones.*
- *El diagrama (programa) termina, es decir no hay posibles ciclos de ejecución infinitos*
- *El diagrama (programa) no se bloquea, es decir su ejecución eventualmente alcanzará una situación final*

Es decir, *consistencia es la propiedad central requerida, lo cual significa que todas las condiciones de consistencia son satisfechas; en otras palabras cada sentencia ejecutada por el programa es válida en el sentido que si en determinado momento*

la ejecución del programa se encuentra en una situación α y la transición t está habilitada y t finaliza en una situación β entonces luego de la ejecución de las sentencias indicadas por t el estado del programa debe satisfacer las restricciones asociadas a β.

La prueba de terminación puede ser realizada de manera separada para cada situación o puede ser chequeada a la par del chequeo de consistencia. Para cada bucle de invariantes dos condiciones necesitan ser probadas: que exista una la función denominada variant que sea acotada por debajo y que la misma sea decrementada cada vez que reingresa al bucle y además no haya sido incrementada antes del reingreso.

Finalmente el programa no se queda bloqueado en alguna situación intermedia si no se da el caso que una determinada situación actual no final no posee una transición saliente habilitada.[1]

La práctica IBP define que antes del desarrollo de un programa se deben pensar las estructuras de datos que el programa utilizará y se deben realizar diagramas que reflejen la evolución y el comportamiento de dichas estructuras durante la ejecución del programa en cuestión. El paso anterior da lugar a identificar los estados en los cuales un programa puede estar. Dichos estados deben ser agrupados en conjuntos de estados a los cuales se le asociará lo que en IBP se llama situaciones. A cada situación se le asocia una o varios predicados lógicos que definen los estados que debe verificar el programa para estar en dicha situación. Luego de que se hayan identificado todas las situaciones es tiempo de relacionarlas mediantes aristas o arcos dirigidos etiquetadas con sentencias de programación. Una vez que se hayan relacionado las situaciones mediante aristas es hora de realizar las comprobaciones para verificar si el programa es consistente, finaliza y/o verifica la propiedad de vitalidad. Para ello IBP consta de una herramienta denominada Socos. En [7] se define a Socos de la siguiente manera: *es el prototipo que soporta IBP, fue diseñado e implementado para soportar IBP, es un prototipo para construir programas y razonar sobre su corrección. Socos soporta la metodología de programación IBP brindando un ambiente diagramático para especificar, implementar, verificar y ejecutar programas procedurales. Los invariantes y contratos (pre y post condiciones) pueden ser evaluadas en tiempo de ejecución siguiendo el paradigma Design by Contract. Socos también puede generar las condiciones de corrección para la verificación estática del programa por medio de la semántica de Weakest Precondition. Además Socos aplica la estrategia de PVS denominada* endgame*, ésta estrategia aplica skolemization y simplificación, luego todas las definiciones relevantes del objetivo de prueba son expandidas, las fórmulas del antecedente son cargadas junto con una lista de lemas brindados de antemano y finalmente se invoca a yices [24]. Ésta táctica probará lo máximo posible aplicando la lista de lemas brindada. Las condiciones que no fueron automáticamente descargadas pueden ser probadas interactivamente utilizando PVS.*

Socos consta de dos partes: un backend y un frontend. El backend toma un archivo de texto con la especificación de un programa y genera todas las condiciones de pruebas necesarias para verificar que el programa es consistente, termina y verifica la propiedad de vitalidad (liveness). Todas las pruebas son

reflejadas en una teoría PVS, Socos no sólo genera la teoría PVS con los lemas a probar sino que intenta llevar a cabo dichas pruebas. En caso de no poder realizar dichas pruebas el usuario debe probarlas interactivamente. Por otro lado Socos cuenta con un frontend que permite dibujar los programas, dicho dibujo es traducido utilizando una sintaxis determinada a un documento de texto que luego será tomado por el compilador de Socos y generará la teoría PVS respectiva. Actualmente Socos puede ser utilizado para desarrollar programas procedurales, en las primeras etapas de desarrollo los errores simples pueden encontrarse sólo por testing; para las etapas de desarrollo más avanzadas el programador puede y debe probar utilizando razonamiento formal que el programa es un programa libre de errores.

Cabe aclarar que tanto el frontend como el backend estaba en proceso de desarrollo al momento en que se desarrollo el trabajo que dio origen a esta tesis. Los aportes del trabajo enfocan a ambos componentes de maneras distinta. Por un lado y con respecto al compilador se construyeron un conjunto de programas de manera textual – es decir sin la interfaz gráfica- que sirvieron para realizar un testing del compilador en paralelo con su desarrollo, además dichos programas pueden ser utilizados para consultas futuras. Por otro lado se implementaron algunas características y se mejoraron otras del frontend.

Modelo de ejecución de IBP

La ejecución de IBP puede ser interpretada de la siguiente manera: el estado de un programa está dado por una tupla de variables, cada variable tiene asociada un valor, dicho valor define para cada momento el estado del programa en tal momento. La tupla de variables esta compuesta por variables de entrada, variables de salida y variables del programa definidas para almacenar temporalmente valores durante la ejecución del programa, es decir el conjunto de estas variables y sus valores definen el estado del programa.

Antes de comenzar la ejecución el programa debe estar en un estado que satisfaga el predicado booleano que caracteriza a su precondición y luego de finalizar la ejecución el estado debe satisfacer al predicado booleano que define a su postcondición. De la misma manera en cada situación intermedia el programa debe satisfacer las condiciones booleanas que caracterizan a su situación actual. Suponemos que el programa se encuentra en un estado que satisface a la condición que caracteriza a la situación actual, luego la ejecución del programa debe avanzar siempre y cuando la situación actual no sea una situación final; como suponemos que el programa no se encuentra en una situación final al menos debe existir una transición con origen en la situación actual habilitada, de todas las transiciones habilitadas el programa decidirá no determinísticamente cuál tomar, luego el programa transicionará a la situación dada por la situación destino de la transición elegida, luego las variables actualizarán sus valores de acuerdo a las sentencias con las cuales estaba etiquetada la transición tomada. Una vez que las variables han sido actualizas el estado del programa deberá satisfacer las condiciones booleanas que caracterizan a esta nueva situación ac-

tual pues como programadores debemos haber probado que dicha transición es consistente. En este punto el programa está en una nueva situación actual, si la situación actual no tiene transiciones salientes entonces el programa está en una situación final, si al menos hay una transición saliente el programa continuará su ejecución como en el caso anterior. Cabe mencionar que si el programa se encuentra en una situación no final tal que exista al menos una transición saliente, una de ellas debe estar habilitada porque de lo contrario el programa estaría bloqueado y otra vez como programadores deberíamos haber probado que el programa verifica liveness para todas sus posibles situaciones.

En esta sección se han introducido las ideas detrás de IBP así como también se presento brevemente la herramienta que soporta IBP que se llama Socos. Para un completo detalle de estas ideas es recomendable leer el material bibliográfico [1], [2], [6] y [17].

2. Casos de estudio

En esta sección se brindará un enfoque práctico a la programación basada en invariantes. Se presentarán un conjunto de programas, algunos de ellos estándar o tradicionales dentro del ámbito de aprendizaje de ciencias de la computación pero que son de gran utilidad para tomar conciencia de cómo se debería trabajar con esta nueva metodología y para identificar los pasos a seguir a la hora de construir piezas de software utilizando IBP. Los siguientes son algunos de los pasos identificados y que se deberían realizar para diseñar e implementar programas utilizando IBP:

1. Dibujar figuras para ilustrar las estructuras de datos básicas, lo cual ayuda a ganar una visión general del programa y también a entenderlo.
2. Identificar las situaciones básicas del programa. En este paso deberíamos preguntarnos £cuáles son los estados principales del programa?
3. Formalizar las restricciones de cada situación utilizando algún lenguaje lógico.
4. Conectar las situaciones identificadas mediante transiciones.
5. Probar que el programa preserva consistencia, terminación y liveness, preferentemente en este orden.
6. Ajustar situaciones y/o transiciones si se identifican errores en el paso anterior y probar nuevamente las veces que sean necesarias.

De alguna manera u otra todos los pasos anteriores fueron realizados en los programas que se presentarán. Todos los programas fueron implementados utilizando Socos y aplicando la estrategia endgame para intentar descargar las pruebas automáticamente, para estos programas la estrategia *endgame* recibió una lista de lemas vacía y el comando PVS dado fue *grind*. Las siguientes subsecciones presentan cada uno de los programas implementados. Si bien existe una interfaz gráfica en desarrollo para llevar a cabo los programas se ha utilizado el modo textual de Socos, es decir se ha interactuado de manera directa con el compilador de Socos. El compilador toma una representación textual de los programas y genera una teoría de PVS que define todas las obligaciones de prueba necesarias para probar la corrección del programa, el compilador también intenta descargar automáticamente las pruebas asociadas a cada condición. Aquellas condiciones no descargadas automáticamente deben ser probadas a mano o utilizando un demostrador de teoremas interactivo que como se ha dicho es PVS.

2.1. Suma

El primer caso de estudio consiste en especificar y construir un programa basado en invariantes para calcular la suma de los primeros N números naturales. Se asume que el atributo N viene dado, luego la tarea consiste en sumar los primeros N números naturales y alojar el resultado obtenido en la variable sum, es decir se debe construir un programa cuya postcondición sea $sum = N(N+1)/2$.

Una vez planteado el problema estamos en condiciones de continuar, lo próximo que debemos pensar es que el estado del programa al principio debe verificar la condición $N \geq 0$ y por lo tanto la precondición del programa será $N \geq 0$. Se utilizará una variable de programa k que iterativamente tomará los valores desde 0 hasta N y por cada vez que la situación reentrante sea revisitada sumaremos el nuevo valor de k a sum. Inicialmente se realiza la siguiente asignación $k, sum := 0, 0$.

Para cada iteración sum contiene el resultado de la suma calculada hasta el momento el cual es $sum = k(k+1)/2$, además para cada iteración se debe verificar que la condición $0 \leq k \leq N$ sea preservada. Como hemos dicho al principio la situación inicial verifica $N \geq 0$ y al final la situación final debe verificar $sum = N(N+1)/2$. Todas estas condiciones pueden expresarse muy fácilmente mediante el diagrama de invariantes.

El diagrama de invariantes de este programa puede verse en el Apéndice A. Todo lo que hemos dicho hasta el momento puede apreciarse claramente en el diagrama. A través de dicho diagrama se pueden identificar todas las transiciones y condiciones que deben ser satisfechas por el programa

Una vez obtenido el diagrama de invariantes estamos en condiciones de generar otras vistas del programa además del diagrama, por ejemplo dado el diagrama estamos en condiciones de generar su representación textual, una vez obtenida su representación textual y por medio de Socos se puede generar su representación en PVS y todas estas representaciones son equivalentes entre sí y se las podría ver como distintas vistas del mismo programa. La implementación previa del compilador de Socos generaba dos teorías de PVS en dos archivos distintos una con la especificación y la segunda con la implementación propiamente dicha en la cual la primer teoría era incluida. Actualmente Socos genera sólo un archivo físico en el cual se incluyen y definen ambas teorías, dicho archivo también incluye la definición del contexto.

En el caso general la teoría en PVS puede generar muchos lemas y condiciones que deben ser probadas para asegurar que la solución presentada es correcta, para nuestro caso dos lemas fueron generados y todos ellos fueron probados automáticamente utilizando la estrategia *endgame* y el comando de PVS *grind*. Por lo tanto estamos convencidos que el programa es correcto y como ventaja adicional el trabajo que tuvimos que realizar para obtener las pruebas fue bastante acotado pues todas las pruebas fueron descargadas automáticamente.

En el Apéndice A se pueden observar la representación textual del programa, la teoría en PVS y un script en Python generado traduciendo el diagrama basado

en invariantes a la sintaxis de Python.

2.2. Raíz cuadrada

Este programa intentará calcular la raíz cuadrada entera de un número entero positivo utilizando sólo las operaciones de suma, resta, multiplicación y división. Para ello definimos que s es la raíz cuadrada entera de un número n si:

$$s^2 \leq n < (s+1)^2$$

El predicado asociado a la presituación será sólo el tipo de las variables implicadas en el programa, en este caso serán el tipo de n y s. El programa comienza inicializando a la variable s a 0 y luego moviéndose hacia la situación LOOP la cual debe satisfacer el siguiente invariante:

$$0 \leq s^2 \leq n$$

Se implementó una búsqueda lineal para encontrar la raíz cuadrada de n. Mientras la condición $(s+1)^2 \leq n$ sea satisfecha, s es incrementado hasta que se satisfaga la condición $(s+1)^2 > n$, una vez que dicha condición es satisfecha s es la raíz cuadrada de n de acuerdo con nuestra definición.

El diagrama de invariantes, así como la representación textual del programa, la teoría en PVS asociada al programa y un script del programa desarrollado en Python son mostrados en el Apéndice A. La teoría en PVS tiene sólo dos lemas que fueron descargados automáticamente.

2.3. Exponencial

En este caso de estudio el problema es elevar un número a la potencia e y alojar el resultado en la variable res. Dados $e \in \mathbb{N}$ y $a \in \mathbb{Z}$ se implementó un programa para calcular a^e en tiempo logarítmico, para ello se hará uso de la siguiente propiedad: $x^b = x^{\frac{b}{2}}\, x^{\frac{b}{2}}$. Se asume que e es un número natural y que a es un número entero y representan la entrada del programa.

Este programa nos permitirá hacer uso del modificador $valres$ por primera vez, éste modificador le indica al compilador que la variable o parámetro precedido por tal modificador puede ser escrito o leído; por ejemplo dada la variable de nombre $variableName$ el compilador definirá una nueva variable lógica $variableName_0$ que permite al programador escribir invariantes utilizando el valor recibido en la variable $variableName$ antes de que el programa comience su ejecución y a dicho valor se hace referencia mediante la variable $variableName_0$.

El resultado del programa será almacenado en una variable entera res, de la misma manera a la variable res se la utilizará para almacenar soluciones intermedias que deberán verificar el invariante $res\, a^e = a_0^{e_0}$. En cada iteración se

chequeará si e es *par* o *impar,* en cada caso se tomará una decisión al respecto de los valores a almacenar en cada variable y las actualizaciones correspondientes. Cuando $e = 0$ entonces $a^e = 1$, si e es impar calcular a^e es lo mismo que calcular $a\, a^{e-1}$ y por último si e es par calcular a^e es lo mismo que calcular $(a^2)^{\frac{e}{2}}$.

Las actualizaciones necesarias así como las transiciones e invariantes involucrados en el programa pueden ser expresados utilizando el diagrama de invariantes mostrado en el Apéndice A. En dicho apéndice también se pueden observar la representación textual del programa, la teoría PVS generada y una implementación del algoritmo utilizando el lenguaje de programación Python utilizando solo las sentencias de las transiciones del diagrama.

Se ha probado que cada transición preserva los invariantes, que el programa termina y que no se bloquea, para ello el compilador Socos ha generado dos condiciones de tipo (TCC) y dos lemas, las TCCs y uno de los lemas fueron descargados automáticamente mientras que el lema restante se lo probo interactivamente utilizando PVS

2.4. Factorial

El cuarto caso de estudio consiste en especificar y construir un programa basado en invariantes para calcular el factorial de un número natural N. Se asume que el atributo N está dado, es decir se supone es la entrada del programa. La tarea consiste en calcular el factorial de dicho número N y almacenarlo en la variable de salida res. Por lo tanto al finalizar el programa debe satisfacer la condición $res = N!$. El factorial de un número natural es el producto de todos los números enteros positivos menores o iguales que el número en cuestión sin incluir al cero, el factorial de cero se lo define como 1.

El predicado asociado a la precondición es $N \geq 0$. También el programa utilizará una nueva variable de programa k que iterativamente tomará los valores desde 0 hasta N inclusive, por cada iteración el valor de k es incrementado y el valor de res es actualizado con la expresión $res(k + 1)$; inicialmente se asigna $res = 1$ y $k = 0$. La situación intermedia denominada LOOP debe satisfacer el invariante $res = factor(k)$ donde la definición de $factor$ puede ser encontrada en el Apéndice A.

Todo esto que se ha dicho hasta el momento puede observarse en el diagrama de invariantes así como también en la representación textual del programa que se alojan también en el Apéndice A junto con la teoría en PVS generada y una implementación en Python del programa.

La teoría en PVS contiene todas las condiciones que se probaron para asegurar que el programa es consistente, termina y no se bloquea. El paso relativo a las pruebas fue muy simple pues Socos generó cuatro TCC y dos lemas los cuales fueron probados automáticamente utilizando la estrategia endgame.

2.5. Palíndrome

La tarea consistió en derivar un programa para chequear si un arreglo a es o no un palíndrome. Se supone que la longitud del arreglo es N, si dicha longitud es 0 o 1 el arreglo es trivialmente un palíndrome y en tal caso el programa transiciona desde la situación inicial a su situación final seteando la variable $flag$ a true para indicar que el arreglo es un palíndrome; en el caso más interesante que la longitud del arreglo sea mayor o igual a 2 se utiliza una variable de programa i que asumirá los valores desde 0 hasta $floor(N/2)$ buscando por coincidencias o discrepancias entre los elementos del arreglo en la posición i y los elementos del arreglo en la posición $N-i-1$, si una discrepancia es encontrada entonces el arreglo no es un palíndrome y el programa debe finalizar transicionando hacia la situación final seteando a la variable $flag$ a $false$, en el caso que una coincidencia sea encontrada la próxima posición debe ser chequeada incrementando el índice i, una vez que el índice i asume el valor $floor(N/2)$ tanto si N es par como si N es impar el arreglo es un palíndrome y el programa finaliza.

El diagrama de invariantes, la representación textual, la teoría en PVS generada y una implementación en Python del programa pueden verse en el Apéndice A.

2.6. Partición

El próximo caso de estudio consiste en construir un programa basado en invariantes para implementar un algoritmo de partición tal que dado un entero x y un arreglo $a[0..N-1]$ de números enteros se reordene el arreglo dividiéndolo en dos partes de manera tal que todos los elementos de la primera parte sean menores o iguales a x y todos los elementos de la segunda parte del arreglo sean mayores que x. También el algoritmo retornará la posición en que indica dónde el arreglo fue dividido, es decir el programa retornará un elemento $0 \leq index \leq N$ tal que:

- $\forall(i : below(N)) : i < index \Rightarrow a(i) \leq x$
- $\forall(i : below(N)) : index \leq i \wedge i < N \Rightarrow a(i) > x$

De hecho las condiciones anteriores son los predicados principales asociados a la postsituación del programa. El programa comienza chequeando si el número de elementos del arreglo es mayor o igual a cero, si es cero el programa finaliza pues no hay nada para hacer; en el caso más interesante que el número de elementos del arreglo a sea mayor a cero una nueva variable de programa k es inicializada por medio de la sentencia de asignación $k, index := 0, 0$, luego el programa continúa su ejecución moviéndose a la situación intermedia que será revisitada hasta que todos los elementos del arreglo sean analizados y procesados. Los invariantes que deben ser satisfechos en esta situación reentrante son los siguientes:

- $0 \leq index \wedge index \leq k \wedge k \leq N$
- $\forall(i : below(N)) : i < index \Rightarrow a(i) \leq x$
- $\forall(i : below(N)) : index \leq i \wedge i < k \Rightarrow a(i) > x$
- $\forall(i : below(N)) : k \leq i \wedge i < N \Rightarrow a_0(i) = a(i)$
- $permutation(a_0, a)$

donde:

```
permutation(a,b): bool =
          (exists (f:[index->index]):
            bijective?(f) and
            forall (i:index): b(i) = a(f(i)))
```

Se recuerda que a_0 es una variable lógica introducida por el compilador al utilizar la clase de variables *valres* que permite tanto a programadores como a diseñadores escribir invariantes haciendo referencia al valor recibido por el procedimiento antes de comenzar la ejecución del mismo.

Una vez que el programa se encuentra en la situación intermedia y se verifica que $k < N$ se presentan dos casos posibles, o el elemento en la posición k es menor o igual a x o el elemento en la posición k es más grande que x; en el primer caso el arreglo es modificado intercambiando los elementos de las posiciones *index* y k respectivamente e incrementando ambas variables; en el caso restante sólo el índice k es incrementado sin que el arreglo sufra modificaciones. Finalmente el programa finaliza cuando se establece a *true* la condición $k = N$.

El diagrama de invariantes, la representación textual del programa, la teoría en PVS y una implementación en Python de este programa se puede observar en el Apéndice A. La teoría en PVS tiene todas las condiciones de prueba para asegurar que el programa es consistente, termina y no se bloquea, dicha teoría está compuesta por 2 lemas y ha generado 5 TCC, todas las TCC fueron descargadas automáticamente, mientras que los dos lemas no pudieron ser verificados automáticamente y debieron ser probados interactivamente a través de comandos de PVS.

2.7. Búsqueda lineal

El algoritmo de búsqueda lineal busca un valor particular en un arreglo $a[0..N-1]$ chequeando los elementos uno por uno hasta que el elemento es encontrado o no hay más elementos a chequear. Éste caso de estudio requiere especificar y construir un programa basado en invariantes para encontrar la primera posición de un elemento dado x dentro del arreglo a o que devuelva la longitud del arreglo si el elemento no está presente en el arreglo. Se asume que el atributo x está dado. El arreglo a es escaneado desde su posición inicial hasta su posición final o hasta que la posición de la primer ocurrencia del elemento x dentro de arreglo a es encontrada. Si el arreglo posee múltiples ocurrencias del

elemento dado el programa sólo retornará el índice de la primer coincidencia, ésta es una decisión de diseño que se la debe tomar de antemano. Si el elemento a buscar no está presente en el arreglo el programa retornará N que es el número de elementos del arreglo. Una variable de programa $s : below(N + 1)$ será utilizada para almacenar la posición actual que está siendo escaneada, s asumirá iterativamente los valores desde 0 hasta N. En otras palabras se necesita construir un programa con los siguientes objetivos:

1. El elemento no está presente en el arreglo
   - $(\forall(i : below(N)) : a(i) \neq x) \Rightarrow s = N$
2. El elemento está presente en el arreglo no vacío y el programa retorna la primer ocurrencia del elemento dentro del arreglo
   - $(\exists(i : below(N)) : a(i) = x) \Rightarrow s < N \wedge a(s) = x$
3. El arreglo es vacío y luego s -el valor de retorno del programa- asume el valor cero. Ésta condición está incluida en la condición 1.

En un principio la única información con la que se cuenta es el tipo de las variables de entrada y de salida, se cuenta con un arreglo a, el número de elementos del arreglo a y la variable de salida de tipo natural s de clase *result*. Inicialmente se setea $s := 0$, desde la situación inicial y siguiendo la transición inicial se alcanza la situación intermedia la cual será revisitada hasta que alguna de las condiciones finales sea alcanzada verificando sus postcondiciones, ésta situación reentrante debe preservar los siguientes invariantes:

- $N > 0$
- $0 \leq s \wedge s \leq N$
- $\forall(i : below(s)) : a(i) \neq x$

Todas estas condiciones son fácilmente expresadas utilizando el diagrama de invariantes mostrado en el Apéndice A. Dicho diagrama de invariantes permite obtener la representación textual del programa, la teoría PVS asociada al programa y una implementación en Python que también se pueden observar en el Apéndice A. La teoría PVS incluye todas las definiciones y define todos los lemas y condiciones que deben ser probados. La teoría incluye 4 TCC y 2 lemas, las TCC fueron probadas automáticamente, por el contrario ambos lemas fueron descargados interactivamente por medio de comandos PVS.

2.8. Búsqueda lineal en dos dimensiones

Éste caso de estudio es similar al anterior, sólo hay una pequeña diferencia, en lugar de realizar la búsqueda en un arreglo unidimensional la búsqueda debe realizarse en un arreglo bidimensional $a[0..N-1, 0..M-1]$. También hay otra diferencia más con respecto a la anterior búsqueda, en este caso interesa saber si el elemento esta presente en el arreglo o no, no se requiere conocer la posición en dónde el elemento buscado se encuentra. Por lo tanto el objetivo que se persigue es construir un programa basado en invariantes cuya postcondición sea la siguiente:

- $(\forall(k1 : below(N)), (k2 : below(M)) : a(k1)(k2) \neq x) \Rightarrow flag = false$
- $(\exists(k1 : below(N)), (k2 : below(M)) : a(k1)(k2) = x) \Rightarrow flag = true$

Una característica particular de este caso de estudio es que este es el primer caso en el que utilizaremos situaciones reentrantes anidadas. Se ha definido una situación externa que establece las condiciones con respecto a las filas del arreglo y por cada fila se definió una situación anidada a la anterior para definir los invariantes con respecto a cada elemento de la fila. La situación externa se define por medio de los siguientes invariantes:

- $0 \leq i \wedge i \leq N$
- $(\forall(k1 : below(i)), (k2 : below(M)) : a(k1)(k2) \neq x)$

Además la situación anidada más interior se define por medio de los siguientes invariantes:

- $0 \leq j \wedge j \leq M$
- $i < N$
- $(\forall(k1 : below(i+1)), (k2 : below(j)) : a(k1)(k2) \neq x)$

Por cada una de las situaciones reentrantes debemos definir una función de terminación, para la situación externa dicha función se define así $N - i$ y para la situación interna de define así $M - j$ donde i y j son variables de programa introducidas para almacenar la locación del elemento del arreglo que se está analizando actualmente.

Al comienzo el programa comienza inicializando i a 0, luego la ejecución del programa transiciona a la situación reentrante externa, una vez allí se debe decidir cómo proseguir; si $i = N$ es porque todas las filas fueron evaluadas y en conclusión el elemento no se encuentra en el arreglo, luego se deberá setear la variable de salida $flag$ a $false$ y el programa finaliza su ejecución; en el caso que $i < N$ implica que al menos una fila del arreglo no se ha analizado y para ello se inicializa la variable j a 0 y la ejecución del programa transiciona a la situación reentrante interior, en esta situación todos los elementos de la fila serán escaneados hasta que el elemento sea encontrado o hasta que no haya más

elementos en la fila actual a analizar, en el caso que el elemento sea encontrado el programa finaliza seteando la variable de salida $flag$ a $true$ y en el caso que no haya más elementos en la fila actual a analizar la variable i es incrementada y la ejecución del programa continúa la búsqueda en la fila siguiente.

El diagrama de invariantes, la representación textual del programa, la teoría PVS generada por Socos son diferentes vistas del mismo programa y se las puede observar en el Apéndice A. La teoría PVS generó 9 TCC que fueron descargadas automáticamente y 3 lemas los cuales tuvieron que ser probados interactivamente a través de PVS. Finalmente el Apéndice A también muestra una implementación de este programa en Python utilizando sólo las sentencias con las cuales se han etiquetado las transiciones del diagrama de invariantes asociado a este caso de estudio.

2.9. Búsqueda binaria

Éste programa implementa una búsqueda binaria sobre un arreglo unidimensional ordenado $a[0..N-1]$. El programa comienza chequeando la longitud del arreglo, si el número de elementos del arreglo es cero entonces no hay nada para hacer y la variable de salida $index$ es seteada a -1 indicando que el elemento x a localizar no se encuentra en el arreglo, en el caso que el número de elementos del arreglo sea mayor que cero la ejecución del programa transiciona hacia una situación reentrante seteando las variables del programa l y r a 0 y $N-1$ respectivamente. Al final de la ejecución la variable $index$ asumirá el valor -1 si el elemento x no se encuentra en el arreglo o la posición del arreglo donde se encuentra dicho elemento. Las variables de programa l y r serán utilizadas para referenciar la posición inferior izquierda y la posición superior derecha de la porción del arreglo actualmente analizado respectivamente. En cada iteración el elemento x será comparado con el elemento en la posición $floor((l+r)/2)$, en el caso que $x \leq a(floor((l+r)/2)$ el índice l será actualizado a $floor((l+r)/2)+1$, en el caso contrario que $x > a(floor((l+r)/2)$ el índice r será actualizado a $floor((l+r)/2)-1$. El programa procederá de esta manera hasta que el elemento x sea encontrado o los índices l y r sean iguales. Además se debe tener en cuenta algunos casos especiales como ser qué sucede cuando se verifica la condición $r = l+1$, en dicho caso el elemento $a(l)$ y $a(r)$ son ambos comparados con x para comprobar si el elemento en cuestión está o no presente en el arreglo.

La postcondición del programa debe verificar los siguientes invariantes:

- $Sorted(a)$
- $(\forall(i : below(N)) : 0 \leq i \wedge i < N \Rightarrow a(i) \neq x) \Rightarrow index = -1$
- $(\exists(i : below(N)) : 0 \leq i \wedge i < N \Rightarrow a(i) = x) \Rightarrow (index \geq 0 \wedge index < N \wedge a(index) = x)$

Donde la definición de *Sorted* puede observarse en la representación textual del programa mostrada en el Apéndice A.

El diagrama de invariantes, la teoría PVS generada a partir de la representación textual del programa y una implementación en Python pueden observarse también en el Apéndice A. El diagrama de invariantes permite ver los invariantes que deben verificarse en cada situación así como también las transiciones que definen las sentencias del programa. Por otro lado la teoría PVS generó 7 TCC que se descargaron automáticamente y dos lemas que no fueron descargados automáticamente y se los probó interactivamente a través de PVS.

2.10. Insertion sort

Este programa ordena un arreglo manteniendo una porción del arreglo ordenado seguida por una porción del arreglo con los elementos restantes sin ordenar aún, en cada iteración se remueve un elemento de la porción no ordenada y se la inserta en la posición correcta en la porción del arreglo ordenada hasta que no queden elementos por ordenar en la porción del arreglo no ordenado.

Este problema muestra una faceta adicional de IBP, como se ha venido observando IBP puede utilizarse de manera similar a cualquier lenguaje de programación pero además IBP es una poderosa herramienta para especificar programas, es decir es de gran utilidad para escribir especificaciones que no necesariamente deban ser ejecutables. Como es sabido especificar una pieza de software o un programa no significa indicar al programador cómo implementar un determinado algoritmo o solución de software sino que el objetivo principal de una especificación es indicar como el sistema se debe comportar, luego el programador deberá poseer las capacidades necesarias para escribir el código correcto para que la pieza de software alcance el comportamiento deseado. En otras palabras IBP permite escribir cuál es el comportamiento deseado para el sistema de software en cuestión así como también escribir el código para alcanzar dicho comportamiento.

Lo dicho en el párrafo anterior lo podemos observar en este caso de estudio en la definición de *update*, la cual permite utilizar una nueva construcción cuyo accionar es la de generar dado un arreglo a y dos índices i y j tales que $j \leq i$ un nuevo arreglo tal que el nuevo arreglo será igual al arreglo original a si el índice es menor que j o mas grande que i, el nuevo arreglo alojará en la posición j el elemento $a(i)$ y para el resto de los elementos el elemento en el nuevo arreglo en una posición dada será el mismo elemento del arreglo original en la posición previa. Luego el programador deberá tener las capacidades necesarias para recibir esta especificación y generar un arreglo que satisfaga el comportamiento descripto por la definición de *update*.

La tarea de este caso de estudio consiste en generar un IBP que verifique las siguientes postcondiciones:

- $Sorted(a, 0, N)$
- $permutation(a_0, a)$

Donde la definición de *Sorted* puede ser chequeada en la representación textual de este programa mostrada en el Apéndice A. El diagrama de invariantes de este

programa está compuesto por dos situaciones anidadas, la situación externa se enfoca en el comportamiento asociado a la división del arreglo entre la porción ordenada del mismo y la porción desordenada mientras que la situación interna define los invariantes necesarios de ser verificados mientras el arreglo está siendo modificado para obtener un nuevo arreglo con un elemento más en la porción de elementos ordenados.

Al comienzo todo el arreglo se lo considera desordenado y por lo tanto la división entre la porción ordenada y desordenada es cero, esto se lo representa mediante la primera asignación $i := 0$, luego la ejecución del programa transiciona hacia la situación anidada externa, dicha situación debe satisfacer los siguientes invariantes:

- $Sorted(a, 0, i)$
- $permutation(a_0, a)$
- $0 \leq i \wedge i \leq N$

Éstos invariantes principalmente aseguran que el arreglo está ordenado hasta la posición i con elementos existentes en el arreglo recibido originalmente, también aseguran que el índice i no se encuentra fuera de rango. La función variant asociada a la situación externa es $N - i$, en esta situación el programa chequea si el índice i ha alcanzado el límite superior del arreglo, en dicho caso el arreglo está ordenado y la ejecución del programa finaliza transicionando hacia la postsituación final, en caso que aún existan elementos en la porción del arreglo no ordenados el arreglo podría no estar ordenado y el elemento actual debe ser comparado con los elementos alojados en la porción ordenada del arreglo en busca de su posición correcta, para hacer esto la variable de programa j se le asigna cero y la ejecución del programa transiciona hacia la situación reentrante interna, dicha situación debe satisfacer los siguientes invariantes además de los invariantes de la situación reentrante externa:

- $0 \leq i \wedge i \leq N$
- $0 \leq j \wedge j \leq i$
- $i < N$
- $\forall(k : below(N)) : k < j \Rightarrow a(k) < a(i)$

los cuales indican que los elementos en posiciones menores a j son menores que el elemento actual y que los índices están dentro de los rangos permitidos; ésta situación define a su función variant como $i-j$. Una vez que la posición en la cual el elemento actual debe ser alojado es encontrada se realizan las actualizaciones correspondientes y la ejecución del programa transiciona nuevamente hacia la situación reentrante exterior y el programa continua su ejecución hasta que el arreglo completo se haya ordenado.

El diagrama de invariantes asociado al IBP, la representación textual del programa, la teoría PVS generada por el compilador y una implementación en

Python son mostradas en el Apéndice A. La teoría PVS generó 15 TCC y tres lemas, las 15 TCC fueron descargadas automáticamente y los tres lemas se probaron interactivamente a través de PVS. La implementación en Python ilustra lo que se ha dicho al comienzo de este caso de estudio, como programadores se debió traducir las especificación de *update* a líneas de código Python.

2.11. Dutch flag problem

En este caso de estudio se introduce el problema de la bandera nacional Alemana (Dutch national flag problem) propuesto por E.W. Dijkstra, el problema es el siguiente, supongamos que tenemos un arreglo $a[0..N-1]$ donde cada elemento es R (red), W (white) o B (blue). De aquí en más se utilizarán la asociación 0, 1 y 2 con R, W y B respectivamente. El arreglo esta ordenado inicialmente de manera aleatoria, el problema radica en reordenar el arreglo de manera tal que los elementos del mismo color estén juntos y además los elementos rojos deben estar antes que los elementos blancos y a su vez los elementos blancos deben ir antes que los elementos azules. Aquí se presentará un programa que soluciona dicho problema con un orden de complejidad lineal.

El programa utilizará tres índices l, r y k, l apunta a la división entre los elementos rojos y los elementos blancos, r apunta a la división entre los elementos no chequeados aún y los elementos azules y k apunta al elemento actual que se le debe ubicar un lugar dentro del arreglo. k será incrementado hasta que la condición $r = k$ sea evaluada a *true*.

El programa comienza chequeando el número de elementos del arreglo, si éste es cero entonces no hay nada por hacer y el problema ya está resuelto, si el número de elementos del arreglo es mayor a cero la ejecución del programa comienza asignando cero a l, la longitud del arreglo a r y cero a k, luego la ejecución programa transiciona hacia la situación reentrante denominada *loop*. Los estados del programa asociados a esta situación loop deben satisfacer los siguientes invariantes:

- $0 \leq l \wedge l \leq k \wedge k \leq r \wedge r \leq N$
- $\forall(i : int) : 0 \leq i \wedge i < l \Rightarrow a(i) = R$
- $\forall(i : int) : l \leq i \wedge i < k \Rightarrow a(i) = W$
- $\forall(i : int) : r \leq i \wedge i < N \Rightarrow a(i) = B$
- $permutation(a_0, a)$

los cuales indican que todos los elementos en posiciones menores a l son rojos, que aquellos elementos entre l y r son blancos y aquellos entre r y k son azules. Éstos invariantes también capturan las restricciones con respecto a los índices implicados en el programa. Como toda situación reentrante ésta debe tener una función variant de terminación que en este caso es $r - k$. Una vez que la ejecución del programa se encuentra en la situación loop el programa debe

elegir qué transición seguir tomando como base el color del elemento actual $a(k)$, si es blanco el índice k es incrementado y el programa retorna a la situación loop, si $a(k)$ es rojo los elementos en las posiciones k y l serán intercambiados y los índices k y l se incrementarán antes de reingresar a la situación loop, finalmente si el elemento $a(k)$ es azul el índice r es decrementado y los elementos en las posiciones k y r serán intercambiados antes de reingresar nuevamente a la situación loop. Éste proceso continua hasta que el índice k alcanza el final del arreglo, es decir cuando $k = N$ el programa transiciona a la postsituación final que aloja los siguientes invariantes:

- $0 \leq l \wedge l \leq r \wedge r \leq N$
- $\forall(i : int) : 0 \leq i \wedge i < l \Rightarrow a(i) = R$
- $\forall(i : int) : l \leq i \wedge i < r \Rightarrow a(i) = W$
- $\forall(i : int) : r \leq i \wedge i < N \Rightarrow a(i) = B$
- $permutation(a_0, a)$

El diagrama de invariantes asociado al programa, la representación textual, la teoría PVS generada por el compilador Socos y una implementación en Python se las puede observar en el Apéndice A. La teoría PVS generó 18 TCC de las cuales 17 fueron probadas automáticamente y dos lemas que junto a la TCC no probada automáticamente fueron probados interactivamente a través de PVS.

2.12. Promedio

Dado un arreglo $a[0..N-1]$ de números reales el problema consiste en retornar un nuevo arreglo tal que el elemento en la posición i sea -1, 0 o 1 dependiendo si el elemento $a(i)$ del arreglo recibido es menor, igual o mayor al promedio del arreglo respectivamente. En este caso se construyó un IBP para resolver el problema planteado, la idea es la siguiente, primero se calculará el promedio del arreglo original, luego cada elemento del arreglo será comparado contra el promedio recientemente calculado y con base en el resultado de dicha comparación se actualizará el arreglo con -1, 0 o 1. El programa debe satisfacer las siguientes postcondiciones:

- $mean = (IF\, N = 0\, then\, 0\, ELSE\, summe(a_0, N-1)/N\, ENDIF)$
- $\forall(i : below(N)) : a_0(i) < mean \Rightarrow a(i) = -1$
- $\forall(i : below(N)) : a_0(i) = mean \Rightarrow a(i) = 0$
- $\forall(i : below(N)) : a_0(i) > mean \Rightarrow a(i) = 1$

Las cuales ponen de manera formal lo dicho en el primer párrafo y también indican que la variable de programa $mean$ debe alojar el promedio del arreglo a. El IBP se construyó utilizando principalmente dos situaciones reentrantes,

la primera situación lee cada elemento del arreglo indicado por la variable de programa k y suma dicho valor a la variable *mean*. Los invariantes asociados a ésta primera situación son:

- $0 < N$
- $1 \leq k \wedge k \leq N$
- $mean = summe(a, k-1)$
- $a = a_0$

Una vez que todos los elementos del arreglo fueron sumados la ejecución del programa transiciona hacia la segunda situación reentrante dividiendo *mean* por el número de elementos del arreglo. Ésta segunda situación chequeará cada elemento del arreglo comparándolo contra el promedio calculado, si el elemento actual es menor que el promedio al elemento en la posición actual se le asigna el valor -1, si es igual se le asigna el valor 0 y si es mayor se le asigna el valor 1. Los invariantes asociados a ésta segunda situación son:

- $0 < N$
- $0 \leq k \wedge k \leq N$;
- $\forall(i : below(k)) : i < k \wedge a_0(i) < mean \Rightarrow a(i) = -1$
- $\forall(i : below(k)) : i < k \wedge a_0(i) = mean \Rightarrow a(i) = 0$
- $\forall(i : below(k)) : i < k \wedge a_0(i) > mean \Rightarrow a(i) = 1$
- $mean = (IF\, N = 0\, then\, 0\, ELSE\, summe(a_0, N-1)/N\, ENDIF)$
- $\forall(i : \{n : nat | n \geq k \wedge n < N\}) : a_0(i) = a(i)$

Para ambas situaciones la función de terminación es $N - k$. Cuando se hayan comparado todos los elementos del arreglo contra el valor de la variable *mean* la ejecución del programa transiciona hacia la postsituación final.

El Apéndice A muestra el diagrama de invariantes, la representación textual del programa, la teoría PVS generada y una implementación en Python del programa. La teoría PVS generó 19 TCC probadas automáticamente y 3 lemas que fueron probados interactivamente a través de PVS.

2.13. Encontrar los M elementos más pequeños

Dado un arreglo de números enteros $a[0..N-1]$ se debe implementar un programa para encontrar los M elementos más pequeños del arreglo a. El programa también deberá almacenar dichos elementos en un nuevo arreglo de números enteros $b[0..M-1]$ donde $0 \leq M \leq N$. Para llevar a cabo dicha tarea el programa contará con un nuevo arreglo de booleanos $aux[0..N-1]$ el cual será de utilidad para registrar si el elemento actual ya está presente en el arreglo b, es decir si el elemento actual ya ha sido seleccionado como uno de los M menores elementos y así evitar que el mismo elemento se seleccione más de una vez.

La primera situación inicializa el arreglo aux, todos los elementos de dicho arreglo se inicializan a $false$ debido a que ninguno de los elementos del arreglo a están en el arreglo b. Una vez que el arreglo auxiliar fue inicializado la variable de programa i es seteada para comenzar el proceso principal, dicha variable asumirá los valores desde 0 hasta M, i es el contador de la cantidad de elementos más pequeños encontrados hasta el momento, por cada iteración i es incrementada y si i no ha asumido el valor M aún entonces el arreglo debe ser escaneado en búsqueda del próximo elemento a asignar en el arreglo b y para ello los índices j y min son seteados a cero antes de ingresar a la situación reentrante denominada $SCANNING$ la cual permitirá encontrar al próximo elemento; min aloja el índice del actual próximo mínimo, min es actualizada siempre que el elemento en el arreglo a en la posición j no esté en el arreglo b ($aux(j) = false$) y $a(j) < a(min)$. Cabe mencionar que el procedimiento $small$ no garantiza una inyección de a a b, sólo implementa un programa para alojar los M menores elementos del arreglo a en el arreglo b.

El Apéndice A muestra el diagrama de invariantes del programa en el cual se pueden observar claramente los invariantes de cada situación junto con sus transiciones. Los invariantes principales de la situación MMIN son:

- $\forall(i_1 : below(N)) : (\forall(k : below(i)) : b(k) \neq a(i_1)) \Rightarrow aux(i_1) = false$
  - Este invariante indica que si no hay elementos en el arreglo b igual a algún elemento del arreglo a entonces el valor para dicho elemento en el arreglo auxiliar debe ser $false$.
- $\forall(i_1 : below(N)) : aux(i_1) = true \Rightarrow (\exists(k : below(i)) : b(k) = a(i_1))$
  - Indica que si algún elemento en el arreglo auxiliar es $true$ se debe a que existe un elemento en el arreglo b que es igual a algún elemento del arreglo a.
- $\forall(k : below(i)) : (\forall(j : below(N)) : b(k) = a(j))$
  - Todos los elementos del arreglo b también son elementos del arreglo a.
- $\forall(k : below(i)), (j : below(N)) : aux(j) = false \Rightarrow b(k) \leq a(j)$

- Indica que todos los elementos del arreglo a no movidos aún al arreglo b son mayores o iguales a aquellos elementos movidos al arreglo b.

- $\forall(i_1 : below(N)), (j_1 : below(N)) : (aux(i_1) = true \wedge aux(j_1) = false) \Rightarrow a(j_1) \geq a(i_1)$

  - Éste invariante fortalece al invariante anterior indicando que el valor de los elementos del arreglo a movidos al arreglo b son menores que aquellos no movidos a b.

La representación textual del programa, el diagrama de invariantes, la teoría PVS y una implementación en Python del programa se muestran en el Apéndice A. El compilador Socos al tomar la representación textual del programa ha generado una teoría PVS con 26 TCC y 4 lemas. Los cuatro lemas y una TCC fueron probados interactivamente a través de PVS; las demás TCC fueron descargadas automáticamente.

2.14. Mapeo

Éste programa aceptará tres parámetros, dos arreglos fijos $b[0..M-1]$ y $c[0..M-1]$ para machear y reemplazar y un arreglo mutable $a[0..N-1]$. Se supone que cada valor de b es único. Por cada elemento en el arreglo a el programa buscará dicho elemento en el arreglo b si el elemento es encontrado entonces la posición actual del arreglo a asumirá el valor dado por el mismo índice en dónde se lo ha encontrado pero del arreglo c, si el elemento no es encontrado en el arreglo b entonces el arreglo a en la posición actual no sufrirá modificaciones y el índice del arreglo a será incrementado; luego el proceso continuará hasta que el índice asociado al arreglo a asuma el valor N.

El programa debe satisfacer las siguientes postcondiciones:

- $\forall(i : below(N)) : (\exists(j : below(M)) : a_0(i) = b(j)) \Rightarrow (\exists(j : below(M)) : a_0(i) = b(j) \wedge a(i) = c(j))$
- $\forall(i : below(N)) : (\forall(j : below(M)) : a_0(i) \neq b(j)) \Rightarrow a(i) = a_0(i)$

Dos índices se utilizan, i asumirá los valores desde 0 hasta N, y por cada elemento $a(i)$ un nuevo índice j será utilizado para buscar el elemento $a(i)$ en el arreglo b, el índice j incrementa su valor hasta que el límite superior del arreglo es alcanzado o el elemento es encontrado; luego el arreglo mutable a es actualizado en la posición i de acuerdo a las siguientes posibilidades:

- El elemento fue encontrado

  - el arreglo a en la posición i es actualizado con el elemento del arreglo c en la posición j (la posición donde el elemento $a(i)$ fue encontrado en b).

- El elemento no fue encontrado
    - el arreglo a en la posición i no se modifica.

El programa debe satisfacer los siguientes invariantes que son similares a los invariantes asociados a la postsituación pero hasta la posición actual i:

- $\forall(i_1 : below(i)) : (\exists(j : below(M)) : a_0(i_1) = b(j)) \Rightarrow (\exists(j : below(M)) : a_0(i_1) = b(j) \wedge a(i_1) = c(j))$
- $\forall(i_1 : below(i)) : (\forall(j : below(M)) : a_0(i_1) \neq b(j)) \Rightarrow a(i_1) = a_0(i_1)$
- $\forall(i1 : below(N)) : i1 \geq i \Rightarrow a_0(i_1) = a(i_1)$

La última condición indica que los elemento en el arreglo a en posiciones mayores a i no han sido aún chequeados. Éste es otro caso en el que la introducción de la variable lógica a_0 por parte del compilador es de gran utilidad para escribir invariantes.

El diagrama de invariantes, la representación textual del programa, la teoría PVS asociada al programa y una implementación en Python del programa se muestran en el Apéndice A. La teoría PVS generó 10 TCC que se probaron automáticamente y 3 lemas que se probaron interactivamente a través de PVS.

2.15. Merge

El último caso de estudio consiste en combinar dos arreglos ordenados en orden no decreciente $a[0..N_a - 1]$ y $b[0..N_b - 1]$ y almacenar el resultado en un nuevo arreglo $c[0..N_a + N_b - 1]$ que deberá contener los mismos elementos de a y b de manera ordenada.

La solución planteada consiste en escanear a y b de izquierda a derecha dividiendo cada arreglo en una parte ya procesada y una parte aún sin procesar, en cada iteración se comparará el valor actual del índice asociado al arreglo a con el valor asociado al índice del arreglo b, si el valor del elemento de a es mayor o igual que el valor del elemento de b entonces el valor actual de b se asignará al arreglo c en la posición actual, si por el contrario el valor actual de b es mayor que el valor actual de a entonces el valor de a será asignado al arreglo c en su posición actual dada por el índice k_c.

Se deben tener en cuenta dos casos excepcionales, el primero ocurre cuando el índice asociado al arreglo a alcanza su límite superior pero el índice asociado al arreglo b aún no, en este caso el resto de los elementos de b se asignan al arreglo c en la posición $k_a + k_b$. El mismo caso sucede si el índice del arreglo b alcanza su límite superior pero el índice de a aún no, en este caso se procede de manera similar al caso previo asignando el resto de los elementos de a al arreglo c en la posición $k_a + k_b$.

Como los arreglos a y b están ordenados el arreglo resultante c estará ordenado también, es decir nuestro IBP debe tener como invariante asociado a su postsituación la condición $Sorted(c,\ N_a + N_b)$.

El Apéndice A muestra el diagrama de invariantes correspondiente a este programa en donde se utilizan algunas de las definiciones siguientes:

$sorted(L : nat, arr : parray[L, int], a : int, b : int) : bool = \forall(i : int), (j : int) : ((0 \leq a \wedge a \leq i \wedge i < j \wedge j \leq b \wedge b < L) \Rightarrow arr(i) <= arr(j))$

$merged(a : parray[N, int], k_a : nonneg\_int, b : parray[M, int], k_b : nonneg\_int, c : parray[N + M, int], k_c : nonneg\_int) : bool = \exists(f : [below[k_c] \rightarrow below[k_c]]) : bijective?[below(k_c), below(k_c)](f) \wedge (\forall(i : below(k_c)) : (k_a < N \wedge k_b < M \wedge k_c < N + M \wedge k_a = k_b + k_c \Rightarrow ((i < k_a \Rightarrow c(f(i)) = a(i)) \wedge (i \geq k_a \Rightarrow c(f(i)) = b(i - k_a)))))$

$smaller(L : nat, X : parray[N+M, int], a : int, b : int, Y : parray[L, int], c : int, d : int) : bool = \forall(i : int), (j : int) : 0 \leq a \wedge a \leq i \wedge i \leq b \wedge b \leq N+M-1 \wedge 0 \leq c \wedge c \leq j \wedge j \leq d \wedge d \leq L - 1 \Rightarrow X(i) \leq Y(j)$

$indexRestrictions(a : parray[N, int], k_a : nonneg\_int, b : parray[M, int], k_b : nonneg\_int, c : parray[N + M, int], k_c : nonneg\_int) : bool = k_a + k_b = k_c \wedge ((k_a < N \wedge k_b < M) \vee (k_a = N \wedge k_b < M) \vee (k_a < N \wedge k_b = M) \vee (k_a = N \wedge k_b = M))$

$C_1(a : parray[N, int], k_a : nonneg\_int, b : parray[M, int], k_b : nonneg\_int, c : parray[N + M, int], k_c : nonneg\_int) : bool = k_a < N \wedge k_b < M \wedge a(k_a) > b(k_b)$

$C_2(a : parray[N, int], k_a : nonneg\_int, b : parray[M, int], k_b : nonneg\_int, c : parray[N + M, int], k_c : nonneg\_int) : bool = k_a < N \wedge k_b < M \wedge a(k_a) \leq b(k_b)$

$C_3(a : parray[N, int], k_a : nonneg\_int, b : parray[M, int], k_b : nonneg\_int, c : parray[N + M, int], k_c : nonneg\_int) : bool = k_a = N \wedge k_b < M$

$C_4(a : parray[N, int], k_a : nonneg\_int, b : parray[M, int], k_b : nonneg\_int, c : parray[N + M, int], k_c : nonneg\_int) : bool = k_a < N \wedge k_b = M$

$C_5(a : parray[N, int], k_a : nonneg\_int, b : parray[M, int], k_b : nonneg\_int, c : parray[N + M, int], k_c : nonneg\_int) : bool = k_a = N \wedge k_b = M \wedge k_c = N + M$

$A_1 = c, k_a, k_b, k_c := update(c)(k_c, b(k_b)), k_a, k_b + 1, k_c + 1$

$A_2 = c, k_a, k_b, k_c := update(c)(k_c, a(k_a)), k_a + 1, k_b, k_c + 1$

$update(parray[N, int])(i : below[N], x : int) = a\ with\ [i := x]$

Junto al diagrama de invariantes se puede observar la representación textual del programa, la teoría PVS y una implementación utilizando Python del programa. La teoría PVS generó 27 TCC que fueron descargadas automáticamente y dos lemas que se probaron interactivamente a través de PVS.

Ésta sección ha mostrado un conjunto bastante amplio de ejemplos utilizando IBP. Éstos ejemplos permitieron apreciar las cualidades y habilidades implicadas en el desarrollo de programas basados en invariantes así como también permitieron describir la metodología de trabajo. Se han mostrado las diversas posibilidades ofrecidas por IBP, se ha mostrado que IBP puede ser utilizado como un lenguaje de programación así como también se ha comprobado la utilidad de IBP a la hora de especificar estructuras más complejas que requieren de habilidades superiores para realizar la traducción de la especificación propiamente dicha a un lenguaje de programación que pueda ser ejecutado, entre las habilidades superiores se podría nombrar el manejo del lenguaje de la lógica de alto orden. También se ha comprobado que IBP permite a la hora de realizar las pruebas formales de consistencia, no bloqueo y terminación que los programadores pongan atención sólo en las condiciones de pruebas más interesantes y dificultosas ya que la mayoría de los lemas triviales son descargados automáticamente por la herramienta Socos.

3. Socos: Detalles de la implementación de su interfaz gráfica

En esta sección serán presentadas las modificaciones llevadas a cabo sobre la GUI del prototipo Socos. Socos 2 tiene por objeto ser la próxima generación de la herramienta que soporta IBP, dicha herramienta se la ha pensado para que trabaje como un plug-in para Eclipse. Como la anterior versión de la herramienta la próxima está compuesta por dos aspectos principales, el compilador y la interfaz gráfica, de hecho el compilador es el core de la aplicación; tanto el compilador como la GUI fueron diseñados por Johannes Eriksson, Eriksson también ha implementado el compilador utilizando el lenguaje Python mientras que el autor de esta tesis formó parte del grupo de programadores que mediante el lenguaje JAVA y utilizando el framework Graphical Editor Framework(GEF) desarrollaron la GUI. GEF es un framework para JAVA que soporta la construcción de aplicaciones para edición de gráficos de manera tal que el sistema sea flexible y reusable, GEF implementa la arquitectura de software Model–View–Controller (MVC). Además de la arquitectura MVC los desarrolladores que utilicen GEF necesitan tener conocimiento en patrones de diseño tales como Factory, Chain of responsibility, Command, Observer y State. GEF es un framework de mucha utilidad para construir sistemas en los cuales existe un modelo compuesto de objetos que almacenan determinada información, una o varias vistas compuestas de objetos que definen figuras en la pantalla, el usuario debe estar libre de modificar tal vista mediante la realización de determinada acción sobre lo que se observa en la pantalla y el controlador define qué sucede sobre el modelo cuando la vista es modificada y viceversa. Para una información más detallada y precisa sobre GEF ver [12].

Antes de comenzar a resumir las modificaciones llevadas a cabo por el autor a la GUI de Socos GUI se explicarán los conceptos Model, View y Controller.

Model hace referencia a todos los datos he información que necesita ser modificada, persistida, mostrada y manipulada, es el bloque principal de cualquier aplicación que utilice GEF, todos los datos e información necesaria deben alojarse en alguna de las clases que implemente el modelo. El modelo no tiene referencia a la vista (View) ni al controlador (Controller), pero debe tener algún mecanismo para notificar al controlador que el modelo ha sufrido cambios lo cual se logra generalmente a través de listeners que tienen la capacidad de registrarse y desregistrarse a determinados eventos o cambios en el modelo. Por último el modelo debe proveer persistencia en el sentido que la información que posee el modelo debe poder ser almacenada en el caso que el editor sea cerrado.

La jerarquía del modelo utilizado por la GUI de Socos se muestra en la Figura 5. Dicho modelo consiste en una clase base denominada Element que representa el elemento base y básico del modelo y posee la información elemental para representar entidades dentro de la aplicación. Ésta clase es extendida utilizando diferentes niveles de especialidad:

- HeaderLabel: almacena información para ser mostrada en los encabezados

de las cajas que definen situaciones y procedimientos.

- HeaderVariant: almacena información asociada a la función variant de procedimientos y situaciones reentrantes.
- ProcedureHeader: almacena información asociada a los encabezados de las cajas de los procedimientos.
- BoxContainer: almacena información de las cajas utilizadas para contener otros elementos de tipo cajas como son BoxElement o BoxPreElement.
- BoxElement: almacena información sobre elementos de tipo caja como son procedimientos, situaciones, presituaciones y postsituaciones.
- Branch: almacena información para representar *if* o *choice*.
- Composite: almacena información de elementos compuestos como son por ejemplo las situaciones que están compuestas por un encabezado, una función variant (opcional), un elemento de tipo caja y varios invariantes.
- Invariant: almacena información sobre invariantes.
- HeaderContainer: almacena información para representar a los contenedores de los encabezados.
- InvariantContainer: almacena información para representar a los contenedores de los invariantes.
- Modules: almacena información asociada a los módulos.
- TextContainer: almacena información para representar a los contenedores de texto.
- Transition: almacena información para representar transiciones desde su punto de origen hasta su punto destino.
- TextElement: utilizada por los invariantes para representar predicados lógicos.

La vista no tiene referencias al controlador y es almacenada en una estructura de árbol para permitir que el editor pueda imprimir a los padres antes que a sus hijos atravesando dicho árbol en orden depth-first. La vista está compuesta por figuras, cada una de éstas figuras son instancias de clases que extienden alguna clase de Drawd2d -la cual provee varias figuras listas para usar basadas en el paquete gráfico SWT- La jerarquía de figuras para la GUI de Socos se muestra en la Figura 6. A continuación se resume la vista de la aplicación:

- CallBorder: representa los bordes de los elementos.
- SituationBorder: utilizada para representar los bordes de las situaciones.

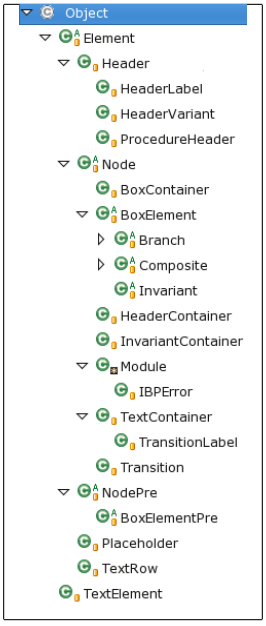

Figura 4: Clases que componen el modelo

- BoxFigure: representa cajas y es utilizada para dibujar procedimientos, situaciones, presituaciones y postsituaciones.
- BranchFigure: representa las figuras para dibujar diagramas de tipo *if*, *call* o *choice*.
- HeaderFigure: representa las figuras para dibujar encabezados.
- InvariantContainerFigure: representa las figuras para dibujar contenedores de invariantes
- EditableLabel: representa etiquetas editables y son utilizadas para dibujar el lugar donde el usuario podrá tipear invariantes, definiciones de procedimientos, nombres de situaciones, etc.
- ProcedureHeaderFigure: representa las figuras utilizadas para dibujar los encabezados de los procedimientos.
- ContainerFigure: representa las figuras utilizadas para dibujar elementos contenedores de otros elementos.
- CompositeFigure: representa las figuras utilizadas para dibujar elementos compuestos.
- TransitionLabelFigure: representa las figuras utilizadas para dibujar etiquetas de transiciones.

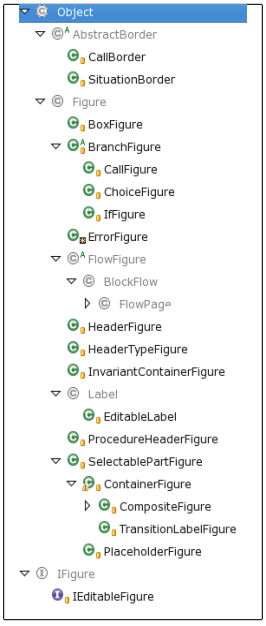

Figura 5: Clases que componen la vista

Finalmente el controlador es el nexo entre el modelo y la vista, como se ha dicho el modelo no tiene referencias de la vista y la vista no tiene referencias del modelo pero el controlador tiene referencias hacia ambos y por lo tanto el controlador es el responsable de actualizar la vista cuando el modelo es modificado y también es el responsable de modificar el modelo si la vista sufre algún tipo de modificación por alguna acción realizada por el usuario. La Figura 6 muestra las partes más relevantes de la jerarquía del controlador de la aplicación el cual está compuesto por las siguientes clases:

- TransitionEditPart: es el responsable de actualizar el aspecto del modelo inherente a las transiciones así como también modificar los dibujos de las transiciones. Por ejemplo cuando el usuario mueve las coordenadas de algún punto de quiebre de una determinada transición el modelo debe actualizar la posición del punto recientemente movido y la transición debe ser redibujada para reflejar el cambio.

- BoxContainerPart: es el responsable de actualizar el aspecto del modelo inherente a las cajas contenedoras así como también dibujar las figuras asociadas a las cajas contenedoras. Por ejemplo cuando el usuario mueve una situación a una nueva posición el modelo de la caja contenedora de la situación debe ser modificado para reflejar la nueva posición y la caja contenedora debe ser redibujada.

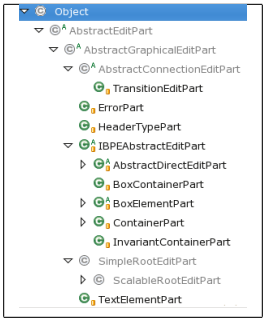

Figura 6: Clases que componen al controlador

- BoxElementPart: cuando el usuario modifica las dimensiones de una caja ésta clase modifica el modelo de la clase y redibuja el diagrama de la clase para capturar el cambio realizado.

- ContainerPart: es responsable de llevar a cabo los cambios realizados sobre contenedores. Por ejemplo cuando una nueva situación anidada es agregada a una situación existente el modelo del contenedor de la situación existente debe ser actualizado para reflejar la presencia de la nueva situación y el diagrama también debe ser redibujado para capturar la presencia de la nueva situación.

- InvariantContainerPart: es el responsable de llevar a cabo las modificaciones realizadas sobre los contenedores de invariantes. Por ejemplo al agregar un invariante a una situación el modelo del contenedor de invariantes asociado a la situación debe ser modificado para reflejar la presencia del nuevo invariante y el diagrama de la situación debe ser redibujado incluyendo el nuevo invariante.

- TextElementPart: responsable de llevar a cabo modificaciones sobre los elementos de texto. Por ejemplo cuando el texto de un invariante es modificado el modelo del texto asociado al invariante se debe actualizar reflejando el nuevo texto y el diagrama del invariante debe ser redibujado para reflejar el nuevo texto.

Además de los conceptos de modelo, vista y controlador muchos otros conceptos están involucrados en la implementación de una interfaz gráfica utilizando GEF como ser peticiones (requests), comandos (commands), acciones (actions) y políticas (polices). Para un estudio más profundo de estos conceptos chequear [11] y [12].

Ahora se está en condiciones de resumir las modificaciones realizadas al editor gráfico de Socos.

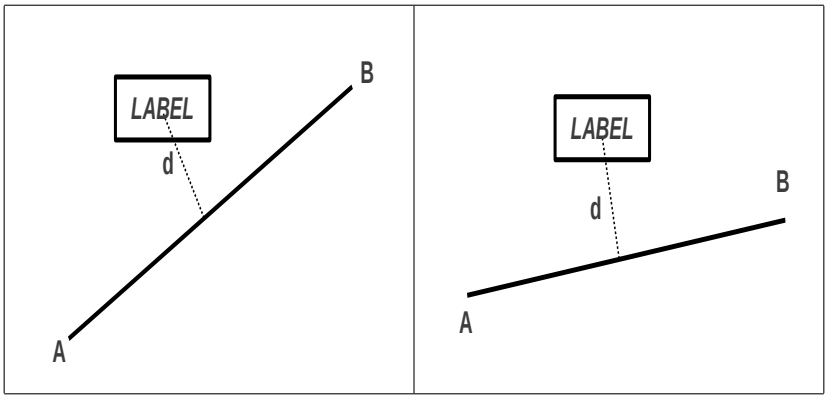

Figura 7: Movimiento de un segmento de una transición

Transiciones

Las transiciones no fueron completamente implementadas en la versión anterior, tenían la habilidad de mover sus puntos de inflexión así como también sus puntos de origen y destino pero no realizaban un movimiento automático de la posición de la etiqueta asociada a la transición, luego el objetivo en este aspecto de la implementación fue dotar al movimiento de los puntos de inflexión de las transiciones (también puntos de origen y destino) de movimiento automático y conjunto de la posición de la etiqueta.

Para tener una idea del algoritmo implementado considere la Figura 7. Se supone el segmento AB es un segmento parte de una transición y se supone también que d es la distancia más corta desde el segmento a la etiqueta asociada a la transición, luego se desea mover el punto de inflexión B a una nueva posición; el algoritmo debe actualizar la posición de la etiqueta a una nueva posición de manera tal que la nueva posición verifique que la distancia d no se haya modificado.

La clase para las etiquetas fue modificada así como la clase que implementa la localización de las etiquetas. Algunas de tales modificaciones pueden observarse en la Figura 8.

En la implementación previa la clase LabelLocator extendía la clase ConnectionLocator la cual reposiciona una determinada figura ligada a un objeto de tipo Connection con respecto a dicho objeto cuando el objeto es movido. Se reemplazó ConnectionLocator por AbstractLocator pues ésta clase posiciona una figura con respecto a un punto relativo definido por la clase hija, en nuestro caso dicho punto se lo implementó mediante el método getReferencePoint mostrado en la Figura 10.

Luego de haber implementado el movimiento de la etiqueta fue el turno de implementar el algoritmo para mover los puntos de inflexión de las transiciones y en el caso necesario actualizar la posición de la etiqueta. El algoritmo propuesto es bastante simple de entender, cuando un punto de inflexión es movido -se

```
public class LabelLocator extends AbstractLocator{ //
    extends ConnectionLocator {

        private Point position;
        private Point origo;
        Connection connection;

        public Point getPosition() {
                return position;
        }

        public void setPosition(Point position) {
                this.position = position;
        }

        public LabelLocator(Connection connection) {
                this.connection = connection;
        }

        public LabelLocator(Connection connection, Point
            orig) {
                origo = orig;
                this.connection = connection;
        }

        public Point getLocation() {
                return position;
        }
```

Figura 8: Definición de la clase LabelLocator

```
public class LabelLocator extends AbstractLocator{ //
    extends ConnectionLocator {

        private Point position;
        private Point origo;
        Connection connection;

        public Point getPosition() {
                return position;
        }

        public void setPosition(Point position) {
                this.position = position;
        }

        public LabelLocator(Connection connection) {
                this.connection = connection;
        }

        public LabelLocator(Connection connection, Point
            orig) {
                origo = orig;
                this.connection = connection;
        }

        public Point getLocation() {
                return position;
        }
```

Figura 9: Definición de la clase LabelLocator

```
protected Point getReferencePoint() {
        Point p = getLocation(connection.getPoints());
        connection.translateToAbsolute(p);
        p.x = position.x + origo.x;
        p.y = position.y + origo.y;
        connection.translateToAbsolute(p);
        return p;
}
```

Figura 10: Método ReferencePoint de la clase LabelLocator

llamara A y A' a la vieja y nueva posición respectivamente- se debe calcular el segmento más cercano de la transición a la etiqueta y si algún extremo de éste segmento es el punto que se esta moviendo entonces la posición de la etiqueta es modificada de acuerdo al algoritmo mostrado en las Figura 11 y Figura 12.

El comportamiento del algoritmo es el siguiente, cuando un segmento es movido el algoritmo encuentra el vector desde la posición de la etiqueta hasta el punto más cercano del segmento más cercano a dicha etiqueta, luego la posición de la etiqueta es movida al nuevo punto manteniendo el módulo del vector constante y el punto más cercano del segmento más cercano a la etiqueta es movido dentro del segmento de manera proporcional al movimiento del punto de inflexión.

Además un detalle adicional fue resuelto, cuando una situación era movida todas las situaciones contenidas en ella y las transiciones con origen y/o destino que son descendientes de la situación deben moverse. Los descendientes de una situación son todas las situaciones que están enmarcadas dentro de la situación actual. En la implementación previa los puntos de inflexión de aquellas transiciones cuyos origen y/o destino eran descendientes de la situación actual no se movían al moverse la situación.

Redimensionar el diagrama de las situaciones

En la implementación previa cuando la longitud de los invariantes crecían hacia la derecha superando el ancho de su situación asociada el ancho de las situaciones no se modificaba automáticamente. Se ha agregado está modificación. La implementación anterior lucía como se observa en la Figura 13.

Lo que se observa es que el invariante es mas grande que el ancho de la situación, esto se modificó permitiendo que el ancho de las situaciones crezcan a medida que los invariantes aumenten su longitud propagando la longitud de los invariantes al ancho de la situación.

```
        public Point getNewLabelPosition(Point A,Point B,
            Point Bprime){
                float oslen,nslen,cosine,sine = 0,newx,newy;
                Point oldmid = new Point(), newmid = new
                    Point(), os = new Point(), ns = new Point
                    (), perp = new Point(),ot = new Point(),p
                     = new Point();

                float ratio;

                Point s = A;
                Point e = B;

                int px = labelPoint.x;
                int py = labelPoint.y + 130;

                ratio=(float)( (e.x-s.x) * (px-s.x ) + (e.y-
                    s.y) * (py-s.y) ) / (float)( (e.x -s.x)*(
                    e.x -s.x) + (e.y-s.y)*(e.y-s.y) );
                ratio = (float) Math.min(Math.max(0.0,ratio)
                    ,1.0);

                oldmid.x= (int) (A.x+(B.x-A.x)*ratio);
                oldmid.y= (int) (A.y+(B.y-A.y)*ratio);

                newmid.x= (int) (A.x+(Bprime.x-A.x)*ratio);
                newmid.y= (int) (A.y+(Bprime.y-A.y)*ratio);

                ot.x=labelPoint.x-oldmid.x;
                ot.y=labelPoint.y-oldmid.y;

                os.x=B.x-A.x;
                os.y=B.y-A.y;

                ns.x=Bprime.x-A.x;
                ns.y=Bprime.y-A.y;
```

Figura 11: Método que calcula la nueva posición de la etiqueta - Parte 1

```
                oslen = (float) Math.sqrt((os.x*os.x+os.y*os
                    .y));
                nslen = (float) Math.sqrt((ns.x*ns.x+ns.y*ns
                    .y));

                cosine = (float)(os.x*ns.x + os.y*ns.y) / (
                    float)(oslen * nslen);
                cosine = (float) Math.max(Math.min(cosine,
                    1.0), -1.0);
                sine = (float) Math.sqrt(1.0-cosine*cosine);

                perp.x=-os.y;
                perp.y=os.x;

                if (perp.x*ns.x + perp.y*ns.y < 0.0)
                        sine=-sine;

                newx= newmid.x + cosine*ot.x - sine*ot.y;
                newy = newmid.y + sine*ot.x + cosine*ot.y;

                p.setLocation((int)newx,(int)newy);

                return p;
        }
```

Figura 12: Método que calcula la nueva posición de la etiqueta - Parte 2

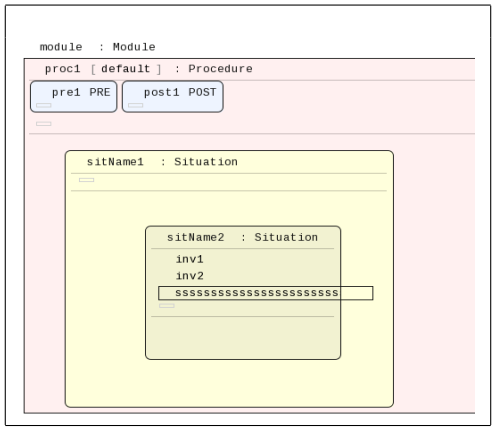

Figura 13: Snapshot del la GUI de Socos: La longitud del invariante es más grande que el ancho de la situación.

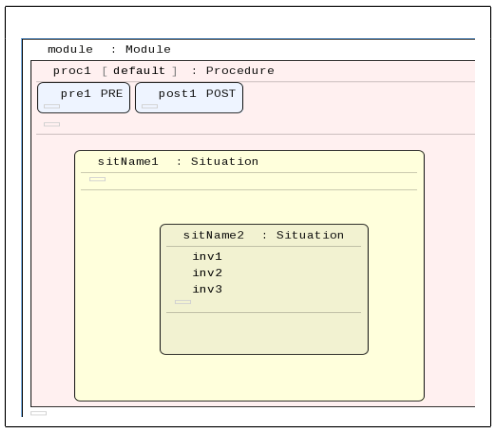

Figura 14: Socos GUI agregando el tipo de los objetos a los identificadores de los objetos.

Eliminación de nombres innecesarios

Las cajas para módulos, procedimientos y situaciones en la implementación anterior tenían la siguiente sintaxis:

$nameModule : Module$
$nameProcedure : Procedure$
$nameSituation : Situation$

para módulos, procedimientos y situaciones respectivamente. Lo mismo se muestra en el pequeño ejemplo mostrado en la Figura 14.

Como se ve esto es totalmente redundante pues los módulos, procedimientos y situaciones son fácilmente identificables y distinguibles por medio de sus formas y colores. La modificación radicó en dejar sin validez una clase cuya única función era la de agregar los strings *Module*, *Procedure* o *Situation* a los identificadores de módulos, procedimientos y situaciones.

Permitir el uso de carácteres especiales en los invariantes.

Los invariantes deben permitir símbolos especiales como $>, <, =$, *etc*, pero no así los identificadores de nombres. La implementación anterior no permitía escribir estos tipos de caracteres en los invariantes, lo cual era de bastante poca utilidad pues en la mayoría de los casos los invariantes utilizan este tipo de símbolos. Se modificó el código para permitir este tipo de caracteres en los invariantes no así en los identificadores. La modificación se realizó de la siguiente manera, se incluyó una clase que chequea el tipo del carácter tipeado, si el carácter es un carácter especial y se está modificando un identificador la aplicación se comporta como si no se hubiera tipeado ningún carácter, pero si

se trata de un invariante la aplicación permite la modificación de tal invariante. La clase se la puede observar en la Figura 15.

La Figura 15 muestra que tanto identificadores, números, espacios en blanco, comas, paréntesis derechos e izquierdos son permitidos en el texto de los invariantes.

Propagar el movimiento de las situaciones

Al mover una situación todas las subsituaciones y transiciones asociadas a ésta deben ser también movidas, también se deben modificar las etiquetas de tales transiciones y si el movimiento producirá un solapamiento con otra u otras situaciones el movimiento debe ser propagado hacia la o las situaciones solapadas. Ésta característica fue agregada al editor, el punto clave fue identificar todas las situaciones descendientes de la situación en cuestión y esto fue hecho agregando el código mostrado en la Figure 16 a la clase Composite que es la superclase asociada la clase que define a las situaciones.

Movimiento de pre y postsituaciones

En un principio las precondiciones y postcondiciones tenían negado el movimiento, ellas permanecían fijas en una sección fija de los procedimientos lo cual significaba una ventaja y al mismo tiempo una desventaja, la ventaja radicaba en la facilidad de encontrar la especificación del programa -entendiendo por especificación al conjunto de las precondiciones y postcondiciones- mientras que la desventaja era la incomodidad para ligar transiciones desde las situaciones del procedimiento hacia las presituaciones o postsituaciones. Junto con el diseñador de Socos se decidió hacer la GUI mas cómoda para el usuario brindando movimiento en las precondiciones y postcondiciones. De todas formas la especificación aún puede ser identificada rápida y fácilmente pues las presituaciones y postsituaciones tienen unos bordes especiales que permiten su identificación como se observa en la Figura 17.

Agregar presituaciones o postcondiciones dentro de otras situaciones no está permitido, de la misma manera tampoco se permite agregar situaciones dentro de las presituaciones o postsituaciones.

Copy y Paste

Un incómodo bug estaba relacionado con las acciones de copiar y pegar y también dicho bug se relacionaba con la inserción de nuevas situaciones utilizando el menú contextual que aparece al presionar el botón derecho del mouse. El problema del bug era que todas las nuevas situaciones eran insertadas en la esquina superior izquierda del objeto donde se presionaba el botón derecho es decir todas las nuevas situaciones por defecto se insertaban en la posición dada por Point(0,0) relativa al objeto en donde se paraba el ratón, pero lo desconcertante era que si se ingresaba nuevas situaciones mediante drag and drop el comportamiento era el esperado. Para un mejor entendimiento la Figura 18 muestra el comportamiento en la implementación previa y en la implementación actual.

```
protected static boolean fromStreamInv(CharStream input) {
                IBP2ModelLexer lexer = new IBP2ModelLexer(
                    input);
                CommonTokenStream tokenStream = new
                    CommonTokenStream(lexer);
                List<Token> tokens = tokenStream.getTokens()
                    ;

                if(tokens.size() == 0)
                        return false;
                else
                        if(tokens.size() == 1 && tokens.get
                            (0).getType() == IBP2ModelLexer.
                            WS)
                                return false;
                else{
                        Integer i=0;

                        while(i<tokens.size() &&
                        (tokens.get(i).getType()==
                            IBP2ModelLexer.ID
                        ||tokens.get(i).getType() ==
                            IBP2ModelLexer.INTEGER
                        ||tokens.get(i).getType() ==
                            IBP2ModelLexer.WS
                        ||tokens.get(i).getType() ==
                            IBP2ModelLexer.COLON
                        ||tokens.get(i).getType() ==
                            IBP2ModelLexer.LBRACKET
                        ||tokens.get(i).getType() ==
                            IBP2ModelLexer.RBRACKET))
                                        i++;

                        if(i==tokens.size())
                                return true;
                        else
                                return false;
                }
}
```

Figura 15: Vista parcial del código que permite caracteres especiales en el texto de los invariantes.

```
public Collection<Composite> getDescendants(){
        Collection<Composite> listRet = new HashSet<
            Composite>();
        Collection<Composite> listAux = new HashSet<
            Composite>();

        for (Element e : this.getBoxContainer().
            getStatementArray())
                if (e instanceof Composite){
                        listRet.add((Composite)e);
                        listAux =((Composite)e).
                            getDescendants();
                        listRet.addAll(listAux);
                }

        return listRet;
}
```

Figura 16: Método *getDescendants* que retorna todos los objetos anidados de un objeto de tipo caja.

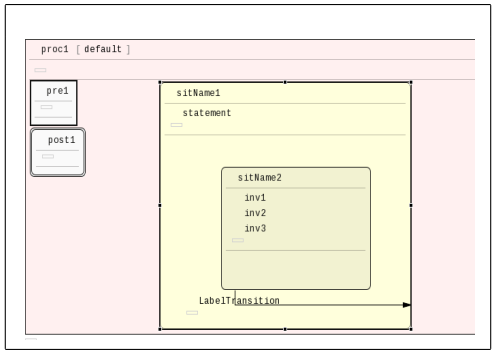

Figura 17: El área de trabajo de Socos: Un borde fue agregado a las figuras de las presituaciones y postsituaciones.

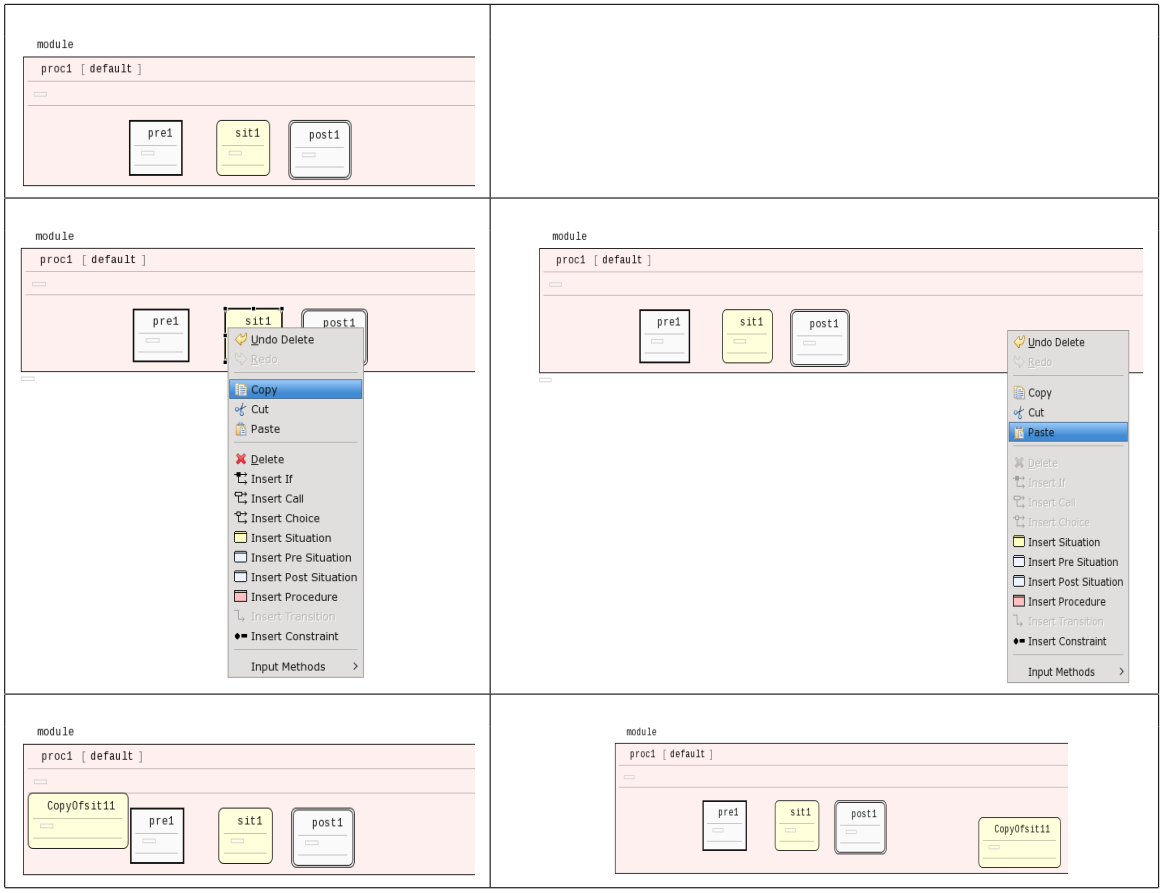

Figura 18: Comportamiento previo y actual de la acción de copy y paste.

La primera figura muestra el diagrama original, la situación *sit*1 será copiada, luego el usuario presiona el botón derecho y se hace visible el menú contextual y el usuario presiona Copy, luego el usuario posiciona el cursor en el lugar donde él quiera pegar la situación y presiona nuevamente el botón derecho, en el menú contextual en este caso selecciona Paste y finalmente las últimas dos figuras muestran los resultados alcanzados en la implementación previa y en la implementación actual respectivamente.

Para manejar y solucionar éste bug se insertó en el editor un listener para el menú contextual con el fin de capturar la posición donde el usuario presionó el botón derecho, luego el listener captura el evento y envía dicha posición a las acciones responsables de generar los comandos que ejecutarán las operaciones para pegar e insertar. La figura 19 muestra las líneas de código agregadas.

Dicho código básicamente agrega a la colección de listeners del editor un nuevo listener que será notificado cuando ocurra un trigger de menú contextual y cuando dicho evento sea detectado la posición será capturada y enviada a la acción para insertar así como también a la acción para pegar responsables de generar y ejecutar los comandos de inserción y pegado respectivamente.

Comunicación con el compilador

El editor permite dibujar diagramas de invariantes utilizando su GUI, la GUI genera la representación textual del diagrama pero en la implementación anterior no permitía la comunicación con el compilador de Socos. Ésta última característica fue implementada para la implementación actual. El compilador de Socos es un programa Python por lo tanto se tuvo que agregar el comando Check que permita la comunicación entre Java y Python, dicho comando toma la representación textual del programa y ejecuta el compilador desde un directorio cuyo path se setea desde las preferencias de Eclipse, luego de la ejecución del compilador el comando chequea los resultados obtenidos e informa al usuario los mismos. También se definió la acción Check mostrada en la Figura 21 que se tuvo que registrar en el editor a través del código mostrado en la Figura 22. El último método *createCheckCommand* será ejecutado cuando un chequeo sea requerido, el mismo crea un comando Check y setea el contexto al contexto actual.

Cuando el editor sea iniciado la acción Check será registrada y así cuando un usuario considere necesario chequear un programa dicha acción creará y ejecutará el comando Check que ejecutará el compilador de Socos. La comunicación con el compilador de Socos fue realizada como lo muestra el código de la Figura 23.

Primero se necesitó una interface entre la aplicación y el entorno en el que la aplicación se ejecuta, esto se hizo utilizando una instancia de la clase Runtime, se obtuvo el runtime actual por medio del método getRuntime() y luego se definió el proceso para ejecutar el compilador. En este caso se ejecuta el compilador utilizando la opción *–format-ibpe* que brinda una salida más fácilmente parseable. El path se obtiene desde las preferencias definidas y el nombre del archivo utilizado el nombre del archivo actual. La última sentencia toma la sali-

```
        viewer.getControl().addMenuDetectListener(new
            MenuDetectListener() {
        public void menuDetected(MenuDetectEvent e) {
                org.eclipse.swt.graphics.Point pt = viewer.
                    getControl().toControl(e.x, e.y);
                        InsertAction action = (InsertAction)
                            (getActionRegistry().getAction("
                            NEW_SITUATION"));
                        pt = viewer.getControl().toControl(e
                            .x, e.y);
                        action.setMouseClickPosition(new
                            Point(pt.x,pt.y));

                                PasteAction action2 = (
                                    PasteAction)(
                                    getActionRegistry().
                                    getAction("paste"));
                                action2.
                                    setMouseClickPosition(new
                                     Point(pt.x,pt.y));

                                InsertAction action3 = (
                                    InsertAction)(
                                    getActionRegistry().
                                    getAction("
                                    NEW_PRESITUATION"));
                                pt = viewer.getControl().
                                    toControl(e.x, e.y);
                                action3.
                                    setMouseClickPosition(new
                                     Point(pt.x,pt.y));

                                InsertAction action4 = (
                                    InsertAction)(
                                    getActionRegistry().
                                    getAction("
                                    NEW_POSTSITUATION"));
                                pt = viewer.getControl().
                                    toControl(e.x, e.y);
                                action4.
                                    setMouseClickPosition(new
                                     Point(pt.x,pt.y));
                }
            });
```

Figura 19: Código agregado al editor.

da arrojada por el compilador y setea todo lo necesario para parsear el resultado y transmitir tales resultados al usuario.

Abrir archivos PVS

Se ha agregado también la posibilidad de abrir archivos de PVS directamente desde la GUI. Si la teoría PVS aún no se ha generado un mensaje de notificación aparecerá notificando dicha situación. De la misma manera que se hizo con el comando Check se definió una acción para abrir archivos PVS y se la registró en el Editor, las Figura 24 y Figura 25 muestran lo dicho anteriormente.

Una vez que el usuario requiera abrir la teoría PVS, si el archivo existe, la teoría es abierta; si el archivo no existe aún un mensaje notificará tal situación. La Figura 26 muestra el código asociado a la ejecución de este comando.

Primero obtenemos el runtime actual, luego se chequea si el archivo existe, en caso negativo un mensaje es mostrado en la pantalla, caso contrario se produce la apertura del archivo.

```
public class CheckAction extends SelectionAction {
        protected String request;
        protected IWorkbenchPart partCheck;

        public CheckAction(IWorkbenchPart part, String id,
            String desc, String icon, String request)    {
                super(part);
                this.partCheck=part;
                this.request = request;
                setId(id);
                setText(desc);
                setToolTipText(desc);
                setEnabled(false);
                setLazyEnablementCalculation(false);
                ImageDescriptor img = AbstractUIPlugin.
                    imageDescriptorFromPlugin("fi.abo.crest.
                    IBPE", icon);            if (img != null)
                        setImageDescriptor(img);
        }

        @Override
        protected void init()
        {
                super.init();
                setText("Check");
                setId("Check");
                ISharedImages sharedImages = PlatformUI.
                    getWorkbench().getSharedImages();
                setHoverImageDescriptor(sharedImages.
                    getImageDescriptor(ISharedImages.
                    IMG_TOOL_COPY));
                setImageDescriptor(sharedImages.
                    getImageDescriptor(ISharedImages.
                    IMG_TOOL_COPY));
                    setDisabledImageDescriptor(sharedImages.
                    getImageDescriptor(ISharedImages.
                    IMG_TOOL_COPY_DISABLED));
                setEnabled(false);
        }

        protected Command createCheckCommand(List<EditPart>
            selectedObjects)
        {
                CheckCommand cmd = new CheckCommand();
                cmd.setModule(((IBPEditor)partCheck).
                    getModule());
                return cmd;
        }
}
```

Figura 20: Código para definir la acción para llevar a cabo el chequeo del programa

```
action = new CheckAction(this,
                "Check",
                "Check Model",
                "icons/new_situation.png",
                "check model");
registry.registerAction(action);
getStackActions().add(action.getId());
```

Figura 21: Código para registrar la acción CheckAction dentro del editor.

```
Runtime r = Runtime.getRuntime();

Pocess p = r.exec(Activator.getDefault().getPreferenceStore
    ().getString(PreferenceConstants.P_PATH) + "/Socos --
    format-ibpe " +  IBPEditor.getFileName());
p.waitFor();
InputStream is = p.getInputStream();
BufferedReader br = new BufferedReader (new
    InputStreamReader (is));
String aux = br.readLine();
```

Figura 22: Código utilizado para la comunicación entre la GUI y el compilador de Socos

Notificación de errores

Una vez establecida la comunicación entre la GUI y el compilador fue el turno de agregar un mecanismo de reporte de errores el cual fue agregado de la siguiente manera, si el programa resulta completamente probado, es decir de manera totalmente automática, un mensaje exitoso aparece en la pantalla; en el caso que el programa no haya podido ser completamente probado de manera automática el compilador retorna a la GUI un reporte conteniendo una lista de procedimientos, situaciones, transiciones y cualquier objeto implicado en las pruebas no exitosas, también es retornado un detalle de las pruebas realizadas, luego la GUI analiza el reporte obtenido y setea un indicador de color rojo en los elementos del programa implicados en las pruebas no exitosas. Los errores también serán impresos en la vista de error de Eclipse (Error View).

Los detalles de implementación más importantes de éste aspecto radicaron en cómo agregar la vista de error a la GUI y cómo analizar el reporte obtenido desde el compilador. Para la primera se tuvo que agregar la vista ErrorView en las extensiones de vista encontradas en el tag Extensions, el cual generó la clase ErrorView.java en el paquete de la aplicación relacionado con las propiedades del editor, en tal clase se tuvo que implementar el método createPartControl el cual declara una lista a la cual le agrega bordes y barras horizontales y verticales, dicha implementación se muestra en la Figura 27.

La clase también tiene una variable de estado para almacenar todos los

```
public class OpenPVSFileAction extends SelectionAction {
        protected String request;

        public OpenPVSFileAction(IWorkbenchPart part, String
             id, String desc, String icon, String request)
        {
                super(part);
                this.request = request;
                setId(id);
                setText(desc);
                setToolTipText(desc);
                setEnabled(false);
                setLazyEnablementCalculation(false);
                ImageDescriptor img = AbstractUIPlugin.
                    imageDescriptorFromPlugin("fi.abo.crest.
                    IBPE", icon);            if (img != null)
                         setImageDescriptor(img);
        }

        @Override
        protected void init()
        {
                super.init();
                setText("PVSFile");
                setId("PVSFile");
                ISharedImages sharedImages = PlatformUI.
                    getWorkbench().getSharedImages();
                setHoverImageDescriptor(sharedImages.
                    getImageDescriptor(ISharedImages.
                    IMG_TOOL_COPY));
                setImageDescriptor(sharedImages.
                    getImageDescriptor(ISharedImages.
                    IMG_TOOL_COPY));
                setDisabledImageDescriptor(sharedImages.
                    getImageDescriptor(ISharedImages.
                    IMG_TOOL_COPY_DISABLED));
                setEnabled(false);
        }

        protected Command createOpenPVSFileCommand(List<
            EditPart> selectedObjects)
        {
                OpenPVSFileCommand cmd = new
                    OpenPVSFileCommand();
                return cmd;
        }
```

Figura 23: Código para definir la acción que abre la teoría PVS asociada al programa

```
action = new OpenPVSFileAction(this,
                "PVSFile",
                "Open PVS File",
                "icons/new_situation.png",
                "open pvs file");
registry.registerAction(action);
getStackActions().add(action.getId());
```

Figura 24: Código para registrar la acción OpenPVSFileAction en el editor

```
Runtime r = Runtime.getRuntime();
String file = IBPEditor.getFileName();
int fileSize = IBPEditor.getFileName().length();

File f = new File(file.substring(0,fileSize-3) + "pvs");

if (f.exists())
        r.exec("pvs " + file.substring(0,fileSize-3) + "pvs
            ");
else{
        JFrame frame = new JFrame("WARNING!");
        JOptionPane.showMessageDialog(frame, "The .pvs file
            was not generated", "WARNING!!!",1);
}
```

Figura 25: Código para abrir la teoría PVS asociada al programa

```
public void createPartControl(Composite parent) {
        Display display = IBPEditor.shell.getDisplay();
        l1 = new List(parent, SWT.BORDER | SWT.V_SCROLL |
            SWT.H_SCROLL | SWT.BORDER_DASH);
        l1.setBackground(display.getSystemColor(SWT.
            COLOR_WIDGET_LIGHT_SHADOW));
        l1.setBounds(0,0,60,100);
}
```

Figura 26: Implementación del método createPartControl

```
public static void setErrorList(){
        StringTokenizer tokens = new StringTokenizer(error
            ,".");
        while(tokens.hasMoreTokens()){
                String strAux = tokens.nextToken();
                l1.add("ERROR " + l1.getItemCount() + "\n" +
                     strAux);
        }
}
```

Figura 27: Implementación del método *setErrorList*

errores mostrados en la vista de error. Cada vez que la vista de error es refrescada se debe invocar al método *setErrorList* que agrega los ítems necesarios a la lista $l1$. La implementación se muestra en la Figura 28.

Primero se tokeniza la cadena de caracteres $l1$ dividiendo su contenido en los puntos, luego cada subcadena de caracteres obtenida es concatenada a la palabra *"ERROR"* y agregada a la lista.

Cómo se realiza el análisis del reporte de error obtenido desde el compilador es un poco intrincado, por ésta razón y porque el tratamiento de errores está en fase de testing aún sólo se realizarán algunos comentarios al respecto. La GUI recibe desde el compilador una representación textual de todos los errores identificados por el compilador, luego la GUI leerá línea por línea dicho reporte identificando qué objetos del programa están implicados en las pruebas no exitosas, luego una indicación de los errores se encenderá en los objetos implicados. Una vez que todos los errores fueron analizados se actualizan el estado de la clase ErrorView invocando al método *setErrorList*. No sólo los errores son tratados sino que también los warnings son tenidos en cuenta. Un ejemplo del reporte de error implementado es mostrado en la Figura 29 donde se puede observar que el error se produjo en la situación $LOOP$ del procedimiento *square* (ambos destacados por un pequeño cuadrado rojo) y el error se debió a un error en la función variant que en la versión correcta debería ser $n - s * s$. Las condiciones no probadas se muestran en el tag de error.

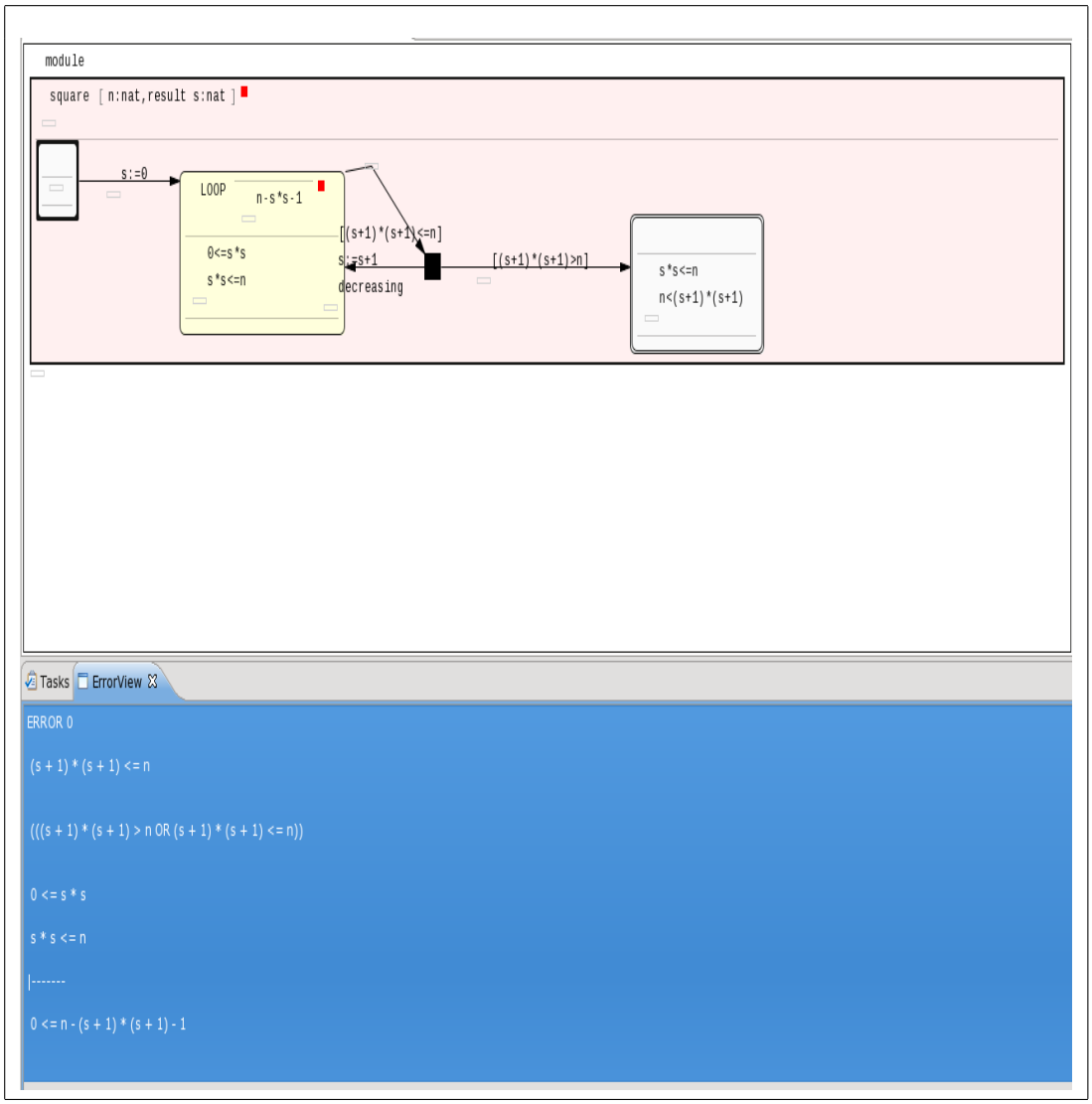

Figura 28: Ejemplo de la notificación de errores para un simple programa

Actualización de la sintaxis

Varias actualizaciones de sintaxis fueron agregadas con respecto a la sintaxis utilizada en la versión anterior. Por ejemplo la Figura 29 muestra el mismo programa presentado en la subsección 5.1 pero realizado mediante la nueva GUI y como se puede apreciar luce distinto a la versión que se muestra en el Apéndice A pero de todas formas ambas versiones son perfectamente entendidas y procesadas por el compilador. El compilador define el símbolo % para representar líneas de comentarios, todos los caracteres luego del carácter % no tendrán efecto ni validez desde el punto de vista del compilador, no así desde el punto de vista de la GUI pues se ha utilizado dicho símbolo para almacenar la información del diseño de los diagramas de invariantes y poder recargar un archivo dentro de la GUI.

La siguientes modificaciones se realizaron sobre la sintaxis:

- Invariantes:
  - La extensa y molesta declaración de invariantes mediante la palabra reservada *invariant* o mediante su versión resumida *inv* fue reemplazada por el símbolo $*$
- Función variant
  - La función variant no era identificada por ningún símbolo en la implementación previa. Se agrego el símbolo $**$ para permitir identificar la función de terminación variant
- Presituación y postsituación
  - Las presituciones y postsituaciones conforman una especificación de alto nivel del programa, y por lo tanto deben estar presentes en todos los programas, es decir no son situaciones típicas, son situaciones especiales y por tanto no pueden ni deben ser tratadas como situaciones normales. Se ha reemplazado la definición:

```
preName : PRE BEGIN %: [ 387 67 202 142 ]
             * inv1;
             * inv2;
END PRE %:
```

por:

```
%: [ 387 67 202 142 ]
             PRE inv1;
             PRE inv2;
%:
```

Modificaciones similares fueron realizadas sobre las postsituaciones. La definición de una posible postsituación era la siguiente:

```
Sum : CONTEXT BEGIN %:

    using "(endgame) grind";
    sumproc [ %:
        N:int, result sum:int
    ] : PROCEDURE %:
        %: [ -1 35 128 80 ]

        PRE N>=0;
        %:
        %: [ 1027 35 164 84 ]

            POST sum=N*(N+1)/2;
        %:
    BEGIN %:
        k:pvar int;
        LOOP : SITUATION BEGIN %: [ 387 67 202 142 ]
            * N>=0;
            * 0<=k and k<=N;
            * sum=k*(k+1)/2;
            ** N-k;
        IF %:
            %: trs [ [ 15 10 ] [ 31 25 ] ]
            [k=N];
            EXIT %:
            %: trs [ [ 14 12 ] 823 184 820 290 520 280 [ 130
                 140 ] : 744 147 ]
            [k<N];
            k,sum:=k+1,sum+k+1;
            decreasing
            GOTO LOOP %:
        ENDIF %:
        END LOOP %:

    IF %:
        %: trs [ [ 106 22 ] [ 17 22 ] : 248 99 ]
        [true];
        k,sum:=0,0;
        GOTO LOOP %:
    ENDIF %:
    END sumproc %:

END Sum %:
```

Figura 29: Sintaxis generada por la nueva GUI de Socos para el programa que suma los primeros números naturales

```
postName : POST BEGIN %: [ 387 67 202 142 ]
             * inv1;
             * inv2;
END POST %:
```

que fue reemplazada por:

```
%: [ 387 67 202 142 ]
             POST inv1;
             POST inv2;
%:
```

- Postsituación

  - Dependiendo del número de postsituaciones definidas dentro de un procedimiento la definición para tales situaciones cambia, es decir existen varias posibilidades, si se define sólo una postsituación entonces la postsituación no posee nombre y la definición es la misma utilizada en el caso anterior, pero en el supuesto caso que existan más de una postsituación cada una de ellas debe estar identificadas por un identificador único. Una posible definición sería la siguiente:

```
        %: [ 571 27 132 104 ]
                post2:
            POST inv2.1;
            POST inv2.2;
        %:
        %: [ 262 40 132 104 ]
        post1:
            POST inv1.1;
            POST inv1.2;
        %:
```

    en dónde cada postsituación se la identifica por un único nombre, en este caso *post2* y *post1*.

- Parsing

  - Un procedimiento es parseado en el siguiente orden, primero el contrato (la especificación de alto nivel dada por las presituaciones y postsituaciones), luego las situaciones comunes y finalmente las transiciones comenzando por las transiciones adyacentes a la presituación. El mismo orden es utilizado para parsear situaciones y contextos.

Preferencias

El editor tiene la posibilidad de agregar o setear preferencias para el mismo. Se han agregado preferencias para identificar archivos a importar dentro de los programas y preferencias para indicar estrategias a utilizar durante el chequeo automático de las condiciones de prueba.

El primero permite agregar una lista de teorías PVS separadas por punto y coma dentro de la especificación del programa, éstas son teorías PVS que serán incluidas en el contexto actual del IBP.

El segundo permite setear los parámetros de la estrategia endgame utilizada para descargar las condiciones de prueba de manera automática.

En el ejemplo de la Figura 30 las teorías *parray.pvs (contiene la especificación en PVS de arreglos)* y *auxiliar.pvs (contiene la especificación en PVS de definiciones útiles para el programa que se desarrollará)* serán incluidas en el contexto y la estrategia *endgame* usará el comando PVS *grind* pero otras opciones también son permitidas, dependiendo el programa que se está desarrollando algunas serán de utilidad y otras carecerán de ésta.

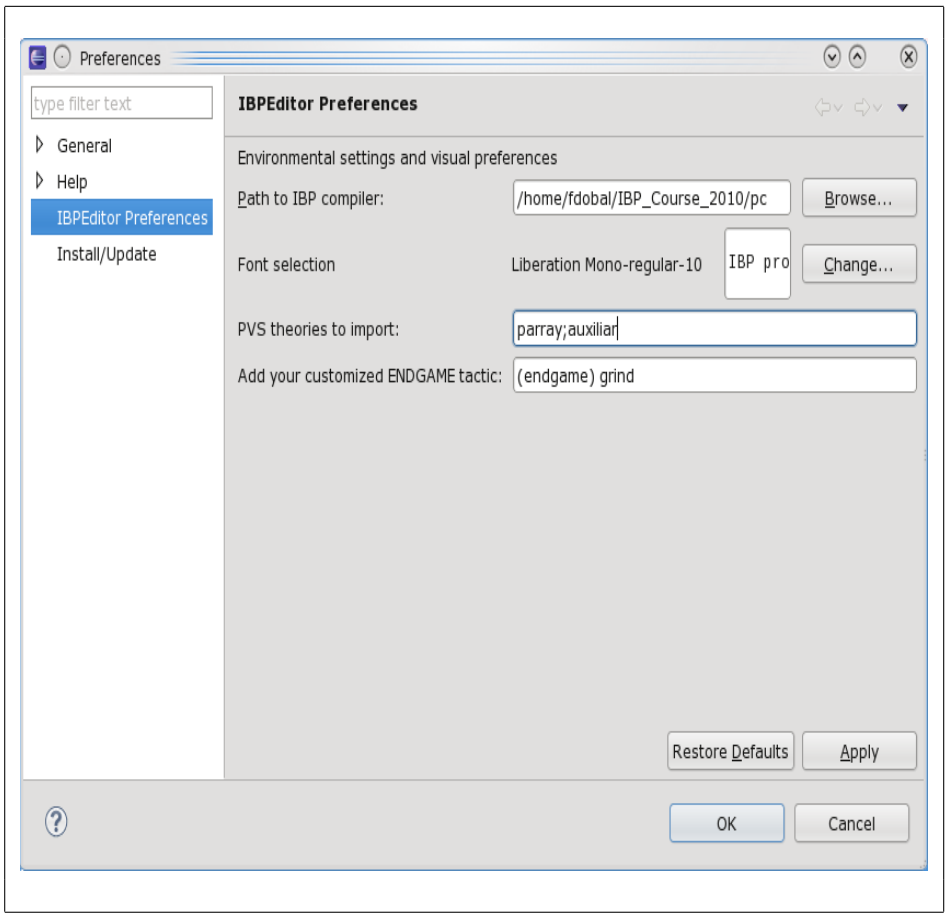

Figura 30: Ventana de preferencias

4. Conclusiones

En esta sección se presentarán las conclusiones del trabajo. Los objetivos principales del trabajo fueron exitosamente alcanzados, se ha podido introducir el marco de trabajo teórico y práctico implicado en IBP, un conjunto de programas fueron implementados utilizando IBP los cuales permitieron reflexionar sobre las fortalezas y debilidades de IBP así como también fueron de utilidad para realizar el testing del compilador de Socos, además se han mostrado las modificaciones realizadas a la GUI de la herramienta que soporta IBP.

Varios resultados han emergidos del trabajo, la primera importante e interesante observación es que las ideas detrás de IBP son fácilmente entendidas y se pueden aprender rápidamente. Luego de algunas horas cualquier entusiasta de las ciencias de la computación adquiere el conocimiento necesario para desarrollar programas utilizando IBP sin mayores inconvenientes, incluso IBP es más fácil de aprender que cualquier nuevo lenguaje de programación a raíz de sus componentes gráficos y que sólo trabaja con condiciones booleanas, transiciones, asignaciones y sentencias de composición las cuales deberían ser bien entendidas por la mayoría de los estudiantes de ciencias de la computación y programadores.

IBP involucra dos tareas principales, la identificación de invariantes y la obtención de las pruebas de las condiciones de pruebas. La primera involucra principalmente habilidades creativas, un buen diseñador debe tener esta habilidad bien desarrollada, esto es la habilidad de abstraerse, la habilidad de traducir hechos desde un lenguaje informal hacia un lenguaje formal. La segunda si bien involucra habilidades creativas también se conforma de habilidades mecánicas ya que IBP distingue el lugar dónde los bugs tienen su origen y luego el programador debe poner en funcionamiento sus habilidades creativas para identificar la razón del error y reformular las sentencias deficientes o incompletas.

IBP fuerza al programador a comprender clara y detalladamente cómo se comporta el programa, pues brinda el marco para entender las razones por la cual el programa podría no comportarse como se esperaba. IBP muestra muy claramente qué parte del programa está generando errores y así ofrece un camino hacia la solución, es decir IBP guía al programador a la región en la cual el error tiene su origen. En el proceso de comprender el comportamiento de los programas son muy importantes y casi imprescindibles los diagramas, gráficos y dibujos de la misma manera que lo son en física; IBP trae a consideración esta necesidad.

Como resultado no positivo se encuentra que tan pronto como la complejidad de los programas se incrementa los invariantes de hacen más difíciles de identificar, incluso el número de invariantes de los programas más complejos se incrementa rápidamente. Este hecho podría desalentar el intento de escalar éstas ideas a algoritmos realmente complejos o sistemas de software de complejidades elevadas, pero de todas formas la impresión es que es posible escalar estas ideas a cualquier tipo de sistema de software. Cabe resaltar que incluso para programas no muy complejos se debe contar con una teoría formal del dominio

de aplicación definida en el lenguaje que se utilizará para realizar las pruebas formales, como Socos trabaja ligado a PVS sería conveniente y recomendable contar con esta teoría en PVS pero no es obligatorio.

IBP trae una novedad importante para el testing, aunque algunos programas permiten que su espacio de estados sea dividido en clases de equivalencia no pueden ser probados completamente en el sentido tradicional debido al gran número de estados potenciales del programa, en algunos casos son infinitos, y por ende no es posible estar completamente seguros que el programa está libre de errores y que se comportará adecuadamente en todos los casos. IBP provee las herramientas para estar completamente seguros que el programa es correcto pues brinda el marco matemático para probar los programas matemáticamente y así estar completamente seguros que el programa es correcto.

También IBP hace una clara diferencia entre validación y verificación, siempre que se hayan probado todas las obligaciones de prueba de un IBP se ha probado que el programa es correcto con respecto a su especificación, y por lo tanto el proceso de verificación esta completo y correcto, luego el proceso de validación podría fallar debido a un error en la especificación.

Por otro lado este trabajo ha agregado algunas características a la GUI de la herramienta Socos:

- Transiciones: Se implementó un algoritmo para manejar el movimiento de las transiciones.
- Redimensionar los diagramas de situaciones: Se implementó un algoritmo para redimensionar los diagramas de situaciones.
- Eliminación de nombres redundantes: Se eliminaron las declaraciones redundantes de la representación textual generada por el editor de Socos.
- Permitir el uso de carácteres especiales en los invariantes: Carácteres especiales son actualmente permitidos dentro de las sentencias de invariantes.
- Propagar el movimiento de las situaciones: Se ha mejorado el movimiento de las situaciones propagando el movimiento de las situaciones externas hacia sus situaciones descendientes.
- Movimiento de presituaciones y postsituaciones: Se dotó de movimientos a las presituaciones y postsituaciones.
- Copiar y Pegar: Se mejoró las acciones de copiar y pegar de la GUI.
- Comunicación con el compilador: Se agregó e implementó la interfaz que permite la comunicación entre la GUI y el compilador de Socos.
- Apertura de archivos de PVS: Se implementó un mecanismo para permitir la apertura de archivos de PVS desde el editor.

- Notificación de errores: Un mecanismo de reporte de errores fue implementado, si bien no se lo implementó en su totalidad dicha implementación puede ser utilizada como punto de partida para implementar un mecanismo de reporte de errores completo y robusto.
- Actualización de la sintaxis: Durante la duración del trabajo la sintaxis de los programas IBP cambio algunos de sus aspectos, la GUI debió ser adaptada a tales modificaciones.
- Preferencias: Se implementaron preferencias para agregar archivos que definen teorías PVS y para pasar parámetros a la estrategia endgame.

Apéndice A

Éste apéndice muestra los diagramas de invariantes, la representación textual de los programas, las teorías PVS generadas por el compilador de Socos y una versión desarrollada en Python de cada uno de los casos de estudio presentados en la Sección 5.

Suma

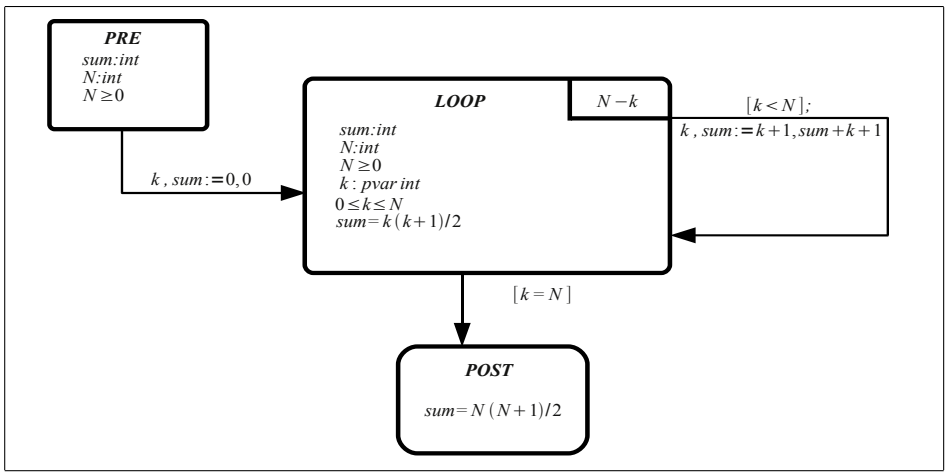

Figura 31: Diagrama de invariantes para calcular la suma de los primeros N números naturales

```
Sum: CONTEXT
BEGIN

  using "(endgame (grind))";

  sumproc [N:int, result sum:int]: PROCEDURE
    PRE N >= 0;
    POST sum=N*(N+1)/2;
    BEGIN

      k:pvar int;

      loop:SITUATION
      BEGIN
        * N>=0;
        * 0 <= k and k <= N;
        * sum=k*(k+1)/2;
        ** N - k;

        IF
          [k<N];k,sum:=k+1,sum+k+1; decreasing GOTO loop
          [k=N]; exit
        ENDIF

      END loop

    k,sum:=0,0; GOTO loop

  END  sumproc

END Sum
```

Figura 32: Representación textual del programa que suma de los primeros N números naturales

```
ctx_Sum: theory
begin
end ctx_Sum

spec_sumproc: theory
begin
    importing ctx_Sum;
    N: var int;
    sum: var int;

    pre_(N): bool =
        (N >= 0);

    post_(N, sum): bool =
        (sum = N * (N + 1) / 2);
end spec_sumproc

impl_sumproc: theory
begin
    importing spec_sumproc;
    sum: var int;

    N: int;
    k: var int;

    sit_ini_(sum, k): bool =
        ((true))
            and (spec_sumproc.pre_(N));

    sit_loop(sum, k): bool =
        N >= 0
            and (0 <= k
                 and k <= N)
            and sum = k * (k + 1) / 2;

    sit_fin_(sum, k): bool =
        (spec_sumproc.post_(N, sum));
```

Figura 33: Teoría PVS del programa que suma de los primeros N números naturales - Parte 1

```
    vc_ini_: lemma
        forall (sum, k) :
              sit_ini_(sum, k)
              => (lambda (k : int, sum : int) :
                sit_loop(sum, k))(0, 0)

    vc_loop: lemma
        forall (sum, k) :
              sit_loop(sum, k)
              => (lambda (v_loop_0 : int) :
                    ((k < N
                      or k = N))
                    and ((k < N
                          => (lambda (k : int, sum : int) :
                                (((0 <= N - k
                                   and N - k < v_loop_0)))
                                and (sit_loop(sum, k)))(k +
                                    1, sum + k + 1))
                         and
                         (k = N
                          => sit_fin_(sum, k))))(N - k)
end impl_sumproc
```

Figura 34: Teoría PVS del programa que suma de los primeros N números naturales - Parte 2

```
def sumFun(N):
  k,sum=0,0
  while (k<N):
    k,sum=k+1,sum+k+1
  return sum
```

Figura 35: Implementación en Python del programa que suma de los primeros N números naturales.

Raíz cuadrada

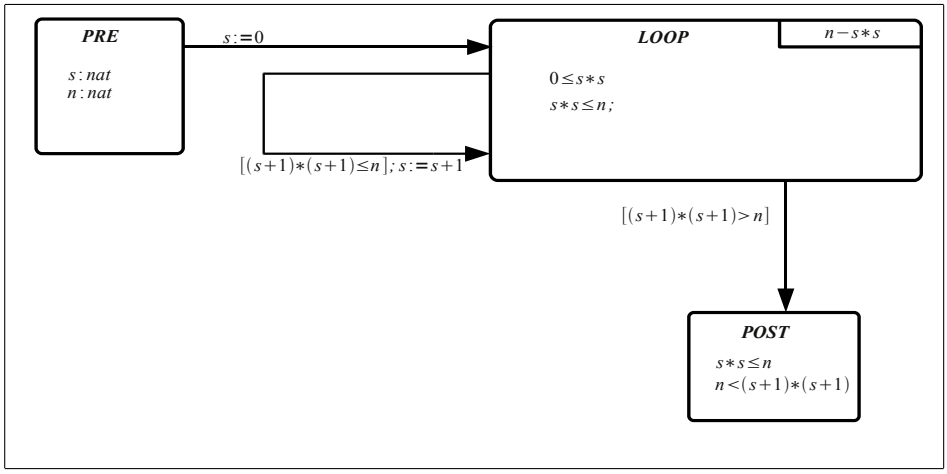

Figura 36: Diagrama de invariantes para calcular la raíz cuadrada

```
module : CONTEXT
BEGIN

  square [n:nat,result s:nat] : PROCEDURE
    POST s*s<=n;
    POST n<(s+1)*(s+1);

  BEGIN

    LOOP : SITUATION BEGIN
      * 0<=s*s ;
      * s*s<=n;
      ** n-s*s;

      IF
        [(s+1)*(s+1)>n]; EXIT
        [(s+1)*(s+1)<=n]; s:=s+1; decreasing GOTO LOOP
      ENDIF
    END LOOP

    s:=0; GOTO LOOP
  END square

END module
```

Figura 37: Representación textual del programa para calcular la raíz cuadrada

```
ctx_module: theory
begin
end ctx_module



spec_square: theory
begin
    importing ctx_module;
    n: var nat;
    s: var nat;

    pre_(n): bool =
        (true);

    post_(n, s): bool =
        s * s <= n
            and n < (s + 1) * (s + 1);
end spec_square



impl_square: theory
begin
    importing spec_square;
    s: var nat;

    n: nat;

    sit_ini_(s): bool =
        ((true))
            and (spec_square.pre_(n));

    sit_LOOP(s): bool =
        0 <= s * s
            and s * s <= n;

    sit_fin_(s): bool =
        (spec_square.post_(n, s));
    vc_ini_: lemma
        forall (s) :
              sit_ini_(s)
              => (lambda (s : nat) :
                sit_LOOP(s))(0)
```

Figura 38: Teoría PVS del programa para calcular la raíz cuadrada - Parte 1

```
    vc_LOOP: lemma
        forall (s) :
              sit_LOOP(s)
              => (lambda (v_LOOP_0 : int) :
                    (((s + 1) * (s + 1) > n
                      or (s + 1) * (s + 1) <= n))
                    and (((s + 1) * (s + 1) > n
                          => sit_fin_(s))
                         and ((s + 1) * (s + 1) <= n
                              => (lambda (s : nat) :
                                    (((0 <= n - s * s
                                       and n - s * s <
                                           v_LOOP_0)))
                                  and (sit_LOOP(s)))(s + 1))))
                                      (n - s * s)
end impl_square
```

Figura 39: Teoría PVS del programa para calcular la raíz cuadrada - Parte 2

```
def squareFun(n):
  s=0
  while (s+1)*(s+1)<=n:
    s=s+1
  return s
```

Figura 40: Implementación en Python del programa para calcular la raíz cuadrada

Exponencial

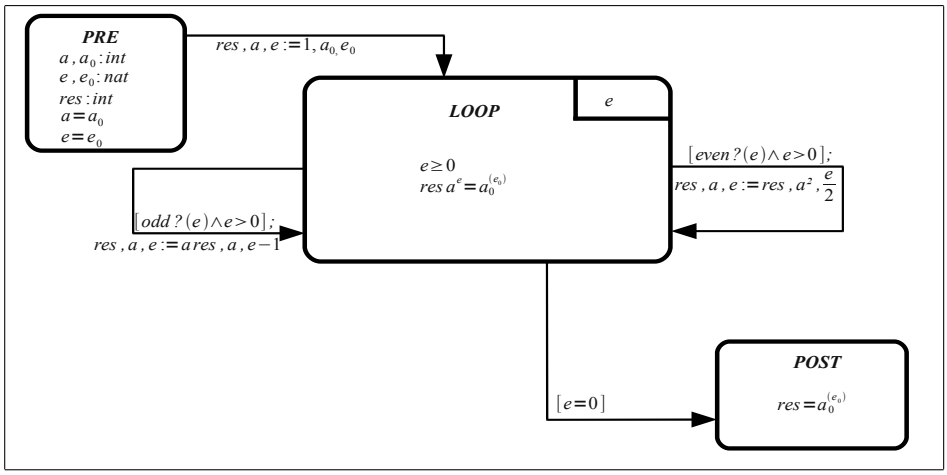

Figura 41: Diagrama de invariantes del programa para calcular potencias

```
ctx_Exp: theory
begin
    %|-*_TCC*:proof (then (skosimp*) (then (expand-defs nil
        "context[Exp]") (flatten-disjunct -) (check-report (
        endgame (grind)) "context[Exp]"))) qed
end ctx_Exp

spec_exproc: theory
begin
    importing ctx_Exp;
    a: var int;
    e: var nat;
    a_0: var int;
    e_0: var nat;
    res: var int;

    pre_(a, e): bool =
        (true);

    post_(a_0, e_0, a, e, res): bool =
        (res = expt(a_0, e_0));
    %|-*_TCC*:proof (then (skosimp*) (then (expand-defs (("
        pre_" 1) ("post_" 1)) "context[Exp].procedure[exproc
        ]") (flatten-disjunct -) (check-report (endgame (
        grind)) "context[Exp].procedure[exproc]"))) qed
end spec_exproc

impl_exproc: theory
begin
    importing spec_exproc;
    a: var int;
    e: var nat;
    res: var int;

    a_0: int;

    e_0: nat;

    sit_ini_(a, e, res): bool =
        ((a = a_0
              and e = e_0))
            and (spec_exproc.pre_(a, e));

    sit_loop(a, e, res): bool =
        e >= 0
            and res * expt(a, e) = expt(a_0, e_0);
```

Figura 42: Representación textual del programa para calcular potencias - Parte 1

```
    sit_fin_(a, e, res): bool =
        (spec_exproc.post_(a_0, e_0, a, e, res));
    vc_ini_: lemma
        forall (a, e, res) :
              sit_ini_(a, e, res)
              => (lambda (res : int, a : int, e : nat) :
                sit_loop(a, e, res))(1, a_0, e_0)
    %|-vc_ini_:proof (skolem-prefer (a e res)) (flatten-
        disjunct 1 :depth 1) (expand sit_ini_) (flatten-
        disjunct -1 :depth 1) (replace-equalities 2) (beta 1)
         (then (expand-defs (("spec_exproc.pre_" 1 (("
        sit_ini_" 1))) ("sit_loop" 2) ("spec_exproc.post_" 1
        (("sit_fin_" 1)))) "context[Exp].procedure[exproc].
        pre[0].transition[0].goto") (flatten-disjunct -) (
        check-report (endgame (grind)) "context[Exp].
        procedure[exproc].pre[0].transition[0].goto")) qed
    vc_loop: lemma
        forall (a, e, res) :
              sit_loop(a, e, res)
              => (lambda (v_loop_0 : int) :
                    (((odd?(e)
                       and e > 0)
                      or (even?(e)
                          and e > 0)
                      or e = 0))
                    and (((odd?(e)
                           and e > 0)
                          => (lambda (res : int, a : int, e
                              : nat) :
                                 (((0 <= e
                                    and e < v_loop_0)))
                                 and (sit_loop(a, e, res)))(a
                                      * res, a, e - 1))
                         and
                         ((even?(e)
                           and e > 0)
                          => (lambda (res : int, a : int, e
                              : nat) :
                                 (((0 <= e
                                    and e < v_loop_0)))
                                 and (sit_loop(a, e, res)))(
                                     res, a * a, e / 2))
                         and
                         (e = 0
                        => sit_fin_(a, e, res))))(e)
```

Figura 43: Representación textual del programa para calcular potencias - Parte 2

```
ctx_Exp: theory
begin
end ctx_Exp



spec_exproc: theory
begin
    importing ctx_Exp;
    a: var int;
    e: var nat;
    a_0: var int;
    e_0: var nat;
    res: var int;

    pre_(a, e): bool =
        (true);

    post_(a_0, e_0, a, e, res): bool =
        (res = expt(a_0, e_0));
end spec_exproc



impl_exproc: theory
begin
    importing spec_exproc;
    a: var int;
    e: var nat;
    res: var int;

    a_0: int;

    e_0: nat;

    sit_ini_(a, e, res): bool =
        ((a = a_0
              and e = e_0))
            and (spec_exproc.pre_(a, e));
```

Figura 44: Teoría PVS del programa para calcular potencias - Parte 1

```
    sit_loop(a, e, res): bool =
        e >= 0
            and res * expt(a, e) = expt(a_0, e_0);

    sit_fin_(a, e, res): bool =
        (spec_exproc.post_(a_0, e_0, a, e, res));
    vc_ini_: lemma
        forall (a, e, res) :
              sit_ini_(a, e, res)
              => (lambda (res : int, a : int, e : nat) :
                sit_loop(a, e, res))(1, a_0, e_0)
    vc_loop: lemma
        forall (a, e, res) :
              sit_loop(a, e, res)
              => (lambda (v_loop_0 : int) :
                    (((odd?(e)
                       and e > 0)
                      or (even?(e)
                          and e > 0)
                      or e = 0))
                    and (((odd?(e)
                           and e > 0)
                          => (lambda (res : int, a : int, e
                              : nat) :
                                 (((0 <= e
                                    and e < v_loop_0)))
                                 and (sit_loop(a, e, res)))(a
                                      * res, a, e - 1))
                         and
                         ((even?(e)
                           and e > 0)
                          => (lambda (res : int, a : int, e
                              : nat) :
                                 (((0 <= e
                                    and e < v_loop_0)))
                                 and (sit_loop(a, e, res)))(
                                     res, a * a, e / 2))
                         and
                         (e = 0
                       => sit_fin_(a, e, res))))(e)
end impl_exproc
```

Figura 45: Teoría PVS del programa para calcular potencias - Parte 2

```
def expFun(a_0,e_0):
  res,a,e=1,a_0,e_0
  while e>0:
    if e % 2 == 0:
      res,a,e=res,a*a,e/2
    else:
      res,a,e=a*res,a,e-1
  return res
```

Figura 46: Implementación en Python del programa para calcular potencias

Factorial

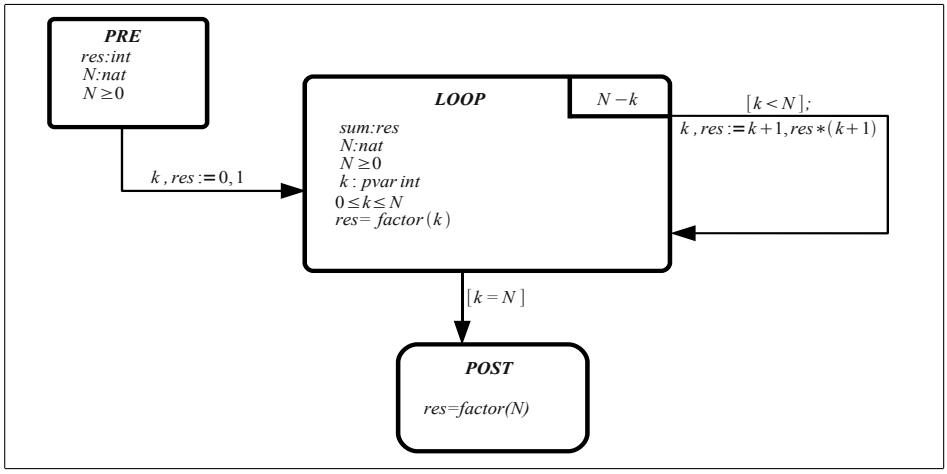

Figura 47: Diagrama de invariantes del programa para calcular el factorial de N

```
factorial: CONTEXT
begin

  using "(endgame (grind))";

  factor(n:nat) : RECURSIVE nat =
    if n = 0 then 1
      else n*factor(n-1)
    endif
  MEASURE n;


  factorialProc [N:nat, result res:int]: PROCEDURE
    PRE N >= 0;
    POST res=factor(N);
    begin

      k:pvar int;

      loop:SITUATION
      begin
        * N>=0;
        * 0 <= k and k <= N;
        * res=factor(k);
        ** N - k;

        if
          [k<N];k,res:=k+1,res*(k+1); decreasing goto loop
          [k=N]; exit
        endif

      end loop

    k,res:=0,1;goto loop

  end  factorialProc

end factorial
```

Figura 48: Representación textual del programa para calcular el factorial de N

```
ctx_factorial: theory
begin

    factor(n : nat): RECURSIVE nat =
        if n = 0 then
                1
            else
                n * factor(n - 1)
            endif
        MEASURE n;
end ctx_factorial



spec_factorialProc: theory
begin
    importing ctx_factorial;
    N: var nat;
    res: var int;

    pre_(N): bool =
        (N >= 0);

    post_(N, res): bool =
        (res = factor(N));
end spec_factorialProc



impl_factorialProc: theory
begin
    importing spec_factorialProc;
    res: var int;

    N: nat;
    k: var int;

    sit_ini_(res, k): bool =
        ((true))
            and (spec_factorialProc.pre_(N));
```

Figura 49: Teoría PVS del programa para calcular el factorial de N - Parte 1.

```
    sit_loop(res, k): bool =
        N >= 0
            and (0 <= k
                 and k <= N)
            and res = factor(k);

    sit_fin_(res, k): bool =
        (spec_factorialProc.post_(N, res));
    vc_ini_: lemma
        forall (res, k) :
              sit_ini_(res, k)
              => (lambda (k : int, res : int) :
                sit_loop(res, k))(0, 1)
    vc_loop: lemma
        forall (res, k) :
              sit_loop(res, k)
              => (lambda (v_loop_0 : int) :
                    ((k < N
                      or k = N))
                    and ((k < N
                          => (lambda (k : int, res : int) :
                                (((0 <= N - k
                                   and N - k < v_loop_0)))
                                and (sit_loop(res, k)))(k +
                                    1, res * (k + 1)))
                        and
                        (k = N
                      => sit_fin_(res, k))))(N - k)
end impl_factorialProc
```

Figura 50: Teoría PVS del programa para calcular el factorial de N - Parte 2.

```
def factFun(N):
  k,res=0,1
  while (k<N):
    k,res=k+1,res*(k+1)
  return res
```

Figura 51: Implementación en Python del programa para calcular el factorial de N

Palíndrome

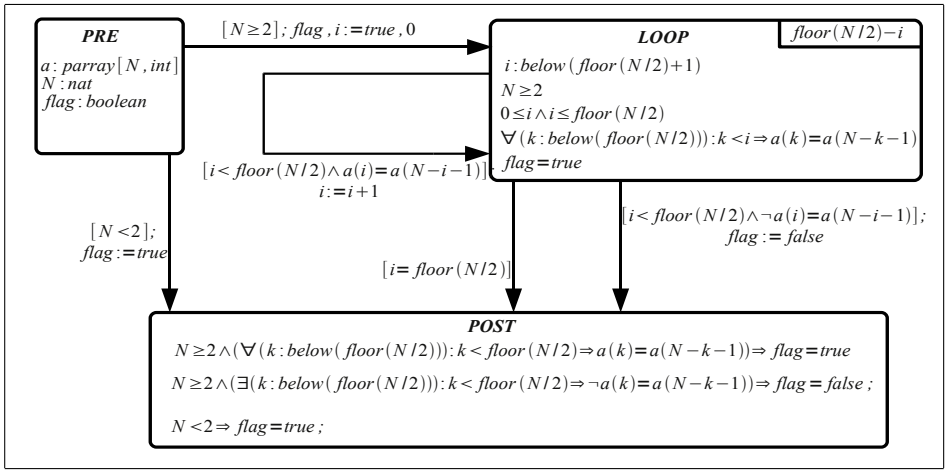

Figura 52: Diagrama de invariantes del programa para decidir si un arreglo es palíndrome

```
module : CONTEXT
BEGIN

  N:nat;

  importing parray;

    palindrome [a:parray[N,int],result flag:boolean] :
        PROCEDURE
      POST N>=2 and (forall (k: below(floor(N / 2))): k<
          floor(N/2) => a(k)=a(N-k-1)) => flag = true;
      POST N>=2 and (exists (k: below(floor(N / 2))): k<
          floor(N/2) => a(k)/=a(N-k-1)) => flag = false;
      POST N<2 => flag = true;

    BEGIN
        i:pvar below(floor(N/2)+1);

        LOOP : SITUATION BEGIN
            * N>=2;
            * flag=true;
            * 0<=i and i<=floor(N/2);
            * forall (k: below(floor(N / 2))): k<i => a(k)=a
                (N-k-1);
            ** floor(N/2)-i;

            IF
                    [i<floor(N/2) and a(i)=a(N-i-1)]; i:=i
                        +1; decreasing GOTO LOOP
                    [i=floor(N/2)]; EXIT
                    [i<floor(N/2) and a(i)/=a(N-i-1)]; flag
                        :=false; EXIT
            ENDIF
        END LOOP

        if
          [N>=2]; flag,i:=true,0; goto LOOP
          [N<2]; flag:=true; exit
        endif

    END palindrome

END module
```

Figura 53: Representación textual del programa para decidir si un arreglo es palíndrome

```
ctx_module: theory
begin

    N: nat;
    importing parray;
end ctx_module

spec_palindrome: theory
begin
    importing ctx_module;
    a: var parray[N, int];
    flag: var boolean;

    pre_(a): bool =
        (true);

    post_(a, flag): bool =
        (N >= 2
             and (forall (k : below(floor(N / 2))) :
                    k < floor(N / 2)
                    => a(k) = a(N - k - 1))
             => flag = true)
            and (N >= 2
                 and (exists (k : below(floor(N / 2))) :
                        k < floor(N / 2)
                        => a(k) /= a(N - k - 1))
                 => flag = false)
            and (N < 2
                 => flag = true);
end spec_palindrome

impl_palindrome: theory
begin
    importing spec_palindrome;
    flag: var boolean;

    a: parray[N, int];
    i: var below(floor(N / 2) + 1);

    sit_ini_(flag, i): bool =
        ((true))
            and (spec_palindrome.pre_(a));
```

Figura 54: Teoría PVS del programa para decidir si un arreglo es palíndrome - Parte 1

```
sit_LOOP(flag, i): bool =
    N >= 2
        and flag = true
        and (0 <= i
             and i <= floor(N / 2))
        and (forall (k : below(floor(N / 2))) :
               k < i
               => a(k) = a(N - k - 1));

sit_fin_(flag, i): bool =
    (spec_palindrome.post_(a, flag));
vc_ini_: lemma
    forall (flag, i) :
          sit_ini_(flag, i)
          => ((N >= 2
               or N < 2))
          and ((N >= 2
                => (lambda (flag : boolean, i : below(
                    floor(N / 2) + 1)) :
                       sit_LOOP(flag, i))(true, 0))
               and (N < 2
                    => (lambda (flag : boolean) :
                       sit_fin_(flag, i))(true)))
```

Figura 55: Teoría PVS del programa para decidir si un arreglo es palíndrome - Parte 2

```
    sit_LOOP(flag, i): bool =
        N >= 2
            and flag = true
            and (0 <= i
                 and i <= floor(N / 2))
            and (forall (k : below(floor(N / 2))) :
                   k < i
                   => a(k) = a(N - k - 1));

    sit_fin_(flag, i): bool =
        (spec_palindrome.post_(a, flag));
    vc_ini_: lemma
        forall (flag, i) :
              sit_ini_(flag, i)
              => ((N >= 2
                   or N < 2))
              and ((N >= 2
                    => (lambda (flag : boolean, i : below(
                        floor(N / 2) + 1)) :
                           sit_LOOP(flag, i))(true, 0))
                   and (N < 2
                        => (lambda (flag : boolean) :
                           sit_fin_(flag, i))(true)))
    vc_LOOP: lemma
        forall (flag, i) :
              sit_LOOP(flag, i)
              => (lambda (v_LOOP_0 : int) :
                    (((i < floor(N / 2)
                       and a(i) = a(N - i - 1))
                      or i = floor(N / 2)
                      or (i < floor(N / 2)
                          and a(i) /= a(N - i - 1))))
                    and (((i < floor(N / 2)
                           and a(i) = a(N - i - 1))
                          => (lambda (i : below(floor(N / 2)
                              + 1)) :
                                (((0 <= floor(N / 2) - i
                                   and floor(N / 2) - i <
                                       v_LOOP_0)))
                                and (sit_LOOP(flag, i)))(i +
                                    1))
                         and (i = floor(N / 2)
                              => sit_fin_(flag, i))
                         and ((i < floor(N / 2)
                               and a(i) /= a(N - i - 1))
                              => (lambda (flag : boolean) :
                                     sit_fin_(flag, i))(false
                                         ))))(floor(N / 2)- i)
end impl_palindrome
```

Figura 56: Teoría PVS del programa para decidir si un arreglo es palíndrome - Parte 3

```
def palindromeFun(a,N):
  if N<2:
    return True
  else:
    flag,i=True,0
    while i<math.floor(N/2):
      if a[i]==a[N-i-1]:
        i=i+1
      if not a[i]==a[N-i-1]:
        flag=False
        return flag
    return flag
```

Figura 57: Implementación en Python del programa para decidir si un arreglo es palíndrome

Partición

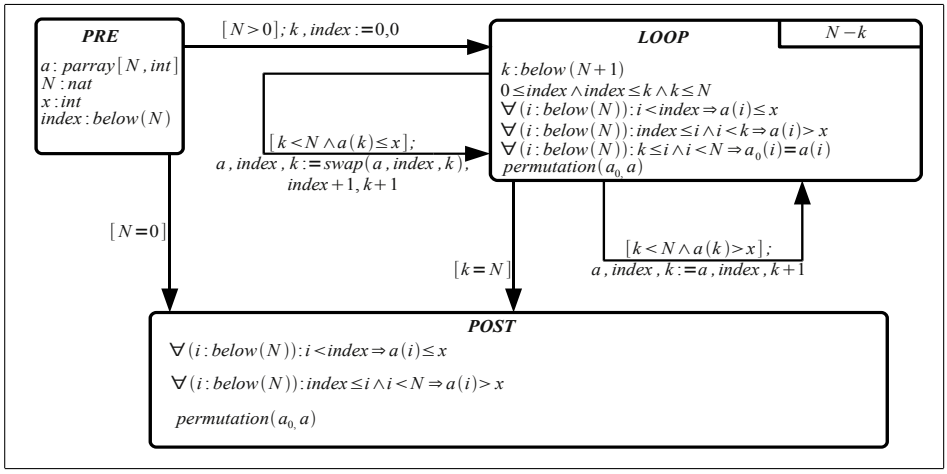

Figura 58: Diagrama de invariantes del programa para particionar un arreglo

```
partition: CONTEXT
BEGIN

  N:nat;

  IMPORTING parray;

  partition[x:int,valres a:parray[N,int],result index:below(
      N+1)]: PROCEDURE
    POST forall (i:below(N)): i<index => a(i)<=x;
    POST forall (i:below(N)): index<=i and i<N => a(i)>x;
    POST permutation(a_0,a);

  BEGIN
    k: pvar below(N+1);

    loop: SITUATION
    begin
      * 0<=index and index<=k and k<=N;
      * forall (i:below(N)): i<index => a(i)<=x;
      * forall (i:below(N)): index<=i and i<k => a(i)>x;
      * forall (i:below(N)): k<=i and i<N => a_0(i)=a(i);
      * permutation(a_0,a);
      ** N-k;

      IF
        [k=N]; exit
        [k<N and a(k)<=x]; a,index,k := swap(a,index,k),
            index+1,k+1; decreasing goto loop
        [k<N and a(k)>x]; a,index,k := a,index,k+1;
            decreasing goto loop
      ENDIF

    END loop

    IF
        [N=0]; exit
        [N>0]; k,index:=0,0; goto loop
    ENDIF

  END partition

END partition
```

Figura 59: Representación textual del programa para particionar un arreglo

```
ctx_partition: theory
begin

    N: nat;
    importing parray;
end ctx_partition

spec_partition: theory
begin
    importing ctx_partition;
    x: var int;
    a: var parray[N, int];
    a_0: var parray[N, int];
    index: var below(N + 1);

    pre_(x, a): bool =
        (true);

    post_(x, a_0, a, index): bool =
        (forall (i : below(N)) :
                i < index
                => a(i) <= x)
            and (forall (i : below(N)) :
                    index <= i
                    and i < N
                    => a(i) > x)
            and (permutation(a_0, a));
end spec_partition

impl_partition: theory
begin
    importing spec_partition;
    a: var parray[N, int];
    index: var below(N + 1);

    x: int;

    a_0: parray[N, int];
    k: var below(N + 1);

    sit_ini_(a, index, k): bool =
        (((a = a_0)))
            and (spec_partition.pre_(x, a));
```

Figura 60: Teoría PVS del programa para particionar un arreglo - Parte 1

```
sit_loop(a, index, k): bool =
    (0 <= index
         and index <= k
         and k <= N)
        and (forall (i : below(N)) :
               i < index
               => a(i) <= x)
        and (forall (i : below(N)) :
               index <= i
               and i < k
               => a(i) > x)
        and (forall (i : below(N)) :
               k <= i
               and i < N
               => a_0(i) = a(i))
        and (permutation(a_0, a));

sit_fin_(a, index, k): bool =
    (spec_partition.post_(x, a_0, a, index));
vc_ini_: lemma
    forall (a, index, k) :
          sit_ini_(a, index, k)
          => ((N = 0
               or N > 0))
          and ((N = 0
                => sit_fin_(a, index, k))
               and (N > 0
                    => (lambda (k : below(N + 1), index
                        : below(N + 1)) :
                       sit_loop(a, index, k))(0, 0)))
```

Figura 61: Teoría PVS del programa para particionar un arreglo - Parte 2

```
    vc_loop: lemma
        forall (a, index, k) :
               sit_loop(a, index, k)
               => (lambda (v_loop_0 : int) :
                     ((k = N
                       or (k < N
                           and a(k) <= x)
                       or (k < N
                           and a(k) > x)))
                     and ((k = N
                           => sit_fin_(a, index, k))
                          and ((k < N
                                and a(k) <= x)
                               => (lambda (a : parray[N, int
                                   ], index : below(N+ 1),
                                            k : below(N + 1))
                                               :
                                      (((0 <= N - k
                                         and N - k < v_loop_0)
                                            ))
                                      and (sit_loop(a, index,
                                          k)))(swap(a, index,k)
                                          , index + 1, k + 1))
                          and
                          ((k < N
                            and a(k) > x)
                           => (lambda (a : parray[N, int],
                               index : below(N + 1),
                                       k : below(N + 1)) :
                                  (((0 <= N - k
                                     and N - k < v_loop_0)))
                                  and (sit_loop(a, index, k)))
                                      (a, index, k + 1))))(N -
                                      k)
end impl_partition
```

Figura 62: Teoría PVS del programa para particionar un arreglo - Parte 3

```
def partitionFun(a,x):
  if len(a)==0:
    return a;
  else:
    k,index=0,0
    while k<len(a):
      if a[k]<=x:
        aux=a[index];
        a[index]=a[k]
        a[k]=aux;
        index,k=index+1,k+1
      else:
        k=k+1
    return a
```

Figura 63: Implementación en Python del programa para particionar un arreglo

Búsqueda lineal

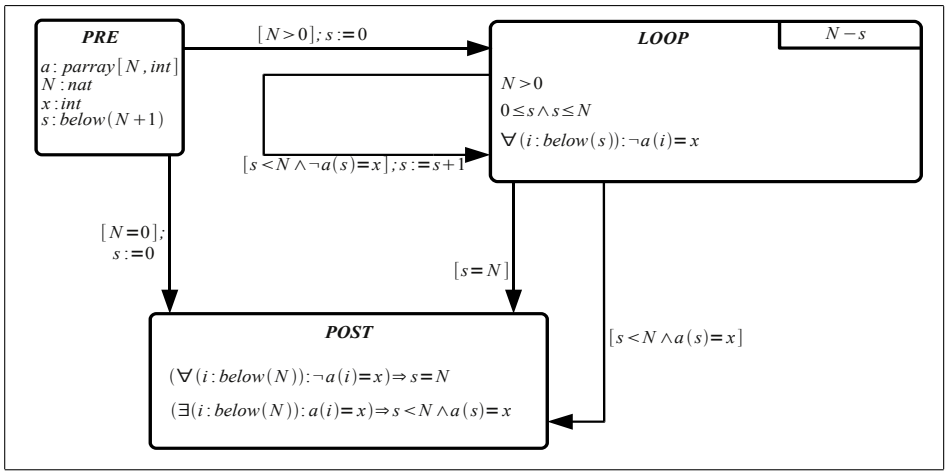

Figura 64: Diagrama de invariantes de la búsqueda lineal

```
LinerSearch:CONTEXT
begin

  using "(endgame (grind))";

  importing parray;

  N:nat;

  SearchElement [a:parray[N,int], x:int, result s:below(N+1)
      ]: PROCEDURE
    POST (forall (i:below(N)): a(i)/=x) => s=N;
    POST (exists (i:below(N)): a(i)=x) => s<N and a(s)=x;
    BEGIN

      Search: SITUATION
      BEGIN
        * N>0;
        * 0 <= s and s <= N;
        * forall (i:below(s)): a(i)/=x;
        ** N-s;

        if [s = N]; exit
           [s < N and a(s)=x]; exit
           [s < N and a(s)/=x]; s:=s+1; decreasing goto
               Search

        endif
      END Search

    if
      [N=0]; s:=0; exit
      [N>0]; s:=0; goto Search
    endif

    END SearchElement

END LinerSearch
```

Figura 65: Representación textual del programa para realizar una búsqueda lineal

```
ctx_LinerSearch: theory
begin
    importing parray;

    N: nat;
end ctx_LinerSearch

spec_SearchElement: theory
begin
    importing ctx_LinerSearch;
    a: var parray[N, int];
    x: var int;
    s: var below(N + 1);

    pre_(a, x): bool =
        (true);

    post_(a, x, s): bool =
        ((forall (i : below(N)) :
                a(i) /= x)
             => s = N)
            and ((exists (i : below(N)) :
                    a(i) = x)
                 => s < N
                  and a(s) = x);
end spec_SearchElement

impl_SearchElement: theory
begin
    importing spec_SearchElement;
    s: var below(N + 1);

    a: parray[N, int];

    x: int;

    sit_ini_(s): bool =
        ((true))
            and (spec_SearchElement.pre_(a, x));
```

Figura 66: Teoría PVS del programa para realizar una búsqueda lineal - Parte 1

```
    sit_Search(s): bool =
        N > 0
            and (0 <= s
                 and s <= N)
            and (forall (i : below(s)) :
                   a(i) /= x);

    sit_fin_(s): bool =
        (spec_SearchElement.post_(a, x, s));
    vc_ini_: lemma
        forall (s) :
              sit_ini_(s)
              => ((N = 0
                   or N > 0))
              and ((N = 0
                    => (lambda (s : below(N + 1)) :
                          sit_fin_(s))(0))
                   and (N > 0
                        => (lambda (s : below(N + 1)) :
                           sit_Search(s))(0)))
    vc_Search: lemma
        forall (s) :
              sit_Search(s)
              => (lambda (v_Search_0 : int) :
                    ((s = N
                      or (s < N
                          and a(s) = x)
                      or (s < N
                          and a(s) /= x)))
                    and ((s = N
                          => sit_fin_(s))
                         and ((s < N
                               and a(s) = x)
                              => sit_fin_(s))
                         and ((s < N
                               and a(s) /= x)
                              => (lambda (s : below(N + 1))
                                  :
                                     (((0 <= N - s
                                        and N - s <
                                            v_Search_0)))
                                and (sit_Search(s)))(s + 1))
                                   ))(N - s)
end impl_SearchElement
```

Figura 67: Teoría PVS del programa para realizar una búsqueda lineal - Parte 2

```
def linearSearchFun(a,x):
  s=0
  if len(a)==0:
    return s;
  else:
    while s<len(a):
      if a[s]!=x:
        s=s+1
      else:
        return s
    return s
```

Figura 68: Implementación en Python del programa para realizar una búsqueda lineal

Búsqueda lineal en dos dimensiones

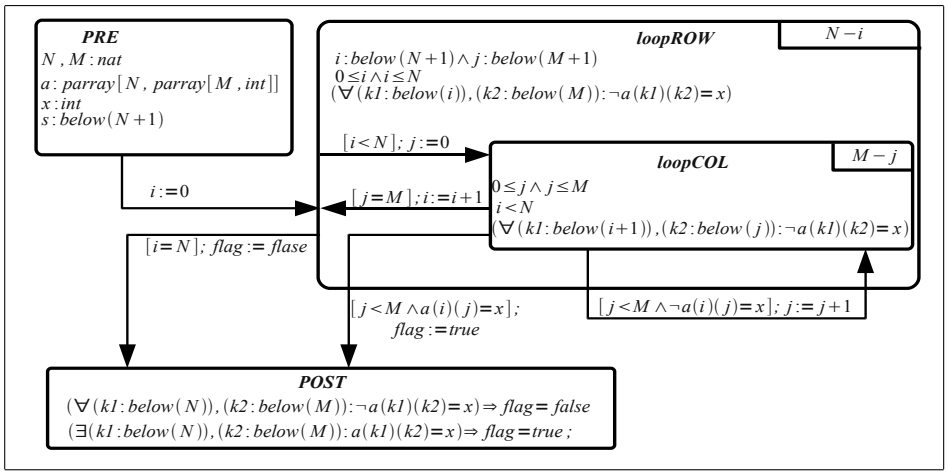

Figura 69: Diagrama de invariantes de la búsqueda lineal en dos dimensiones

```
LS2Dim: CONTEXT
BEGIN
  N:nat;
  M:nat;
  importing parray;

  LS2Dim[a:parray[N,parray[M,int]],x:int,result flag:boolean
      ]: PROCEDURE
    POST (forall (k1:below(N)),(k2:below(M)): a(k1)(k2)/=x)
        => flag=false;
    POST (exists (k1:below(N)),(k2:below(M)): a(k1)(k2)=x)
        => flag=true;
  BEGIN
    i:pvar below(N+1);
    j:pvar below(M+1);

    loopROW: SITUATION
    BEGIN
      * 0<=i and i<=N;
      * (forall (k1:below(i)),(k2:below(M)): a(k1)(k2)/=x);
      ** N-i;

      loopCOL: SITUATION
      BEGIN
        * 0<=j and j<=M;
        * i<N;
        * (forall (k1:below(i+1)),(k2:below(j)): a(k1)(k2)/=
            x);
        ** M-j;

        IF
          [j=M]; i:=i+1; decreasing GOTO loopROW
          [j<M and a(i)(j)/=x]; j:=j+1; decreasing GOTO
              loopCOL
          [j<M and a(i)(j)=x]; flag:=true; exit
        ENDIF
      END loopCOL

      IF
        [i=N]; flag:=false; exit
        [i<N]; j:=0; GOTO loopCOL
      ENDIF
    END loopROW
    i:= 0; GOTO loopROW
  END LS2Dim
END LS2Dim
```

Figura 70: Representación textual del programa para realizar una búsqueda lineal en dos dimensiones

```
ctx_LS2Dim: theory
begin

    N: nat;

    M: nat;
    importing parray;
end ctx_LS2Dim

spec_LS2Dim: theory
begin
    importing ctx_LS2Dim;
    a: var parray[N, parray[M, int]];
    x: var int;
    flag: var boolean;

    pre_(a, x): bool =
        (true);

    post_(a, x, flag): bool =
        ((forall (k1 : below(N)), (k2 : below(M)) :
                a(k1)(k2) /= x)
             => flag = false)
            and ((exists (k1 : below(N)), (k2 : below(M)) :
                    a(k1)(k2) = x)
                 => flag = true);
end spec_LS2Dim



impl_LS2Dim: theory
begin
    importing spec_LS2Dim;
    flag: var boolean;

    a: parray[N, parray[M, int]];

    x: int;
    i: var below(N + 1);
    j: var below(M + 1);

    sit_ini_(flag, i, j): bool =
        ((true))
            and (spec_LS2Dim.pre_(a, x));
```

Figura 71: Teoría PVS del programa para realizar una búsqueda lineal en dos dimensiones - Parte 1

```
    sit_loopROW(flag, i, j): bool =
        (0 <= i
             and i <= N)
            and ((forall (k1 : below(i)), (k2 : below(M)) :
                    a(k1)(k2) /= x));

    sit_loopCOL(flag, i, j): bool =
        (sit_loopROW(flag, i, j))
            and (0 <= j
                 and j <= M)
            and i < N
            and ((forall (k1 : below(i + 1)), (k2 : below(j)
                ) :
                    a(k1)(k2) /= x));

    sit_fin_(flag, i, j): bool =
        (spec_LS2Dim.post_(a, x, flag));
    vc_ini_: lemma
        forall (flag, i, j) :
              sit_ini_(flag, i, j)
              => (lambda (i : below(N + 1)) :
                sit_loopROW(flag, i, j))(0)
    vc_loopROW: lemma
        forall (flag, i, j) :
              sit_loopROW(flag, i, j)
              => (lambda (v_loopROW_0 : int) :
                    ((i = N
                      or i < N))
                    and ((i = N
                          => (lambda (flag : boolean) :
                                sit_fin_(flag, i, j))(false)
                              )
                         and (i < N
                              => (lambda (j : below(M + 1))
                                  :
                                     (((0 <= N - i
                                        and N - i <=
                                            v_loopROW_0)))
                                     and (sit_loopCOL(flag, i
                                         , j)))(0))))(N - i)
```

Figura 72: Teoría PVS del programa para realizar una búsqueda lineal en dos dimensiones - Parte 2

```
    vc_loopCOL: lemma
        forall (flag, i, j) :
              sit_loopCOL(flag, i, j)
              => (lambda (v_loopROW_0 : int, v_loopCOL_0 :
                  int) :
                     ((j = M
                       or (j < M
                           and a(i)(j) /= x)
                       or (j < M
                           and a(i)(j) = x)))
                     and ((j = M
                           => (lambda (i : below(N + 1)) :
                                  (((0 <= N - i
                                     and N - i < v_loopROW_0))
                                      )
                                  and (sit_loopROW(flag, i, j)
                                      ))(i + 1))
                          and ((j < M
                                and a(i)(j) /= x)
                               => (lambda (j : below(M + 1))
                                   :
                                      (((0 <= M - j
                                         and M - j <
                                             v_loopCOL_0)
                                        and (0 <= N - i
                                             and N - i <=
                                                 v_loopROW_0)))
                                      and (sit_loopCOL(flag, i
                                          , j)))(j + 1))
                          and
                          ((j < M
                            and a(i)(j) = x)
                           => (lambda (flag : boolean) :
                              sit_fin_(flag, i, j))(true))))(N
                                   - i, M - j)
end impl_LS2Dim
```

Figura 73: Teoría PVS del programa para realizar una búsqueda lineal en dos dimensiones - Parte 3

```
def LS2DimFun(a,M,x):
  i=0
  while i<len(a):
    j=0
    while(j<M):
      if a[i][j]!=x:
        j=j+1
      else:
        flag=True
        return flag
    i=i+1
  flag=False
  return flag
```

Figura 74: Implementación en Python del programa para realizar una búsqueda lineal en dos dimensiones

Búsqueda binaria

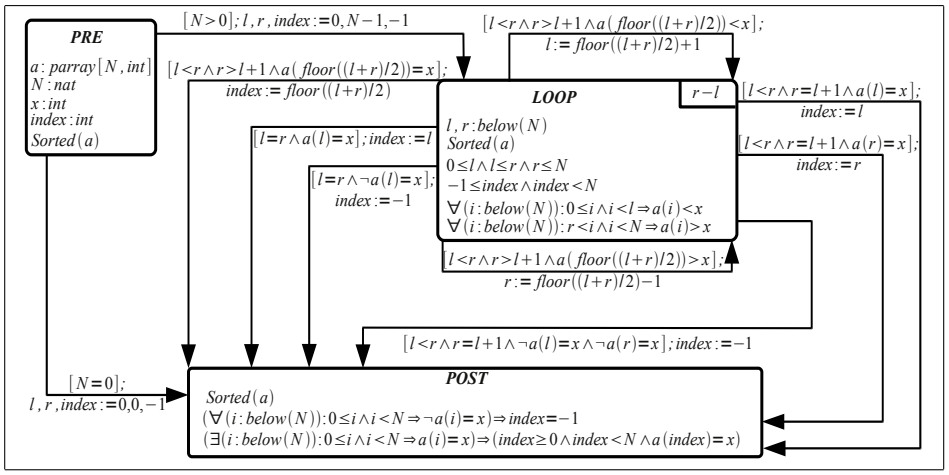

Figura 75: Diagrama de invariantes del programa para realizar una búsqueda binaria

```
binarySearch: CONTEXT
BEGIN
  N:nat;
  importing parray;
  Sorted(a:parray[N,int]) : bool = (forall (i:below(N)),(j:
      below(N)): (i<=j => a(i)<=a(j)));
  binarySearch[valres a:parray[N,int],x:int,result index:int
      ]: PROCEDURE
    PRE Sorted(a);
    POST Sorted(a);
    POST (forall (i:below(N)): 0<=i and i<N => a(i)/=x) =>
        index = -1;
    POST (exists (i:below(N)): 0<=i and i<N => a(i)=x) =>(
        index >= 0 and index<N and a(index)=x);
  BEGIN
    l: pvar below(N);
    r: pvar below(N);

    LOOP: SITUATION
    BEGIN
      * Sorted(a);
      * 0<=l and l<=r and r<=N;
      * -1<=index and index<N;
      * forall (i:below(N)): 0<=i and i<l => a(i)<x;
      * forall (i:below(N)): r<i and i<N => a(i)>x;
      ** r-l;
     IF
        [l=r and a(l)/=x]; index:=-1; exit
        [l=r and a(l)=x]; index:=l; exit
        [l<r and r>l+1 and a(floor((l+r)/2))=x]; index:=
            floor((l+r)/2); exit
        [l<r and r>l+1 and a(floor((l+r)/2))<x]; l:=floor((l
            +r)/2)+1; decreasing GOTO LOOP
        [l<r and r>l+1 and a(floor((l+r)/2))>x]; r:=floor((l
            +r)/2)-1; decreasing GOTO LOOP
        [l<r and r=l+1 and a(l)=x]; index:=l; exit
        [l<r and r=l+1 and a(r)=x]; index:=r; exit
        [l<r and r=l+1 and a(l)/=x and a(r)/=x ]; index:=-1;
             exit
      ENDIF
    END LOOP
    IF
      [N>0]; l,r,index:=0, N-1,-1; GOTO LOOP
      [N=0]; l,r,index:=0,0,-1; exit
    ENDIF
  END binarySearch
END binarySearch
```

Figura 76: Representación textual del programa para realizar una búsqueda binaria

```
ctx_binarySearch: theory
begin

    N: nat;
    importing parray;

    Sorted(a : parray[N, int]): bool =
        (forall (i : below(N)), (j : below(N)) :
                (i <= j
                 => a(i) <= a(j)));
end ctx_binarySearch

spec_binarySearch: theory
begin
    importing ctx_binarySearch;
    a: var parray[N, int];
    x: var int;
    a_0: var parray[N, int];
    x_0: var int;
    index: var int;

    pre_(a, x): bool =
        (Sorted(a));

    post_(a_0, x_0, a, x, index): bool =
        (Sorted(a))
            and ((forall (i : below(N)) :
                     0 <= i
                     and i < N
                     => a(i) /= x)
                 => index = -1)
            and ((exists (i : below(N)) :
                     0 <= i
                     and i < N
                     => a(i) = x)
                 => (index >= 0
                      and index < N
                      and a(index) = x));
end spec_binarySearch
```

Figura 77: Teoría PVS del programa para realizar una búsqueda binaria - Parte 1

```
impl_binarySearch: theory
begin
    importing spec_binarySearch;
    a: var parray[N, int];
    x: var int;
    index: var int;

    a_0: parray[N, int];

    x_0: int;
    l: var below(N);
    r: var below(N);

    sit_ini_(a, x, index, l, r): bool =
        ((a = a_0
              and x = x_0))
            and (spec_binarySearch.pre_(a, x));

    sit_LOOP(a, x, index, l, r): bool =
        (Sorted(a))
            and (0 <= l
                 and l <= r
                 and r <= N)
            and (-1 <= index
                 and index < N)
            and (forall (i : below(N)) :
                   0 <= i
                   and i < l
                   => a(i) < x)
            and (forall (i : below(N)) :
                   r < i
                   and i < N
                   => a(i) > x);

    sit_fin_(a, x, index, l, r): bool =
        (spec_binarySearch.post_(a_0, x_0, a, x, index));
```

Figura 78: Teoría PVS del programa para realizar una búsqueda binaria - Parte 2

```
    vc_ini_: lemma
        forall (a, x, index, l, r) :
               sit_ini_(a, x, index, l, r)
               => ((N > 0
                    or N = 0))
               and ((N > 0
                     => (lambda (l : below(N), r : below(N),
                         index : int) :
                            sit_LOOP(a, x, index, l, r))(0, N
                                - 1, -1))
                    and (N = 0
                         => (lambda (l : below(N), r : below(
                             N), index : int) :
                            sit_fin_(a, x, index, l, r))(0, 0,
                                -1)))
    vc_LOOP: lemma
        forall (a, x, index, l, r) :
               sit_LOOP(a, x, index, l, r)
               => (lambda (v_LOOP_0 : int) :
                     (((l = r
                        and a(l) /= x)
                       or (l = r
                           and a(l) = x)
                       or (l < r
                           and r > l + 1
                           and a(floor((l + r) / 2)) = x)
                       or (l < r
                           and r > l + 1
                           and a(floor((l + r) / 2)) < x)
                       or (l < r
                           and r > l + 1
                           and a(floor((l + r) / 2)) > x)
                       or (l < r
                           and r = l + 1
                           and a(l) = x)
                       or (l < r
                           and r = l + 1
                           and a(r) = x)
                       or (l < r
                           and r = l + 1
                           and a(l) /= x
                           and a(r) /= x)))
```

Figura 79: Teoría PVS del programa para realizar una búsqueda binaria - Parte 3

```
                    and (((l = r
                           and a(l) /= x)
                          => (lambda (index : int) :
                                 sit_fin_(a, x, index, l, r))
                                (-1))
                         and ((l = r
                               and a(l) = x)
                              => (lambda (index : int) :
                                     sit_fin_(a, x, index, l,
                                          r))(l))
                         and ((l < r
                               and r > l + 1
                               and a(floor((l + r) / 2)) = x
                                   )
                              => (lambda (index : int) :
                                     sit_fin_(a, x, index, l,
                                          r))(floor((l + r) /
                                         2)))
                         and
                         ((l < r
                           and r > l + 1
                           and a(floor((l + r) / 2)) < x)
                          => (lambda (l : below(N)) :
                                 (((0 <= r - l
                                    and r - l < v_LOOP_0)))
                                 and (sit_LOOP(a, x, index, l
                                     , r)))(floor((l + r) / 2)
                                      + 1))
                         and ((l < r
                               and r > l + 1
                               and a(floor((l + r) / 2)) > x
                                   )
                              => (lambda (r : below(N)) :
                                     (((0 <= r - l
                                        and r - l < v_LOOP_0)
                                          ))
                                     and (sit_LOOP(a, x,
                                         index, l, r)))(floor
                                         ((l+ r) / 2) - 1))
```

Figura 80: Teoría PVS del programa para realizar una búsqueda binaria - Parte 4

```
                            and
                            ((l < r
                              and r = l + 1
                              and a(l) = x)
                             => (lambda (index : int) :
                                   sit_fin_(a, x, index, l, r))
                                       (l))
                            and ((l < r
                                  and r = l + 1
                                  and a(r) = x)
                                 => (lambda (index : int) :
                                       sit_fin_(a, x, index, l,
                                            r))(r))
                            and ((l < r
                                  and r = l + 1
                                  and a(l) /= x
                                  and a(r) /= x)
                                 => (lambda (index : int) :
                                   sit_fin_(a, x, index, l, r))
                                       (-1))))(r - l)
end impl_binarySearch
```

Figura 81: Teoría PVS del programa para realizar una búsqueda binaria - Parte 5

```
def binarySearchFun(a,x):
  if len(a)==0:
    index=-1;
    return index;
  else:
    l,r,index=0,len(a)-1,-1
    while l<r:
      if r>l+1:
        if a[int(math.floor((l+r)/2))]==x:
          index=int(math.floor((l+r)/2))
          return index
        else:
          if a[int(math.floor((l+r)/2))]<x:
            l=int(math.floor((l+r)/2))+1
          else:
            r=int(math.floor((l+r)/2))-1
      else:
        if a[l]==x:
          index=l
          return index
        if a[r]==x:
          index=r
          return index
        if a[l]!=x and a[r]!=x:
          index=-1
          return index
             if a[l]==x:
      index=l
      return index
    else:
      index=-1
      return index
```

Figura 82: Implementación en Python del programa para realizar una búsqueda binaria

Insertion sort

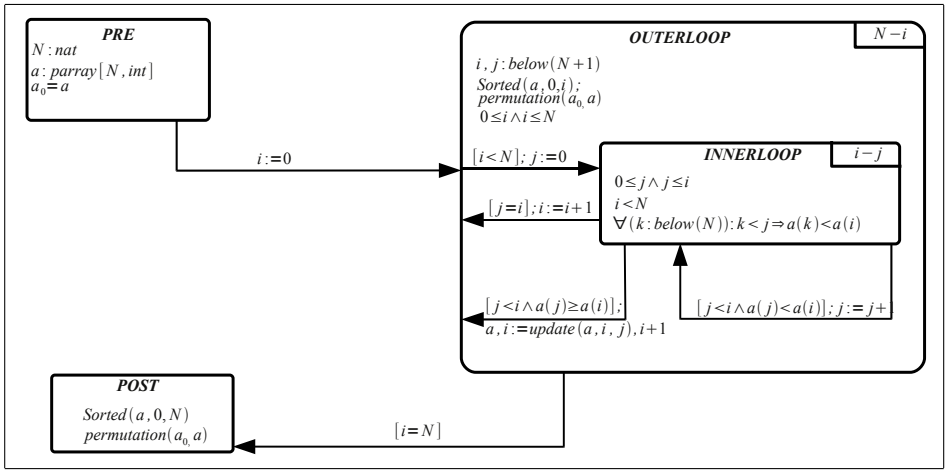

Figura 83: Diagrama de invariantes del programa Insertion sort

```
insertionSort: CONTEXT
BEGIN
  N:nat;
  importing parray;
  Sorted(a:parray[N,int],i:below(N+1),j:{x:below(N+1)|x>=i})
       : bool = (forall (k1:below(j)),(k2:below(j)): (k1<=k2
      => a(k1)<=a(k2)));
  update(a:parray[N,int],i:below(N),j:{x:below(N)| x<=i}):
      parray[N,int] =(lambda (k:below(N)) : if k<j or k>i
      then a(k) else if k=j then a(i) else a(k-1) endif endif
      );

  insertionSort[valres a:parray[N,int]]: PROCEDURE
    POST Sorted(a,0,N);
    POST permutation(a_0,a);
  BEGIN
    i: pvar below(N+1);
    j: pvar below(N+1);
    outerloop: SITUATION
    BEGIN
      * Sorted(a,0,i);
      * permutation(a_0,a);
      * 0<=i and i<=N;
      ** N-i;
        innerloop: SITUATION
        BEGIN
          * Sorted(a,0,i);
          * permutation(a_0,a);
          * 0<=i and i<=N;
          * 0<=j and j<=i;
          * i<N;
          * forall (k:below(N)): k<j => a(k)<a(i);
          ** i-j;
        IF
          [j=i];i:=i+1; decreasing GOTO outerloop
          [j<i and a(j)>=a(i)]; a,i:=update(a,i,j),i+1;
              decreasing GOTO outerloop
          [j<i and a(j)<a(i)]; j:=j+1; decreasing GOTO
              innerloop
        ENDIF
        END innerloop
      IF
        [i=N]; exit
        [i<N]; j:=0; GOTO innerloop
      ENDIF
    END outerloop
    i:=0; GOTO outerloop
  END insertionSort
END insertionSort
```

Figura 84: Representación textual del programa Insertion sort

```
ctx_insertionSort: theory
begin

    N: nat;
    importing parray;

    Sorted(a : parray[N, int], i : below(N + 1), j : { x :
        below(N + 1) | x >= i }): bool = (forall (k1 : below(
        j)), (k2 : below(j)) : (k1 <= k2 => a(k1) <= a(k2)));

    update(a : parray[N, int], i : below(N), j : { x : below
        (N) | x <= i }): parray[N, int] = (lambda (k : below(
        N)) : if k < j or k > i then a(k) else if k = j then
        a(i) else a(k - 1) endif endif);
end ctx_insertionSort

spec_insertionSort: theory
begin
    importing ctx_insertionSort;
    a: var parray[N, int];
    a_0: var parray[N, int];

    pre_(a): bool =
        (true);

    post_(a_0, a): bool =
        (Sorted(a, 0, N))
            and (permutation(a_0, a));
end spec_insertionSort

impl_insertionSort: theory
begin
    importing spec_insertionSort;
    a: var parray[N, int];

    a_0: parray[N, int];
    i: var below(N + 1);
    j: var below(N + 1);

    sit_ini_(a, i, j): bool =
        (((a = a_0)))
            and (spec_insertionSort.pre_(a));
```

Figura 85: Teoría PVS del programa Insertion sort - Parte 1

```
    sit_outerloop(a, i, j): bool =
        (Sorted(a, 0, i))
            and (permutation(a_0, a))
            and (0 <= i
                 and i <= N);

    sit_innerloop(a, i, j): bool =
        (sit_outerloop(a, i, j))
            and (Sorted(a, 0, i))
            and (permutation(a_0, a))
            and (0 <= i
                 and i <= N)
            and (0 <= j
                 and j <= i)
            and i < N
            and (forall (k : below(N)) :
                   k < j
                   => a(k) < a(i));

    sit_fin_(a, i, j): bool =
        (spec_insertionSort.post_(a_0, a));
    vc_ini_: lemma
        forall (a, i, j) :
              sit_ini_(a, i, j)
              => (lambda (i : below(N + 1)) :
                sit_outerloop(a, i, j))(0)
    vc_outerloop: lemma
        forall (a, i, j) :
              sit_outerloop(a, i, j)
              => (lambda (v_outerloop_0 : int) :
                    ((i = N
                      or i < N))
                    and ((i = N
                          => sit_fin_(a, i, j))
                         and (i < N
                              => (lambda (j : below(N + 1))
                                  :
                                     (((0 <= N - i
                                        and N - i <=
                                            v_outerloop_0)))
                                   and (sit_innerloop(a, i, j))
                                  )(0))))(N - i)
```

Figura 86: Teoría PVS del programa Insertion sort - Parte 2

```
    vc_innerloop: lemma
        forall (a, i, j) :
              sit_innerloop(a, i, j)
              => (lambda (v_outerloop_0 : int, v_innerloop_0
                   : int) :
                    ((j = i
                      or (j < i
                          and a(j) >= a(i))
                      or (j < i
                          and a(j) < a(i))))
                    and ((j = i
                          => (lambda (i : below(N + 1)) :
                                 (((0 <= N - i
                                    and N - i < v_outerloop_0
                                        )))
                                 and (sit_outerloop(a, i, j))
                                     )(i + 1))
                               and ((j < i
                                     and a(j) >= a(i))
                                    => (lambda (a : parray[N, int
                                        ], i : below(N + 1)) :
                                           (((0 <= N - i
                                              and N - i <
                                                  v_outerloop_0)))
                                           and (sit_outerloop(a, i,
                                                j)))(update(a,i,j),i
                                                + 1))
                               and ((j < i
                                     and a(j) < a(i))
                                    => (lambda (j : below(N + 1))
                                        :
                                           (((0 <= i - j
                                              and i - j <
                                                  v_innerloop_0)
                                             and (0 <= N - i
                                                  and N - i <=
                                                      v_outerloop_0)
                                                  ))
                                           and (sit_innerloop(a, i,
                                                j)))(j + 1))))(N - i
                                             , i - j)
end impl_insertionSort
```

Figura 87: Teoría PVS del programa Insertion sort - Parte 3

```
def update(a,i,j):
  b=a[:]
  k=0
  while k<len(a):
    if k<j or k>i:
      b[k]=a[k]
    else:
      if k==j:
        b[k]=a[i]
      else:
        b[k]=a[k-1]
    k=k+1
  return b

def insertionSortFun(a):
  i=0
  while i<len(a):
    j=0
    while(j<i):
      if a[j]>=a[i]:
        a=update(a,i,j)
        break
      else:
        j=j+1
    i=i+1
  return a
```

Figura 88: Implementación en Python del programa Insertion sort

Problema de la bandera alemana

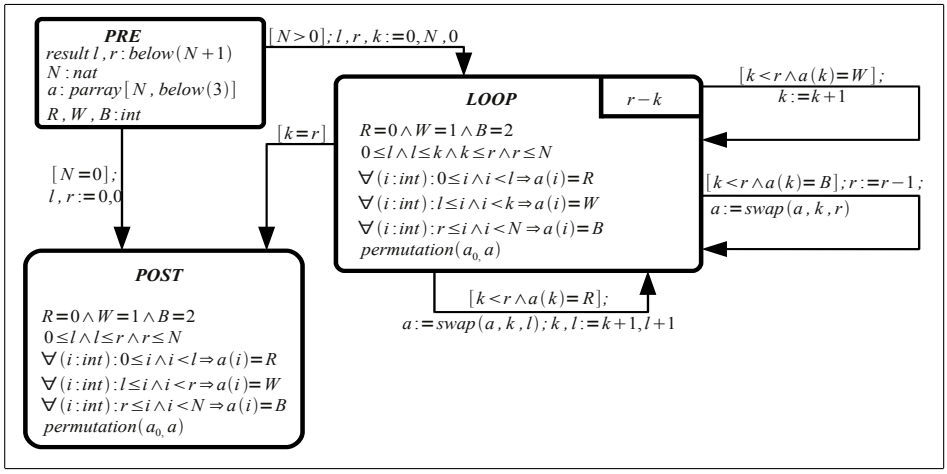

Figura 89: Diagrama de invariantes del programa para resolver el problema de la bandera alemana

```
flag: CONTEXT
BEGIN
  N:nat;
  importing parray;
  R: int = 0;
  W: int = 1;
  B: int = 2;
  flag[valres a:parray[N,below(3)],result l:below(N+1), r:
      below(N+1)]: PROCEDURE
    POST 0<=l and l<=r and r<=N;
    POST forall (i:int): 0<=i and i<l => a(i)=R;
    POST forall (i:int): l<=i and i<r => a(i)=W;
    POST forall (i:int): r<=i and i<N => a(i)=B;
    POST permutation(a_0,a);
  BEGIN
    k: pvar below(N+1);
    loop: SITUATION
    BEGIN
      * 0<=l and l<=k and k<=r and r<=N;
      * forall (i:int): 0<=i and i<l => a(i)=R;
      * forall (i:int): l<=i and i<k => a(i)=W;
      * forall (i:int): r<=i and i<N => a(i)=B;
      * permutation(a_0,a);
      ** r-k;
      IF
        [k=r]; exit
        [k<r and a(k)=W]; k:=k+1; decreasing goto loop
        [k<r and a(k)=R]; a:=swap(a,k,l); k,l:=k+1,l+1;
            decreasing goto loop
        [k<r and a(k)=B]; r:=r-1; a:=swap(a,k,r); decreasing
             goto loop
      ENDIF
    END loop
    IF
      [N>0];l,r,k := 0,N,0; goto loop
      [N=0];l,r := 0,0; exit
    ENDIF
  END flag
END flag
```

Figura 90: Representación textual del programa para resolver el problema de la bandera alemana

```
ctx_flag: theory
begin

    N: nat;
    importing parray;

    R: int =
        0;

    W: int =
        1;

    B: int =
        2;
end ctx_flag

spec_flag: theory
begin
    importing ctx_flag;
    a: var parray[N, below(3)];
    a_0: var parray[N, below(3)];
    l: var below(N + 1);
    r: var below(N + 1);

    pre_(a): bool =
        (true);

    post_(a_0, a, l, r): bool =
        (0 <= l
             and l <= r
             and r <= N)
            and (forall (i : int) :
                   0 <= i
                   and i < l
                   => a(i) = R)
            and (forall (i : int) :
                   l <= i
                   and i < r
                   => a(i) = W)
            and (forall (i : int) :
                   r <= i
                   and i < N
                   => a(i) = B)
            and (permutation(a_0, a));
end spec_flag
```

Figura 91: Teoría PVS del programa para resolver el problema de la bandera alemana - Parte 1

```
impl_flag: theory
begin
    importing spec_flag;
    a: var parray[N, below(3)];
    l: var below(N + 1);
    r: var below(N + 1);

    a_0: parray[N, below(3)];
    k: var below(N + 1);

    sit_ini_(a, l, r, k): bool =
        (((a = a_0)))
            and (spec_flag.pre_(a));

    sit_loop(a, l, r, k): bool =
        (0 <= l
             and l <= k
             and k <= r
             and r <= N)
            and (forall (i : int) :
                   0 <= i
                   and i < l
                   => a(i) = R)
            and (forall (i : int) :
                   l <= i
                   and i < k
                   => a(i) = W)
            and (forall (i : int) :
                   r <= i
                   and i < N
                   => a(i) = B)
            and (permutation(a_0, a));

    sit_fin_(a, l, r, k): bool =
        (spec_flag.post_(a_0, a, l, r));
    vc_ini_: lemma
        forall (a, l, r, k) :
              sit_ini_(a, l, r, k)
              => ((N > 0
                   or N = 0))
              and ((N > 0
                    => (lambda (l : below(N + 1), r : below(
                        N + 1), k : below(N + 1)) :
                           sit_loop(a, l, r, k))(0, N, 0))
                   and (N = 0
                        => (lambda (l : below(N + 1), r :
                             below(N + 1)) :
                           sit_fin_(a, l, r, k))(0, 0)))
```

Figura 92: Teoría PVS del programa para resolver el problema de la bandera alemana - Parte 2

```
    vc_loop: lemma
        forall (a, l, r, k) :
              sit_loop(a, l, r, k)
              => (lambda (v_loop_0 : int) :
                    ((k = r
                      or (k < r
                          and a(k) = W)
                      or (k < r
                          and a(k) = R)
                      or (k < r
                          and a(k) = B)))
                    and ((k = r
                          => sit_fin_(a, l, r, k))
                         and ((k < r
                               and a(k) = W)
                              => (lambda (k : below(N + 1))
                                  :
                                     (((0 <= r - k
                                        and r - k < v_loop_0)
                                         ))
                                     and (sit_loop(a, l, r, k
                                         )))(k + 1))
                         and ((k < r
                               and a(k) = R)
                              => (lambda (a : parray[N,
                                  below(3)]) :
                                     (lambda (k : below(N +
                                         1), l : below(N+ 1))
                                         :
                                         (((0 <= r - k
                                            and r - k <
                                                v_loop_0)))
                                         and (sit_loop(a, l, r
                                             , k)))(k + 1, l +
                                             1))
                               (swap(a, k, l)))
                         and ((k < r and a(k) = B)
                              => (lambda (r : below(N + 1))
                                  :
                                     (lambda (a : parray[N,
                                         below(3)]) :
                                         (((0 <= r - k and r -
                                              k < v_loop_0)))
                                             and (sit_loop(a, l
                                             , r, k)))(swap(a,k
                                             ,r)))(r - 1))))(r
                                             - k)
end impl_flag
```

Figura 93: Teoría PVS del programa para resolver el problema de la bandera alemana - Parte 3

```
def flagFun(a):
  R,W,B=0,1,2
  l,r,k=0,len(a),0
  while k<r:
    if a[k]==W:
      k=k+1
    else:
      if a[k]==R:
        aux=a[l]
        a[l]=a[k]
        a[k]=aux
        k,l=k+1,l+1
      else:
        if a[k]==B:
          r=r-1
          aux=a[r]
          a[r]=a[k]
          a[k]=aux
  return a
```

Figura 94: Implementación en Python del programa para resolver el problema de la bandera alemana

Promedio

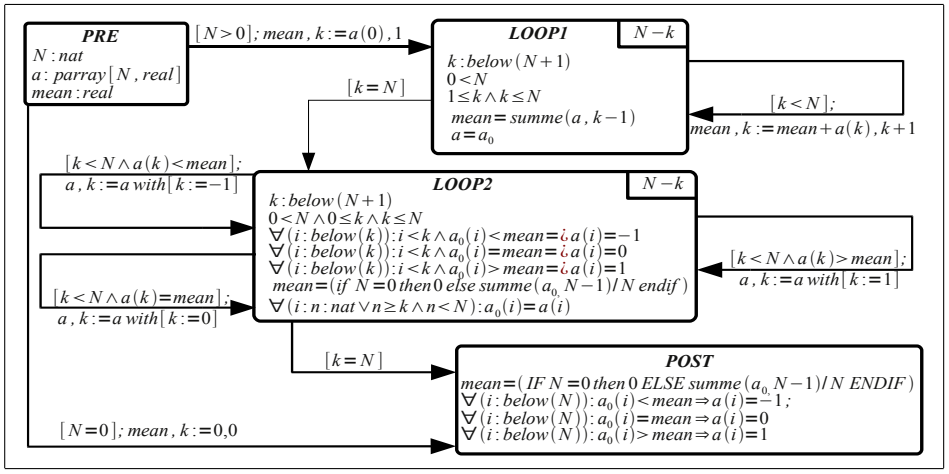

Figura 95: Diagrama de invariantes del programa que procesa el promedio de un arreglo

```
mean:  CONTEXT
BEGIN

  N:nat;

  importing parray;

  summe(a:parray[N,real],i:{x:int|x>=0 and x<N}): RECURSIVE
      real =
  IF i<0 THEN 0
  ELSE
    IF i = 0 THEN a(i)
      ELSE a(i) + summe(a,i-1)
    ENDIF
  ENDIF
  MEASURE i;


  mean[valres a:parray[N,real],mean:real]: PROCEDURE
  POST mean = (IF N=0 then 0 ELSE summe(a_0,N-1)/N ENDIF);
  POST forall (i:below(N)): a_0(i)<mean  => a(i)=-1;
  POST forall (i:below(N)): a_0(i)=mean  => a(i)=0;
  POST forall (i:below(N)): a_0(i)>mean  => a(i)=1;

  BEGIN
    k: pvar below(N+1);

    loop1: SITUATION
    BEGIN
      * 0<N;
      * 1<=k and k<=N;
      * mean=summe(a,k-1);
      * a=a_0;
      ** N-k;

      IF
        [k=N]; mean,k:=mean/N,0; GOTO loop2
        [k<N]; mean,k := mean + a(k), k+1;  decreasing GOTO
            loop1
      ENDIF
    END loop1
```

Figura 96: Representación textual del programa que procesa el promedio de un arreglo - Parte 1

```
    loop2: SITUATION
    BEGIN
      * 0<N;
      * 0<=k and k<=N;
      * forall (i:below(k)): i<k and a_0(i)<mean  => a(i)
          =-1;
      * forall (i:below(k)): i<k and a_0(i)=mean  => a(i)=0;
      * forall (i:below(k)): i<k and a_0(i)>mean  => a(i)=1;
      * mean=(if N=0 then 0 else summe(a_0,N-1)/N endif);
      * forall (i:{n:nat|n>=k and n<N}): a_0(i) = a(i);
      ** N-k;

      IF
        [k=N]; exit
        [k<N and a(k)<mean]; a,k := a with [k:=-1], k+1;
            decreasing GOTO loop2
        [k<N and a(k)=mean]; a,k := a with [k:=0], k+1;
            decreasing GOTO loop2
        [k<N and a(k)>mean]; a,k := a with [k:=1], k+1;
            decreasing GOTO loop2
      ENDIF
    END loop2

    IF
      [N=0]; mean,k := 0,0; exit
      [N>0]; mean,k := a(0),1; GOTO loop1
    ENDIF

  END mean

END mean
```

Figura 97: Representación textual del programa que procesa el promedio de un arreglo - Parte 2

```
ctx_mean: theory
begin

    N: nat;
    importing parray;

    summe(a : parray[N, real], i : { x : int | x >= 0
                                               and x < N }):
                                                   RECURSIVE
                                                   real =
        if i < 0 then
               0
            else
               if i = 0 then
                  a(i)
               else
                  a(i) + summe(a, i - 1)
               endif
            endif
        MEASURE i;
end ctx_mean

spec_mean: theory
begin
    importing ctx_mean;
    a: var parray[N, real];
    mean: var real;
    a_0: var parray[N, real];
    mean_0: var real;

    pre_(a, mean): bool =
        (true);

    post_(a_0, mean_0, a, mean): bool =
        mean = (if N = 0 then
                       0
                    else
                       summe(a_0, N - 1) / N
                    endif)
            and (forall (i : below(N)) :
                   a_0(i) < mean
                   => a(i) = -1)
            and (forall (i : below(N)) :
                   a_0(i) = mean
                   => a(i) = 0)
            and (forall (i : below(N)) :
                   a_0(i) > mean
                   => a(i) = 1);
end spec_mean
```

Figura 98: Teoría PVS del programa que procesa el promedio de un arreglo - Parte 1

```
impl_mean: theory
begin
    importing spec_mean;
    a: var parray[N, real];
    mean: var real;

    a_0: parray[N, real];

    mean_0: real;
    k: var below(N + 1);

    sit_ini_(a, mean, k): bool =
        ((a = a_0
              and mean = mean_0))
            and (spec_mean.pre_(a, mean));

    sit_loop1(a, mean, k): bool =
        0 < N
            and (1 <= k
                 and k <= N)
            and mean = summe(a, k - 1)
            and a = a_0;

    sit_loop2(a, mean, k): bool =
        0 < N
            and (0 <= k
                 and k <= N)
            and (forall (i : below(k)) :
                   i < k
                   and a_0(i) < mean
                   => a(i) = -1)
            and (forall (i : below(k)) :
                   i < k
                   and a_0(i) = mean
                   => a(i) = 0)
            and (forall (i : below(k)) :
                   i < k
                   and a_0(i) > mean
                   => a(i) = 1)
            and mean = (if N = 0 then
                           0
                        else
                           summe(a_0, N - 1) / N
                        endif)
            and (forall (i : { n : nat | n >= k and n < N })
                 : a_0(i) = a(i));
```

Figura 99: Teoría PVS del programa que procesa el promedio de un arreglo - Parte 2

```
    sit_fin_(a, mean, k): bool =
        (spec_mean.post_(a_0, mean_0, a, mean));
    vc_ini_: lemma
        forall (a, mean, k) :
              sit_ini_(a, mean, k)
              => ((N = 0
                   or N > 0))
              and ((N = 0
                    => (lambda (mean : real, k : below(N +
                        1)) :
                           sit_fin_(a, mean, k))(0, 0))
                   and (N > 0
                        => (lambda (mean : real, k : below(N
                             + 1)) :
                           sit_loop1(a, mean, k))(a(0), 1)))
    vc_loop1: lemma
        forall (a, mean, k) :
              sit_loop1(a, mean, k)
              => (lambda (v_loop1_0 : int) :
                    ((k = N
                      or k < N))
                    and ((k = N
                          => (lambda (mean : real, k : below
                              (N + 1)) :
                                 sit_loop2(a, mean, k))(mean
                                    / N, 0))
                         and (k < N
                              => (lambda (mean : real, k :
                                  below(N + 1)) :
                                     (((0 <= N - k
                                        and N - k < v_loop1_0
                                            )))
                                     and (sit_loop1(a, mean,
                                         k)))(mean + a(k), k +
                                          1)))) (N - k)
```

Figura 100: Teoría PVS del programa que procesa el promedio de un arreglo - Parte 3

```
    vc_loop2: lemma
        forall (a, mean, k) :
              sit_loop2(a, mean, k)
              => (lambda (v_loop2_0 : int) :
                    ((k = N
                      or (k < N
                          and a(k) < mean)
                      or (k < N
                          and a(k) = mean)
                      or (k < N
                          and a(k) > mean)))
                    and ((k = N
                          => sit_fin_(a, mean, k))
                         and ((k < N
                               and a(k) < mean)
                              => (lambda (a : parray[N, real
                                  ], k : below(N + 1)) :(((0
                                  <= N - k
                                        and N - k < v_loop2_0
                                            )))
                                    and (sit_loop2(a, mean,
                                        k)))(a with [k:= -1],
                                         k + 1))
                         and ((k < N
                               and a(k) = mean)
                              => (lambda (a : parray[N, real
                                  ], k : below(N + 1)) :
                                    (((0 <= N - k
                                        and N - k < v_loop2_0
                                            )))
                                    and (sit_loop2(a, mean,
                                        k)))(a with [k:= 0],
                                        k + 1))
                         and ((k < N
                               and a(k) > mean)
                              => (lambda (a : parray[N, real
                                  ], k : below(N + 1)) :
                                    (((0 <= N - k
                                        and N - k < v_loop2_0
                                            )))
                                    and (sit_loop2(a, mean,
                                        k)))(a with [k:= 1],
                                        k + 1))))(N - k)
end impl_mean
```

Figura 101: Teoría PVS del programa que procesa el promedio de un arreglo - Parte 4

```
def meanFun(a):
  b=a[:]
  if len(a)==0:
    return 0
  else:
    mean,k=a[0],1
    while k<len(a):
      mean,k=mean+a[k],k+1
         mean,k=mean/len(a),0
         while k<len(a):
      if a[k]<mean:
        b[k],k = -1, k+1
      else:
        if a[k]==mean:
          b[k],k = 0, k+1
        else:
          if a[k]>mean:
            b[k],k = 1, k+1
  return b
```

Figura 102: Implementación en Python del programa que procesa el promedio de un arreglo

Encontrar los M elementos más pequeños

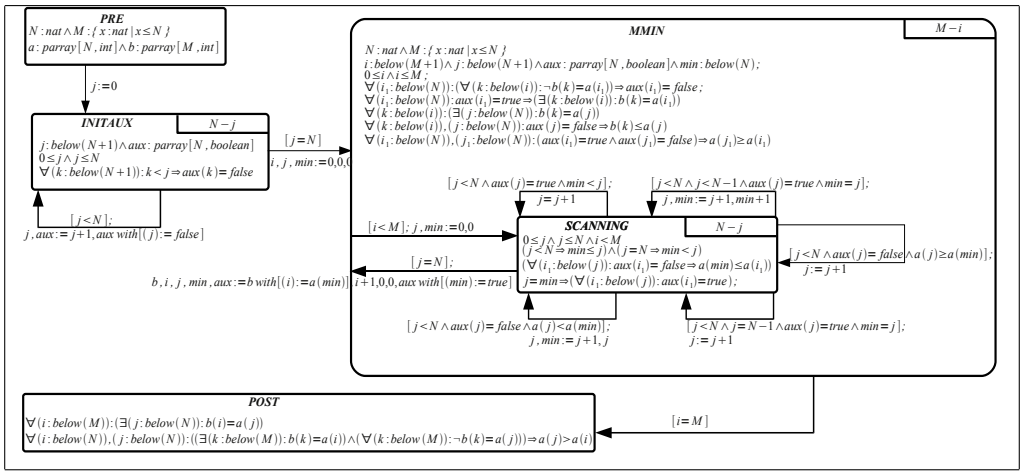

Figura 103: Diagrama de invariantes del programa para calcular los M elementos más pequeños de un arreglo

```
module : CONTEXT
BEGIN
  N:nat;
  M: {x:nat|x<=N};
  importing parray;
    small [a:parray[N,int],result b:parray[N,int]] :
        PROCEDURE
        POST forall (i: below(M)): (exists (j: below(N)) : b
            (i)=a(j));
        POST forall (i:below(N)),(j:below(N)): ((exists (k:
            below(M)): b(k)=a(i)) and (forall (k: below(M)):
            b(k)/=a(j))) => a(j)>a(i);
    BEGIN
        i:pvar below(M+1);
        j:pvar below(N+1);
        aux:pvar parray[N,boolean];
        min:pvar below(N);
        INITAUX : SITUATION
        BEGIN
            * 0<=j and j<=N;
            * forall (k: below(N+1)): k<j => aux(k)=false;
            ** N-j;
            IF
              [j<N]; b,i,j,min,aux:=b,0,j+1,0,aux with [(j)
                  :=false]; decreasing GOTO INITAUX
              [j=N];b,i,j,min,aux:=b,0,0,0,aux; goto MMIN
            ENDIF
        END INITAUX
        MMIN : SITUATION BEGIN
            * 0<=i and i<=M;
            * forall (i1:below(N)): (forall (k: below(i)): b
                (k)/=a(i1)) => aux(i1) = false;
            * forall (i1:below(N)): aux(i1) = true => (
                exists (k:below(i)): b(k)=a(i1));
            * forall (k: below(i)): (exists (j: below(N)) :
                b(k)=a(j));
            * forall (k:below(i)),(j:below(N)): aux(j)=false
                 => b(k)<=a(j);
            * forall (i1:below(N)),(j1:below(N)): (aux(i1)=
                true and aux(j1)=false) => a(j1)>=a(i1);
            ** M-i;
```

Figura 104: Representación textual del programa para calcular los M elementos más pequeños de un arreglo - Parte 1

```
                SCANNING : SITUATION
                BEGIN
                    * 0<=j and j<=N;
                    * i<M;
                    * j<N => min<=j;
                    * j=N => min<j;
                    * (forall (i1:below(j)): aux(i1)=false
                        => a(min)<=a(i1));
                    * j=min => (forall (i1:below(j)):aux(i1)
                        =true);
                    ** N-j;
                    IF
                            [j=N]; b,i,j,min,aux:= b with [(
                                i):=a(min)],i+1,0,0,aux with
                                [(min):=true]; decreasing
                                MMIN goto MMIN
                            [j<N and aux(j)=true and min<j];
                                 b,i,j,min,aux:=b,i,j+1,min,
                                aux; decreasing GOTO SCANNING
                            [j<N and j<N-1 and aux(j)=true
                                and min=j]; b,i,j,min,aux:=b,
                                i,j+1,min+1,aux; decreasing
                                GOTO SCANNING
                            [j<N and j=N-1 and aux(j)=true
                                and min=j]; b,i,j,min,aux:=b,
                                i,j+1,min,aux; decreasing
                                GOTO SCANNING
                            [j<N and aux(j)=false and a(j)<a
                                (min)];b,i,j,min,aux:=b,i,j
                                +1,j,aux; decreasing GOTO
                                SCANNING
                            [j<N and aux(j)=false and a(j)>=
                                a(min)];b,i,j,min,aux:=b,i,j
                                +1,min,aux; decreasing GOTO
                                SCANNING
                    ENDIF
                END SCANNING
            IF
                    [i<M]; b,i,j,min,aux:=b,i,0,0,aux; GOTO
                        SCANNING
                    [i=M]; EXIT
            ENDIF
        END MMIN
        b,i,j,min,aux:=b,0,0,0,aux; goto INITAUX
    END small
END module
```

Figura 105: Representación textual del programa para calcular los M elementos más pequeños de un arreglo - Parte 2

```
ctx_module: theory
begin

    N: nat;

    M: { x : nat | x <= N };
    importing parray;

end ctx_module

spec_small: theory
begin
    importing ctx_module;
    a: var parray[N, int];
    b: var parray[N, int];

    pre_(a): bool =
        (true);

    post_(a, b): bool =
        (forall (i : below(M)) :
               (exists (j : below(N)) :
                  b(i) = a(j)))
            and (forall (i : below(N)), (j : below(N)) :
                   ((exists (k : below(M)) :
                       b(k) = a(i))
                    and (forall (k : below(M)) :
                           b(k) /= a(j)))
                   => a(j) > a(i));
end spec_small

impl_small: theory
begin
    importing spec_small;
    b: var parray[N, int];

    a: parray[N, int];
    i: var below(M + 1);
    j: var below(N + 1);
    aux: var parray[N, boolean];
    min: var below(N);

    sit_ini_(b, i, j, aux, min): bool =
        ((true))
            and (spec_small.pre_(a));
```

Figura 106: Teoría PVS del programa para calcular los M elementos más pequeños de un arreglo - Parte 1

```
    sit_INITAUX(b, i, j, aux, min): bool =
        (0 <= j
             and j <= N)
            and (forall (k : below(N + 1)) :
                   k < j
                   => aux(k) = false);

    sit_MMIN(b, i, j, aux, min): bool =
        (0 <= i
             and i <= M)
            and (forall (i1 : below(N)) :
                   (forall (k : below(i)) :
                      b(k) /= a(i1))
                   => aux(i1) = false)
            and (forall (i1 : below(N)) :
                   aux(i1) = true
                   => (exists (k : below(i)) :
                         b(k) = a(i1)))
            and (forall (k : below(i)) :
                   (exists (j : below(N)) :
                      b(k) = a(j)))
            and (forall (k : below(i)), (j : below(N)) :
                   aux(j) = false
                   => b(k) <= a(j))
            and (forall (i1 : below(N)), (j1 : below(N)) :
                   (aux(i1) = true
                    and aux(j1) = false)
                   => a(j1) >= a(i1));

    sit_SCANNING(b, i, j, aux, min): bool =
        (sit_MMIN(b, i, j, aux, min))
            and (0 <= j
                 and j <= N)
            and i < M
            and (j < N
                 => min <= j)
            and (j = N
                 => min < j)
            and ((forall (i1 : below(j)) :
                    aux(i1) = false
                    => a(min) <= a(i1)))
            and (j = min
                 => (forall (i1 : below(j)) :
                       aux(i1) = true));
```

Figura 107: Teoría PVS del programa para calcular los M elementos más pequeños de un arreglo - Parte 2

```
    sit_fin_(b, i, j, aux, min): bool =
        (spec_small.post_(a, b));
    vc_ini_: lemma
        forall (b, i, j, aux, min) :
              sit_ini_(b, i, j, aux, min)
              => (lambda (b : parray[N, int], i : below(M +
                  1), j : below(N + 1),
                           min : below(N), aux : parray[N,
                               boolean]) :
                 sit_INITAUX(b, i, j, aux, min))(b, 0, 0, 0,
                    aux)
    vc_INITAUX: lemma
        forall (b, i, j, aux, min) :
              sit_INITAUX(b, i, j, aux, min)
              => (lambda (v_INITAUX_0 : int) :
                    ((j < N
                      or j = N))
                    and ((j < N
                          => (lambda (b : parray[N, int], i
                              : below(M + 1),
                                       j : below(N + 1), min
                                           : below(N), aux :
                                       parray[N, boolean]) :
                                 (((0 <= N - j
                                    and N - j < v_INITAUX_0))
                                     )
                                 and (sit_INITAUX(b, i, j,
                                     aux, min)))(b, 0, j + 1,
                                     0, aux with [(j):= false
                                     ]))
                         and
                         (j = N
                          => (lambda (b : parray[N, int], i
                              : below(M + 1), j : below(N +
                              1), min : below(N), aux :
                              parray[N, boolean]) : sit_MMIN(
                              b, i, j, aux, min))(b, 0, 0, 0,
                               aux))))(N - j)
```

Figura 108: Teoría PVS del programa para calcular los M elementos más pequeños de un arreglo - Parte 3

```
    vc_MMIN: lemma
        forall (b, i, j, aux, min) :
               sit_MMIN(b, i, j, aux, min)
               => ((i < M
                    or i = M))
               and ((i < M
                     => (lambda (b : parray[N, int], i :
                         below(M + 1), j : below(N + 1), min :
                          below(N), aux : parray[N, boolean])
                         : sit_SCANNING(b, i, j, aux, min))(b,
                          i, 0, 0, aux))
                    and
                    (i = M => sit_fin_(b, i, j, aux, min)))
    vc_SCANNING: lemma
        forall (b, i, j, aux, min) :
               sit_SCANNING(b, i, j, aux, min)
               => (lambda (v_MMIN_0 : int, v_SCANNING_0 : int
                   ) :
                      ((j = N
                        or (j < N
                            and aux(j) = true
                            and min < j)
                        or (j < N
                            and j < N - 1
                            and aux(j) = true
                            and min = j)
                        or (j < N
                            and j = N - 1
                            and aux(j) = true
                            and min = j)
                        or (j < N
                            and aux(j) = false
                            and a(j) < a(min))
                        or (j < N
                            and aux(j) = false
                            and a(j) >= a(min))))
```

Figura 109: Teoría PVS del programa para calcular los M elementos más pequeños de un arreglo - Parte 4

```
                    and ((j = N
                          => (lambda (b : parray[N, int], i
                              : below(M + 1),
                                       j : below(N + 1), min
                                           : below(N), aux :
                                       parray[N, boolean]) :
                                 (((0 <= M - i
                                    and M - i < v_MMIN_0)))
                                 and (sit_MMIN(b, i, j, aux,
                                     min)))(b with[(i):= a(min
                                     )],i + 1, 0, 0, aux with
                                     [(min):= true]))
                         and
                         ((j < N
                           and aux(j) = true
                           and min < j)
                          => (lambda (b : parray[N, int], i
                              : below(M + 1),
                                       j : below(N + 1), min
                                           : below(N), aux :
                                       parray[N, boolean]) :
                                 (((0 <= N - j
                                    and N - j < v_SCANNING_0)
                                     ))
                                 and (sit_SCANNING(b, i, j,
                                     aux, min)))(b, i, j + 1,
                                     min, aux))
                         and
                         ((j < N
                           and j < N - 1
                           and aux(j) = true
                           and min = j)
                          => (lambda (b : parray[N, int], i
                              : below(M + 1),
                                       j : below(N + 1), min
                                           : below(N), aux :
                                       parray[N, boolean]) :
                                 (((0 <= N - j
                                    and N - j < v_SCANNING_0)
                                     ))
                                 and (sit_SCANNING(b, i, j,
                                     aux, min)))(b, i, j + 1,
                                     min + 1, aux))
                         and
```

Figura 110: Teoría PVS del programa para calcular los M elementos más pequeños de un arreglo - Parte 5

```
                          ((j < N
                            and j = N - 1
                            and aux(j) = true
                            and min = j)
                           => (lambda (b : parray[N, int], i
                               : below(M + 1),
                                       j : below(N + 1), min
                                           : below(N), aux :
                                       parray[N, boolean]) :
                                  (((0 <= N - j
                                     and N - j < v_SCANNING_0)
                                      ))
                                  and (sit_SCANNING(b, i, j,
                                      aux, min)))(b, i, j + 1,
                                      min, aux))
                          and
                          ((j < N
                            and aux(j) = false
                            and a(j) < a(min))
                           => (lambda (b : parray[N, int], i
                               : below(M + 1),
                                       j : below(N + 1), min
                                           : below(N), aux :
                                       parray[N, boolean]) :
                                  (((0 <= N - j
                                     and N - j < v_SCANNING_0)
                                      ))
                                  and (sit_SCANNING(b, i, j,
                                      aux, min)))(b, i, j + 1,
                                      j, aux))
                          and
                          ((j < N
                            and aux(j) = false
                            and a(j) >= a(min))
                           => (lambda (b : parray[N, int], i
                               : below(M + 1),
                                       j : below(N + 1), min
                                           : below(N), aux :
                                       parray[N, boolean]) :
                                  (((0 <= N - j
                                     and N - j < v_SCANNING_0)
                                      ))
                                  and (sit_SCANNING(b, i, j,
                                      aux, min)))(b, i, j + 1,
                                      min, aux))))(M - i, N - j
                                      )
end impl_small
```

Figura 111: Teoría PVS del programa para calcular los M elementos más pequeños de un arreglo - Parte 6

```
def smallFun(a,M):
  b=range(M)
  aux=range(len(a))
  N=len(a)
  false=0
  true=1
  j=0
  while j<N:
    aux[j]=false
    j=j+1
  i,j,min1=0,0,0
  while i<M:
    j,min1=0,0
    while j<N:
      if aux[j]==true:
        if min1<j:
          i,j,min1=i,j+1,min1
        else:
          if j<N-1:
            i,j,min1=i,j+1,min1+1
           else:
            i,j,min1=i,j+1,min1
       else:
        if aux[j]==false and a[j]<a[min1]:
          i,j,min1=i,j+1,j
        else:
          if aux[j]==false and a[j]>=a[min1]:
            i,j,min1=i,j+1,min1
         b[i]=a[min1]
    aux[min1]=true
    i=i+1
  return b
```

Figura 112: Implementación en Python del programa para calcular los M elementos más pequeños de un arreglo

Mapeo

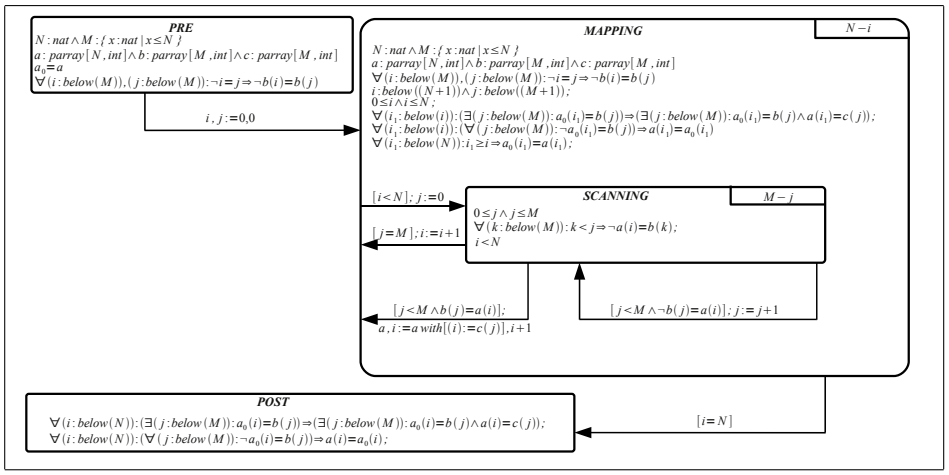

Figura 113: Diagrama de invariantes del programa para mapear elementos entre dos arreglos

```
mapping : CONTEXT
BEGIN

  N:nat;
  M: {x:nat|x<=N};

  importing parray;

    mapping [ b:parray[M,int], c:parray[M,int], valres a:
        parray[N,int]] : PROCEDURE
        PRE forall (i:below(M)),(j:below(M)): i/=j => b(i)/=
            b(j);
        POST forall (i: below(N)): (exists (j: below(M)) :
            a_0(i) = b(j)) => (exists (j: below(M)) : a_0(i)
            = b(j) and a(i) = c(j));
        POST forall (i: below(N)): (forall (j: below(M)) :
            a_0(i) /= b(j)) => a(i) = a_0(i);

    BEGIN
        i:pvar below((N+1));
        j:pvar below((M+1));

        MAPPING : SITUATION BEGIN
            * 0<=i and i<=N;
            * forall (i1: below(i)): (exists (j: below(M)) :
                a_0(i1) = b(j)) => (exists (j: below(M)) :
               a_0(i1) = b(j) and a(i1) = c(j));
            * forall (i1: below(i)): (forall (j: below(M)) :
                a_0(i1) /= b(j)) => a(i1) = a_0(i1);
            * forall (i1: below(N)): i1>=i => a_0(i1) = a(i1
               );
            ** N-i;
```

Figura 114: Representación textual del programa para mapear elementos entre dos arreglos - Parte 1

```
                SCANNING : SITUATION BEGIN
                    * 0<=j and j<=M;
                    * i<N;
                    * forall (k: below(M)): k<j => a(i)/=b(k
                      );
                    * 0<=i and i<=N;
                    * forall (i1: below(i)): (exists (j:
                      below(M)) : a_0(i1) = b(j)) => (
                      exists (j: below(M)) : a_0(i1) = b(j)
                       and a(i1) = c(j));
                    * forall (i1: below(i)): (forall (j:
                      below(M)) : a_0(i1) /= b(j)) => a(i1)
                       = a_0(i1);
                    * forall (i1: below(N)): i1>=i => a_0(i1
                      ) = a(i1);
                    ** M-j;

                    IF
                            [j=M];i:= i+1; decreasing goto
                                MAPPING
                            [j<M and b(j)=a(i)];a,i:=a with
                                [(i):=c(j)],i+1; decreasing
                                GOTO MAPPING
                            [j<M and b(j)/=a(i)];j:= j+1;
                                decreasing GOTO SCANNING
                    ENDIF
                END SCANNING

            IF
                    [i<N]; j:=0; GOTO SCANNING
                    [i=N]; EXIT
            ENDIF
        END MAPPING

        i,j:=0,0; goto MAPPING

    END mapping

END mapping
```

Figura 115: Representación textual del programa para mapear elementos entre dos arreglos - Parte 2

```
ctx_mapping: theory
begin

    N: nat;

    M: { x : nat | x <= N };
    importing parray;
end ctx_mapping

spec_mapping: theory
begin
    importing ctx_mapping;
    b: var parray[M, int];
    c: var parray[M, int];
    a: var parray[N, int];
    a_0: var parray[N, int];

    pre_(b, c, a): bool =
        (forall (i : below(M)), (j : below(M)) :
               i /= j
               => b(i) /= b(j));

    post_(b, c, a_0, a): bool =
        (forall (i : below(N)) :
               (exists (j : below(M)) :
                  a_0(i) = b(j))
               => (exists (j : below(M)) :
                     a_0(i) = b(j)
                     and a(i) = c(j)))
            and (forall (i : below(N)) :
                   (forall (j : below(M)) :
                      a_0(i) /= b(j))
                   => a(i) = a_0(i));
end spec_mapping
```

Figura 116: Teoría PVS del programa para mapear elementos entre dos arreglos - Parte 1

```
impl_mapping: theory
begin
    importing spec_mapping;
    a: var parray[N, int];

    b: parray[M, int];

    c: parray[M, int];

    a_0: parray[N, int];
    i: var below((N + 1));
    j: var below((M + 1));

    sit_ini_(a, i, j): bool =
        (((a = a_0)))
            and (spec_mapping.pre_(b, c, a));

    sit_MAPPING(a, i, j): bool =
        (0 <= i
             and i <= N)
            and (forall (i1 : below(i)) :
                   (exists (j : below(M)) :
                      a_0(i1) = b(j))
                   => (exists (j : below(M)) :
                         a_0(i1) = b(j)
                         and a(i1) = c(j)))
            and (forall (i1 : below(i)) :
                   (forall (j : below(M)) :
                      a_0(i1) /= b(j))
                   => a(i1) = a_0(i1))
            and (forall (i1 : below(N)) :
                   i1 >= i
                   => a_0(i1) = a(i1));
```

Figura 117: Teoría PVS del programa para mapear elementos entre dos arreglos - Parte 2

```
    sit_SCANNING(a, i, j): bool =
        (sit_MAPPING(a, i, j))
            and (0 <= j
                 and j <= M)
            and i < N
            and (forall (k : below(M)) :
                   k < j
                   => a(i) /= b(k))
            and (0 <= i
                 and i <= N)
            and (forall (i1 : below(i)) :
                   (exists (j : below(M)) :
                      a_0(i1) = b(j))
                   => (exists (j : below(M)) :
                         a_0(i1) = b(j)
                         and a(i1) = c(j)))
            and (forall (i1 : below(i)) :
                   (forall (j : below(M)) :
                      a_0(i1) /= b(j))
                   => a(i1) = a_0(i1))
            and (forall (i1 : below(N)) :
                   i1 >= i
                   => a_0(i1) = a(i1));

    sit_fin_(a, i, j): bool =
        (spec_mapping.post_(b, c, a_0, a));
    vc_ini_: lemma
        forall (a, i, j) :
              sit_ini_(a, i, j)
              => (lambda (i : below((N + 1)), j : below((M +
                   1))) :
                    sit_MAPPING(a, i, j))(0, 0)
```

Figura 118: Teoría PVS del programa para mapear elementos entre dos arreglos - Parte 3

```
    vc_MAPPING: lemma
        forall (a, i, j) :
              sit_MAPPING(a, i, j)
              => (lambda (v_MAPPING_0 : int) :
                    ((i < N
                      or i = N))
                    and ((i < N
                          => (lambda (j : below((M + 1))) :
                                (((0 <= N - i
                                   and N - i <= v_MAPPING_0)
                                    ))
                                and (sit_SCANNING(a, i, j)))
                                    (0))
                           and (i = N
                            => sit_fin_(a, i, j)))))(N - i)
    vc_SCANNING: lemma
        forall (a, i, j) :
              sit_SCANNING(a, i, j)
              => (lambda (v_MAPPING_0 : int, v_SCANNING_0 :
                  int) :
                    ((j = M or (j < M and b(j) = a(i)) or (j
                        < M and b(j) /= a(i))))
                    and ((j = M
                          => (lambda (i : below((N + 1))) :
                                (((0 <= N - i and N - i <
                                    v_MAPPING_0))) and (
                                    sit_MAPPING(a, i, j)))(i
                                    + 1))
                           and ((j < M and b(j) = a(i)) => (
                               lambda (a : parray[N, int], i :
                               below((N + 1))) : (((0 <= N - i
                               and N - i < v_MAPPING_0))) and (
                               sit_MAPPING(a, i, j)))(a with [(
                               i) := c(j)], i + 1))
                           and ((j < M and b(j) /= a(i)) => (
                               lambda (j : below((M + 1))) :
                               (((0 <= M - j and M - j <
                               v_SCANNING_0) and (0 <= N - i
                               and N - i <= v_MAPPING_0))) and
                               (sit_SCANNING(a, i, j)))(j + 1))
                               ))(N - i, M - j)
end impl_mapping
```

Figura 119: Teoría PVS del programa para mapear elementos entre dos arreglos - Parte 4

```
def mappingFun(a,b,c):
  i,j=0,0
  while i<len(a):
    j=0
    while j<len(b):
      if b[j]==a[i]:
        a[i]=c[j]
        break
      else:
        j=j+1
    i=i+1
  return a
```

Figura 120: Implementación en Python del programa para mapear elementos entre dos arreglos

Merge

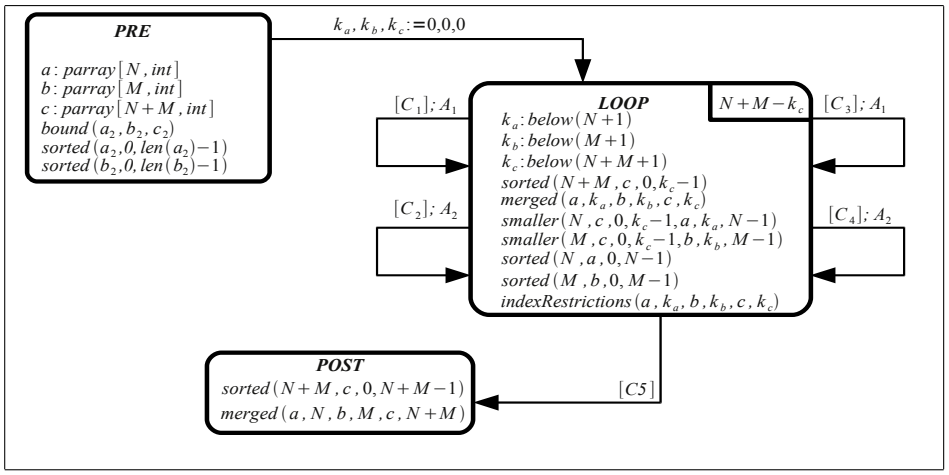

Figura 121: Diagrama de invariantes del programa para realizar el merge ordenado de dos arreglos

```
Merge: Context
begin

  importing parray;

  N,M: nat;

  sorted(L:nat,arr:parray[L,int],a:int,b:int): bool = forall
       (i:int),(j:int): ((0<= a and a <= i and  i < j and j<=
      b and b<L) => arr(i) <= arr(j));

  merged(a:parray[N,int],ka:nonneg_int,b: parray[M,int],kb:
      nonneg_int,c: parray[N+M,int],kc:nonneg_int): bool
   = exists  (f:[below[kc]->below[kc]]): bijective?[below(kc
       ),below(kc)](f) and (forall (i:below(kc)): (ka<N and
       kb<M and kc<N+M and ka=kb+kc =>((i<ka => c(f(i))=a(i))
        and  (i>=ka =>  c(f(i))=b(i-ka)))));

  smaller(L:nat,X:parray[N+M,int], a:int, b:int, Y:parray[L,
      int], c:int, d:int): bool = forall (i:int), (j:int) :
      0<=a and a<=i and i<=b and b<=N+M-1 and 0<=c and c<=j
      and j<=d and d<=L-1 => X(i)<=Y(j);

  indexRestrictions(a:parray[N,int],ka:nonneg_int,b:parray[M
      ,int],kb:nonneg_int,c:parray[N+M,int],kc:nonneg_int):
      bool =  ka+kb=kc and  ((ka < N and kb < M) or (ka = N
      and kb < M) or (ka < N and kb = M) or (ka = N and kb =
      M));

  C1(a:parray[N,int],ka:nonneg_int,b:parray[M,int],kb:
      nonneg_int,c:parray[N+M,int],kc:nonneg_int): bool = ka
      < N and kb < M and a(ka) > b(kb);

  C2(a:parray[N,int],ka:nonneg_int,b:parray[M,int],kb:
      nonneg_int,c:parray[N+M,int],kc:nonneg_int): bool = ka
      < N and kb < M and a(ka) <= b(kb);

  C3(a:parray[N,int],ka:nonneg_int,b:parray[M,int],kb:
      nonneg_int,c:parray[N+M,int],kc:nonneg_int): bool = ka
      = N and kb < M;

  C4(a:parray[N,int],ka:nonneg_int,b:parray[M,int],kb:
      nonneg_int,c:parray[N+M,int],kc:nonneg_int): bool = ka
      < N and kb = M;

  C5(a:parray[N,int],ka:nonneg_int,b:parray[M,int],kb:
      nonneg_int,c:parray[N+M,int],kc:nonneg_int): bool = ka
      = N and kb = M and kc = N + M;
```

Figura 122: Representación textual del programa para realizar el merge ordenado de dos arreglos - Parte 1

```
  mergeproc[a: parray[N,int],b: parray[M,int],result c:
      parray[N+M,int]]:procedure
    PRE sorted(N,a, 0, N-1) and sorted(M,b, 0, M-1);
    POST  sorted(N+M,c, 0, N+M-1);
    POST  merged(a,N, b, M, c, N+M);
  begin


    ka:pvar below(N+1);
    kb:pvar below(M+1);
    kc:pvar below(N+M+1);


    LOOP:situation
    begin
      * sorted(N+M,c, 0, kc-1);
      * merged(a, ka, b, kb, c, kc);
      * smaller(N,c, 0, kc-1, a, ka, N-1);
      * smaller(M,c, 0, kc-1, b, kb, M-1);
      * sorted(N,a, 0, N-1);
      * sorted(M,b, 0, M-1);
      * indexRestrictions(a,ka,b,kb,c,kc);
      ** N+M-kc;

      if
        [C1(a,ka,b,kb,c,kc)];c,ka,kb,kc:=c with [(kc):=b(kb)
            ],ka,kb+1,kc+1; decreasing goto LOOP
        [C2(a,ka,b,kb,c,kc)];c,ka,kb,kc:=c with [(kc):=a(ka)
            ],ka+1,kb,kc+1; decreasing goto LOOP
        [C3(a,ka,b,kb,c,kc)];c,ka,kb,kc:=c with [(kc):=b(kb)
            ],ka,kb+1,kc+1; decreasing goto LOOP
        [C4(a,ka,b,kb,c,kc)];c,ka,kb,kc:=c with [(kc):=a(ka)
            ],ka+1,kb,kc+1; decreasing goto LOOP
        [C5(a,ka,b,kb,c,kc)]; exit
      endif

    end LOOP

    ka,kb,kc:=0,0,0;goto LOOP

  end  mergeproc

end Merge
```

Figura 123: Representación textual del programa para realizar el merge ordenado de dos arreglos - Parte 2

```
ctx_Merge: theory
begin
    importing parray;

    N, M: nat;

    sorted(L : nat, arr : parray[L, int], a : int, b : int):
         bool =
        forall (i : int), (j : int) :
               ((0 <= a
                 and a <= i
                 and i < j
                 and j <= b
                 and b < L)
                => arr(i) <= arr(j));

    merged(a : parray[N, int], ka : nonneg_int, b : parray[M
        , int], kb : nonneg_int,
          c : parray[N + M, int], kc : nonneg_int): bool =
        exists (f : [below[kc] -> below[kc]]) :
               bijective?[below(kc), below(kc)](f)
               and (forall (i : below(kc)) :
                      (ka < N
                       and kb < M
                       and kc < N + M
                       and ka = kb + kc
                       => ((i < ka
                            => c(f(i)) = a(i))
                           and (i >= ka
                                => c(f(i)) = b(i - ka)))));

    smaller(L : nat, X : parray[N + M, int], a : int, b :
        int,Y : parray[L,int], c : int, d : int): bool =
        forall (i : int), (j : int) :
               0 <= a
               and a <= i
               and i <= b
               and b <= N + M - 1
               and 0 <= c
               and c <= j
               and j <= d
               and d <= L - 1
               => X(i) <= Y(j);
```

Figura 124: Teoría PVS del programa para realizar el merge ordenado de dos arreglos - Parte 1

```
    indexRestrictions(a : parray[N, int], ka : nonneg_int, b
         : parray[M, int],
          kb : nonneg_int, c : parray[N + M, int], kc :
              nonneg_int): bool =
        ka + kb = kc
            and ((ka < N
                  and kb < M)
                 or (ka = N
                     and kb < M)
                 or (ka < N
                     and kb = M)
                 or (ka = N
                     and kb = M));

    C1(a : parray[N, int], ka : nonneg_int, b : parray[M,
        int], kb : nonneg_int,
          c : parray[N + M, int], kc : nonneg_int): bool =
        ka < N
            and kb < M
            and a(ka) > b(kb);

    C2(a : parray[N, int], ka : nonneg_int, b : parray[M,
        int], kb : nonneg_int,
          c : parray[N + M, int], kc : nonneg_int): bool =
        ka < N
            and kb < M
            and a(ka) <= b(kb);

    C3(a : parray[N, int], ka : nonneg_int, b : parray[M,
        int], kb : nonneg_int,
          c : parray[N + M, int], kc : nonneg_int): bool =
        ka = N
            and kb < M;

    C4(a : parray[N, int], ka : nonneg_int, b : parray[M,
        int], kb : nonneg_int,
          c : parray[N + M, int], kc : nonneg_int): bool =
        ka < N
            and kb = M;

    C5(a : parray[N, int], ka : nonneg_int, b : parray[M,
        int], kb : nonneg_int,
          c : parray[N + M, int], kc : nonneg_int): bool =
        ka = N
            and kb = M
            and kc = N + M;
end ctx_Merge
```

Figura 125: Teoría PVS del programa para realizar el merge ordenado de dos arreglos - Parte 2

```
spec_mergeproc: theory
begin
    importing ctx_Merge;
    a: var parray[N, int];
    b: var parray[M, int];
    c: var parray[N + M, int];

    pre_(a, b): bool =
        (sorted(N, a, 0, N - 1)
             and sorted(M, b, 0, M - 1));

    post_(a, b, c): bool =
        (sorted(N + M, c, 0, N + M - 1))
            and (merged(a, N, b, M, c, N + M));
end spec_mergeproc

impl_mergeproc: theory
begin
    importing spec_mergeproc;
    c: var parray[N + M, int];

    a: parray[N, int];

    b: parray[M, int];
    ka: var below(N + 1);
    kb: var below(M + 1);
    kc: var below(N + M + 1);

    sit_ini_(c, ka, kb, kc): bool =
        ((true))
            and (spec_mergeproc.pre_(a, b));

    sit_LOOP(c, ka, kb, kc): bool =
        (sorted(N + M, c, 0, kc - 1))
            and (merged(a, ka, b, kb, c, kc))
            and (smaller(N, c, 0, kc - 1, a, ka, N - 1))
            and (smaller(M, c, 0, kc - 1, b, kb, M - 1))
            and (sorted(N, a, 0, N - 1))
            and (sorted(M, b, 0, M - 1))
            and (indexRestrictions(a, ka, b, kb, c, kc));
```

Figura 126: Teoría PVS del programa para realizar el merge ordenado de dos arreglos - Parte 3

```
    sit_fin_(c, ka, kb, kc): bool =
        (spec_mergeproc.post_(a, b, c));
    vc_ini_: lemma
        forall (c, ka, kb, kc) :
              sit_ini_(c, ka, kb, kc)
              => (lambda (ka : below(N + 1), kb : below(M +
                  1), kc : below(N + M + 1)) :
                sit_LOOP(c, ka, kb, kc))(0, 0, 0)
    vc_LOOP: lemma
        forall (c, ka, kb, kc) :
              sit_LOOP(c, ka, kb, kc)
              => (lambda (v_LOOP_0 : int) :
                    (((C1(a, ka, b, kb, c, kc))
                      or (C2(a, ka, b, kb, c, kc))
                      or (C3(a, ka, b, kb, c, kc))
                      or (C4(a, ka, b, kb, c, kc))
                      or (C5(a, ka, b, kb, c, kc))))
                    and (((C1(a, ka, b, kb, c, kc))
                          => (lambda (c : parray[N + M, int
                              ], ka : below(N + 1),
                                       kb : below(M + 1), kc
                                           : below(N + M + 1))
                                        :
                                 (((0 <= N + M - kc and N + M
                                      - kc < v_LOOP_0)))
                                 and (sit_LOOP(c, ka, kb, kc)
                                     ))(c with [(kc):= b(kb)],
                                      ka, kb + 1, kc + 1))
```

Figura 127: Teoría PVS del programa para realizar el merge ordenado de dos arreglos - Parte 4

```
                          and ((C2(a, ka, b, kb, c, kc)) => (
                              lambda (c : parray[N + M, int],
                              ka : below(N + 1), kb : below(M
                              + 1), kc : below(N + M + 1)) :
                              (((0 <= N + M - kc and N + M -
                              kc < v_LOOP_0))) and (sit_LOOP(c
                              , ka, kb, kc)))(c with [(kc):= a
                              (ka)], ka + 1, kb, kc + 1))
                          and ((C3(a, ka, b, kb, c, kc)) => (
                              lambda (c : parray[N + M, int],
                              ka : below(N + 1), kb : below(M
                              + 1), kc : below(N + M + 1)) :
                              (((0 <= N + M - kc and N + M -
                              kc < v_LOOP_0))) and (sit_LOOP(c
                              , ka, kb, kc)))(c with [(kc):= b
                              (kb)], ka, kb + 1, kc + 1))
                          and ((C4(a, ka, b, kb, c, kc))
                               => (lambda (c : parray[N + M,
                                   int], ka : below(N + 1), kb
                                    : below(M + 1), kc : below
                                   (N + M + 1)) : (((0 <= N +
                                   M - kc and N + M - kc <
                                   v_LOOP_0))) and (sit_LOOP(c
                                   , ka, kb, kc)))(c with [(kc
                                   ) := a(ka)], ka + 1, kb, kc
                                    + 1))
                          and ((C5(a, ka, b, kb, c, kc)) =>
                              sit_fin_(c, ka, kb, kc))))(N + M
                               - kc)
end impl_mergeproc
```

Figura 128: Teoría PVS del programa para realizar el merge ordenado de dos arreglos - Parte 5

```
def mergeFun(a,b):
  ka,kb,kc=0,0,0
  c=a[:]+b[:]
  N=len(a)
  M=len(b)
  while kc<(N+M):
    if ka < N and kb < M and a[ka] > b[kb]:
      c[kc],ka,kb,kc=b[kb],ka,kb+1,kc+1
    else:
      if ka < N and kb < M and a[ka] <= b[kb]:
        c[kc],ka,kb,kc=a[ka],ka+1,kb,kc+1
      else:
        if ka == N and kb < M:
          c[kc],ka,kb,kc=b[kb],ka,kb+1,kc+1
        else:
          if ka < N and kb == M:
            c[kc],ka,kb,kc=a[ka],ka+1,kb,kc+1
  return c
```

Figura 129: Implementación en Python del programa para realizar el merge ordenado de dos arreglos

Referencias

1. Ralph-Johan Back. Invariant Based Programming.
2. Ralph-Johan Back. Invariant Based Programming Revisited. TUCS Technical Report No 661, January 2005
3. Ralph-Johan Back | Johannes Eriksson | Magnus Myreen. Testing and Verifying Invariant Based Programs in the Socos Environment. TUCS Technical Report No 797, December 2006
4. Ralph-Johan Back. Incremental Software Construction with Refinement Diagrams. TUCS Technical Report No 660, January 2005
5. Ralph-Johan Back | Viorel Preoteasa. Semantics and Proof Rules of Invariant Based Programs. TUCS Technical Report No 903, June 2008
6. Ralph - Johan Back and Magnus Myreen. Tool Support for Invariant Based Programming. TUCS Technical Report No 666, February 2005.
7. Ralph-Johan Back, Johannes Eriksson and Magnus Myreen. Testing and Verifying Invariant Based Programs in the Socos Environment.
8. Johannes Eriksson. Socos: An Introductory Tutorial.
9. Jeannette M. Wing. Carnegie Mellon University. A Specifier's Introduction to formal methods.
10. Eclipse platform homepage. http://www.eclipse.org
11. Jean-Charles Mammana , Romain Meson, Jonathan Gramain. GEF (Graphical Editing Framework) Tutorial, Document version: 1.1. INRIA, Initial release: October 17, 2007, April 26.
12. Gef homepage. http://www.eclipse.org/gef.
13. Ralph-Johan Back. Invariant Based Programming, Basic Approach and Teaching Experience book (pages 227-244). Formal Aspects of Computing, 2009.
14. Ralph-Johan Back. Invariant Based Programming, Proceedings of the 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency book (pages 1-18). 2006.
15. J. C. Reynolds. Programming with transition diagrams, Programming Methodology book (pages 153-165). 1978.
16. M. H. van Emden. Programming with Verification Conditions, IEEE Transactions on Software Engineering (pages 148-159). 1979.
17. Ralph-Johan Back. Invariant Based Programs and Their Correctness, Automatic Program Construction Techniques book (pages 223-242). 1983.

18. Edsger W. Dijkstra. A constructive approach to the problem of program correctness, Numerical Mathematics Journal (174-186). 1968.

19. N. Shankar, S. Owre, J. M. Rushby and D. W. J. Stringer-Calvert. PVS System Guide. SRI International Computer Science Laboratory. Versión 2.4. November 2001.

20. N. Shankar, S. Owre, J. M. Rushby and D. W. J. Stringer-Calvert. PVS Language Reference. SRI International Computer Science Laboratory. Versión 2.4. November 2001.

21. N. Shankar, S. Owre, J. M. Rushby and D. W. J. Stringer-Calvert. PVS Prover Guide. SRI International Computer Science Laboratory. Versión 2.4. November 2001.

22. S. Owre and N. Shankar. The PVS Prelude Library. CSL Technical Report SRI-CSL-03-01. March 7, 2003

23. Myla Archer, Ben Di Vito and César Munoz. Developing User Strategies in PVS: A Tutorial. Naval Research Laboratory, NASA Langley Research Center and National Institute of Aerospace. 2003.

24. Bruno Dutertre and Leonardo de Moura. The YICES SMT Solver. Computer Science Laboratory, SRI International 333 Ravenswood Avenue, Menlo Park, CA 94025 - USA