INFORME DE PRÁCTICAS

Repositorio del proyecto:

- ***https://github.com/andreu-barro/VVS-DFA-JAVA***
- ***https://github.com/srjavimoure/VVS-DFA-C***

Participantes del proyecto:

- ***F. Javier Moure López,***
- ***Emma Oitavén Carracedo,***
- ***Xoan Andreu Barro Torres***

Validación e Verificación de Software

1. Descripción del proyecto

Aplicación que modifica un Autómata Finito Determinista (DFA), eliminando estados y transiciones superfluas de modo que devuelva un DFA equivalente más sencillo.

Esta aplicación fue realizada en los lenguajes de programación Java y C.

2. Estado actual

Nuestra aplicación se divide en los siguientes módulos:

- Clase *State*: Representación de un estado del autómata.
- Clase *Symbol*: Representación de un símbolo del lenguaje aceptado por el autómata.
- Clase *GenList*: Implementación de una lista de elementos de tipo no predeterminado
- Clase *Alphabet*: Lista de símbolos de un lenguaje, Alfabeto del autómata.
- Clase *Transition*: Lista de transiciones del autómata. Contienen un estado inicial, un estado final y un símbolo de transición.
- Clase *DFA*: La representación del autómata. Implementación de varias funcionalidades como obtención de estados conectados, generación de autómata a partir de una cadena de texto, y más.

El objetivo es probar que todas estas módulos funcionan correctamente, para ello aplicaremos pruebas de unidad, pruebas dinámicas, pruebas de rendimiento y chequearemos el estilo de programación. Una vez usemos las diversas herramientasa nuestra disposición, como por ejemplo cobertura y PIT (Java) o lcov y Unity (C), podremos demostrar que nuestra aplicación, y los módulos que la componen, son correctos y estables y dispondremos de una serie de pruebas para demostrarlo.

3. Especificación de pruebas

Para el repositorio de Java: Ir a doc/Informe/Informes.

Para el repositorio de C: Ir a doc/

En cada documento se encuentra la descripción de las pruebas a realizar, describiendo la entrada proporcionada, salida esperada (lo cual consideramos nuestro criterio de éxito), función a probar y el tipo de prueba, todas ellas divididas primero en documentos por cada clase de prueba (unidad, integración,...) y segundo en apartados según el módulo.

Para referencia y facilidad de búsqueda, cada prueba cuenta con un código de identificación, referenciado tanto en los documentos de especificación de pruebas como en el código (p.ej. *JAVA-SYM-NEW-01*).

Las pruebas definidas son comunes a ambos lenguajes, con añadidos concretos necesarios según la idiosincracia propia de cada lenguaje de programación.

4. Registro de pruebas

4.1. Pruebas unidad: JUnit y Unity

Estas herramientas se utilizan para realizar pruebas unitarias sobre nuestra aplicación, nos sirven para encontrar errores y solventar los problemas en la programación de forma manual. Se realizan pruebas de todas las funciones implementadas en la aplicación. Los casos de prueba se implementan manualmente como parte de funciones de prueba, utilizamos la directiva assertEqual(Expected, Expr).

4.2. Pruebas de unidad: Mockito y CMock

Para poder crear un buen conjunto de pruebas unitarias, es necesario que nos centremos exclusivamente en la clase a testear, simulando el funcionamiento de las capas inferiores (pensad por ejemplo en olvidarnos de la capa de acceso a datos, DAO). De esta manera estaremos creando test unitarios potentes que nos permitiría detectar y solucionar los errores que tengamos o que se cometan durante el futuro del desarrollo de nuestra aplicación. Para esta tarea nos apoyaremos en el uso de mock objects, que no son más que objetos que simulan parte del comportamiento de una clase.

Los mocks son sólo necesarios para los módulos Transition, Alphabet, y DFA, pues dependen de otros módulos para su funcionamiento.

4.3. Pruebas basadas en propiedades: Quickcheck y QuickCheck4c

QuickCheck es una herramienta para generar automáticamente y ejecutar casos de prueba aleatorios, basados en especificaciones de propiedades. Es decir, ejecutar pruebas con esta herramienta, significa instanciar las propiedades n veces. Las pruebas se detienen al encontrar un caso concreto en el que la propiedad no se cumple (contraejemplo). La ejecución con éxito significa que ninguno de los casos generados incumplió la propiedad.
Se crean generadores de:

- GeneradorAlphabet
- GeneradorDFA
- GeneradorState
- GeneradorSymbol
- GeneradorTransition
- GeneradorGenList

Java: La programación de generadores para aplicar Quickcheck se realiza en el siguiente paquete: es.udc.fic.vvs.vvsproject.generadorTest

C: Las pruebas de QCC se ejecutan independientemente de las de Unity. Se pueden encontrar en los archivos con nombre modulo-qcc.c

4.4. Validación de calidad de las pruebas: Cobertura y Icov

Estas herramientas nos permiten comprobar qué parte del código estamos cubriendo con nuestras pruebas, revelando partes del código pendientes de prueba y permitiéndonos hacer pruebas más exhaustivas.
De esta forma Cobertura se convierte en una potente herramienta de trabajo, ya que lo podemos usar como medida de calidad (mientras más código tengamos probado, más garantías tenemos de que cubrimos todos nuestros casos de uso). Nuestro objetivo es llegar a cobertura cercana al 100 % para nuestros módulos.

4.5. Pruebas no funcionales: JETM

JETM permite realizar pruebas de rendimiento y comprobar la velocidad de ejecuciones al implementar unas pruebas con n iteraciones. Se pueden generar test con múltiples iteraciones para detectar problemas de rendimiento.

Pruebas que se realizarán en el paquete es.udc.vvs.dfa.rendimiento

En C, nos valdremos del comando *time* en consola de comandos para determinar los tiempos de ejecución de la aplicación compilada bajo distintas estrategias de optimización de código.

4.6. Validación de calidad de las pruebas (mutación testing): PIT y Milu

El método que utilizaban en el sistema para realizar estas mediciones lo denominaba "Programa mutado". Básicamente, el Mutation testing consiste en introducir pequeñas modificaciones en el código fuente de la aplicación, a las que denominaremos mutaciones o mutantes.

Si las pruebas pasan al ejecutarse sobre el mutante, el mutante sobrevive. Si las pruebas no pasan al ejecutarse sobre el mutante, el mutante muere El objetivo es que todos los mutantes mueran, así podremos decir que el test responde a la definición concreta del código y que lo prueba correctamente. Este concepto se basa en dos hipótesis:

- Hipótesis del programador competente: La mayoría de los errores introducidos por programadores Senior consisten en pequeños errores sintácticos.
- Hipótesis del efecto de acoplamiento: Pequeños fallos acoplados pueden dar lugar a otros problemas mayores.

Problemas de mayor orden serán revelados por mutantes de mayor orden, que se crean mediante la unión de multiples mutaciones.

4.7. Pruebas estructurales: CheckStyle y CPPCheck

Checkstyle es una herramienta de desarrollo que ayudar a los programadores a escribir código Java para que se adhiera a un estándar de codificación. Automatiza el proceso de comprobación de código Java. Esto lo hace ideal para los proyectos a los que se desea aplicar un estándar de codificación.

Checkstyle es altamente configurable y se puede hacer para apoyar casi cualquier estándar de codificación. De tal manera que se puedan suministrar diferentes estándares de código para su posterior comprobación mediante la herramienta.

Reglas del CheckStyle El conjunto de reglas disponible es muy completo y está clasificado en los siguientes grupos:

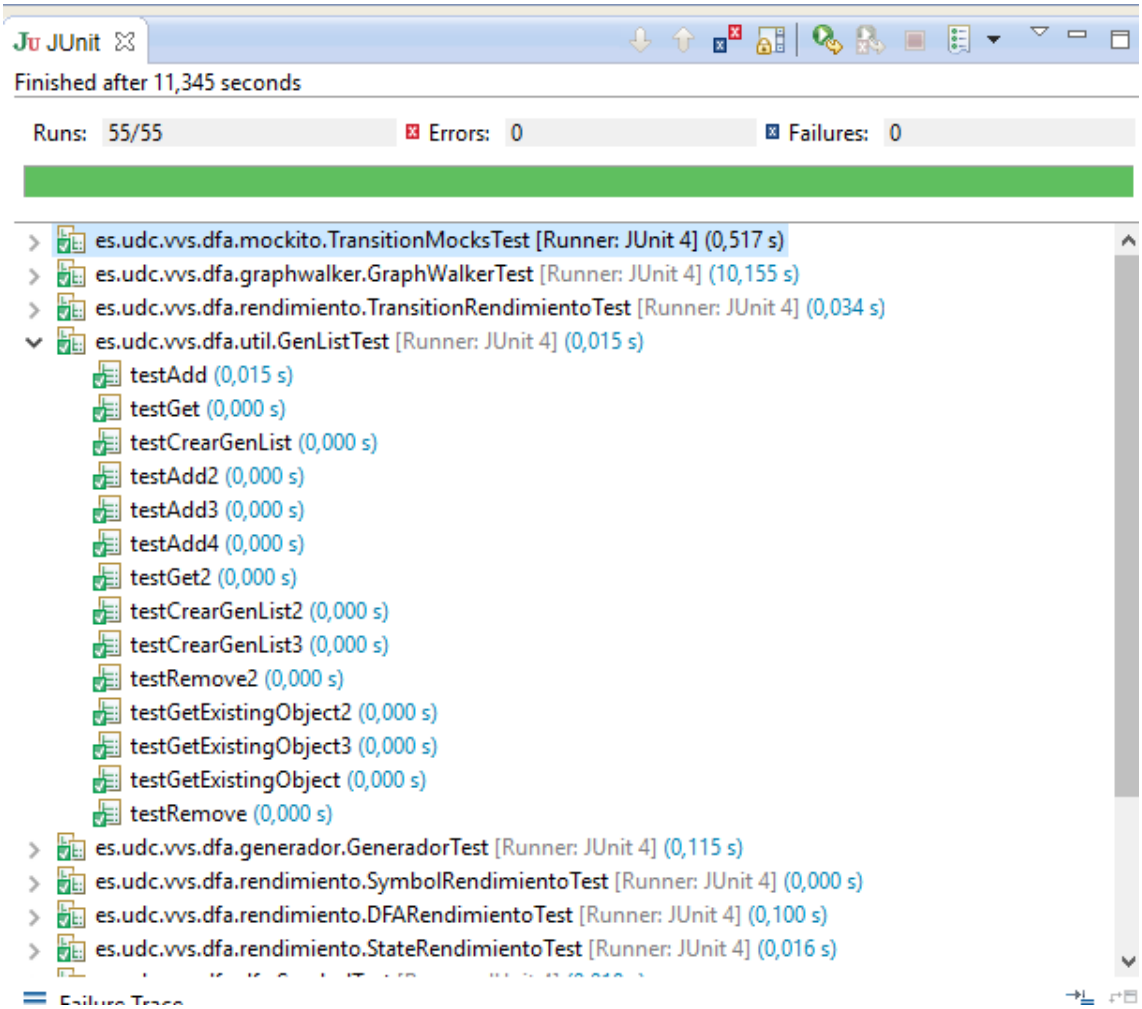
- Comentarios Javadoc: facilitar el mantenimiento pasa por comentar el código, pero luego los comentarios también hay que mantenerlos... CheckStyle tiene muchas reglas para los javadoc y es muy flexible. Te permite, por ejemplo, obligar a comentar los nombres de clases, todos los métodos menos los get/set y los atributos públicos.
- Convenciones de nombres: puedes definir una expresión regular para el nombre de todo.
- Cabeceras: expresiones regulares para la cabecera de los ficheros.
- Imports: reglas para los import, como no usar *, imports sin usar, etc.
- Violaciones de tamaño: define un máximo para el tamaño de tus clases, métodos, líneas y número de parámetros de un método. Espacios en blanco: un montón de reglas para definir donde se ponen espacios en blanco y tabuladores en el código.
- Modificadores: establece un orden para los modificadores y evita modificadores innecesarios.
- Bloques: reglas para los bloques de código y sus llaves.
- Problemas en la codificación: Acá hay de todo, desde malas prácticas tipo asignaciones internas y posibles fuentes de bugs como definir un método equals que no es el equals(Object), a cosas más estéticas o poco prolijas, como que el default sea el último elemento en un switch o paréntesis innecesarios.
- Diseño de clases: varias reglas sobre el diseño de interfaces y clases, con especial atención en las excepciones.
- Duplicados: te permite definir un mínimo de líneas para buscar código duplicado en tus clases.
- Métricas: define máximos para métricas como complejidad ciclomática, complejidad de expresiones lógicas, npath, líneas de código seguidas sin comentar y dependencia de clases.
- Misceláneo: variables final, indentación, un buscador de expresiones regulares y varias cosas más.
- J2EE: reglas para EJBs.
- Otros: internos a CheckStyle y activados por defecto.
- Filtros: para eventos de auditoria del propio CheckStyle, no hace falta mirarlos.

C: Por parte de C, CppCheck realiza otras comprobaciones como vigilar la petición y liberación de memoria dinámica, advirtiendo de fugas potenciales de memoria así como cualquier uso peligroso de un puntero.

5. Registro de errores

5.1. Pruebas unidad: JUnit y Unity

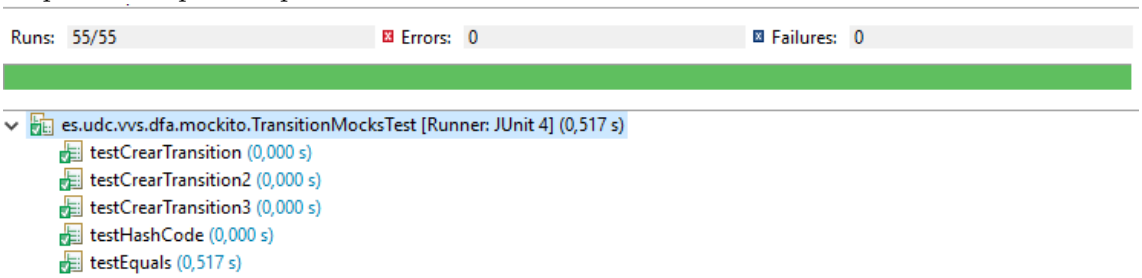
Se puede comprobar que no fallan.



C: La mayoría de issues encontradas se deben a un uso incorrecto de punteros. La mayoría de ellos ni siquiera llegaba a fallar el error, simplemente los tests paraban con un segmentation fault, lo cual llevó a la búsqueda manual del problema.

5.2. Pruebas dinámicas de unidad: Mockito y CMock

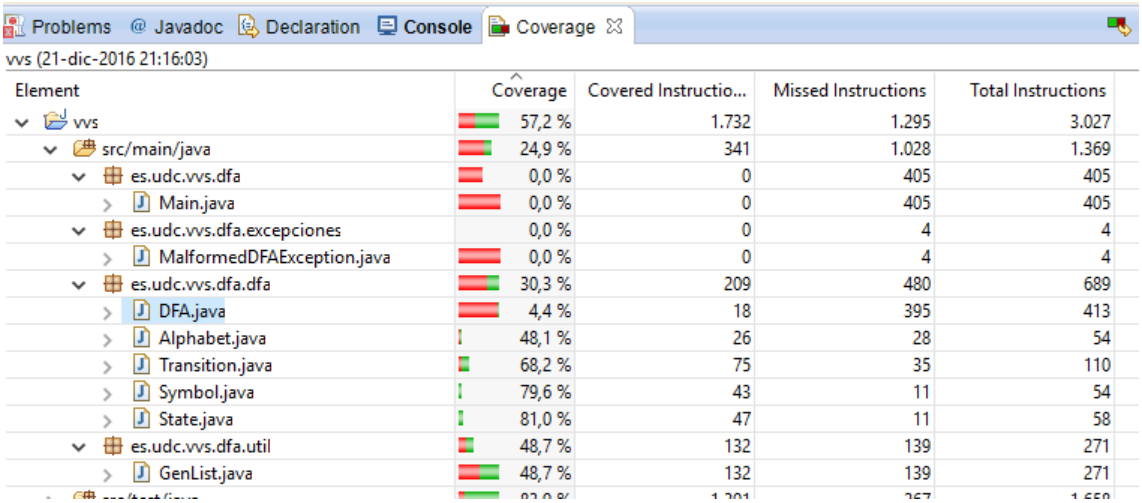
Se puede comprobar que no fallan.



C: No fue posible emplear esta herramienta.

5.3. Validación de la calidad de las pruebas: Cobertura y lcov

Estado inicial:

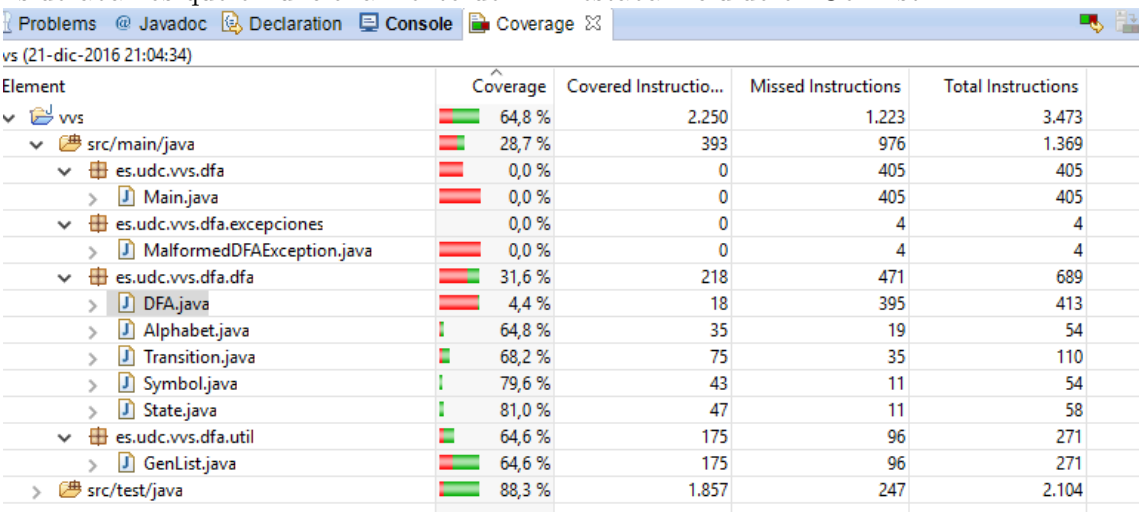

Problems Javadoc Declaration Console Coverage

vvs (21-dic-2016 21:16:03)

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
vvs	57,2 %	1.732	1.295	3.027
src/main/java	24,9 %	341	1.028	1.369
es.udc.vvs.dfa	0,0 %	0	405	405
Main.java	0,0 %	0	405	405
es.udc.vvs.dfa.excepciones	0,0 %	0	4	4
MalformedDFAException.java	0,0 %	0	4	4
es.udc.vvs.dfa.dfa	30,3 %	209	480	689
DFA.java	4,4 %	18	395	413
Alphabet.java	48,1 %	26	28	54
Transition.java	68,2 %	75	35	110
Symbol.java	79,6 %	43	11	54
State.java	81,0 %	47	11	58
es.udc.vvs.dfa.util	48,7 %	132	139	271
GenList.java	48,7 %	132	139	271

Estado final:

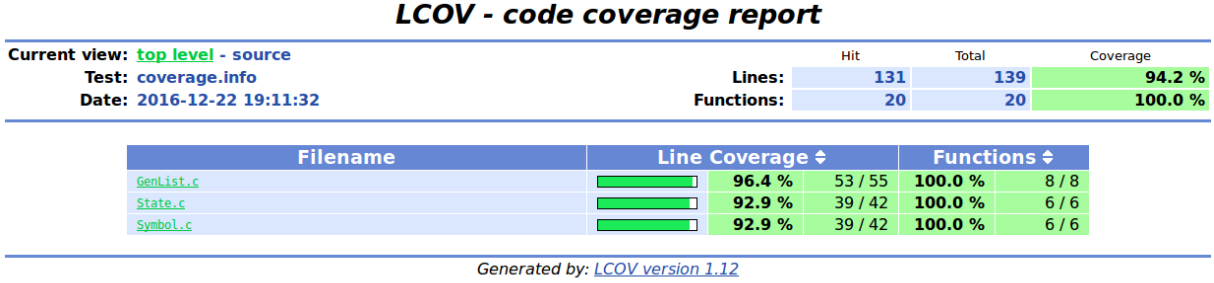
Hemos realizado aumentado las pruebas en GenList para mejorar la cobertura de las pruebas, por definición habíamos decidido no realizar pruebas sobre DFA porque considerábamos que el funcionamiento de DFA estaba incluido en GenList.


Problems Javadoc Declaration Console Coverage

vs (21-dic-2016 21:04:34)

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
vvs	64,8 %	2.250	1.223	3.473
src/main/java	28,7 %	393	976	1.369
es.udc.vvs.dfa	0,0 %	0	405	405
Main.java	0,0 %	0	405	405
es.udc.vvs.dfa.excepciones	0,0 %	0	4	4
MalformedDFAException.java	0,0 %	0	4	4
es.udc.vvs.dfa.dfa	31,6 %	218	471	689
DFA.java	4,4 %	18	395	413
Alphabet.java	64,8 %	35	19	54
Transition.java	68,2 %	75	35	110
Symbol.java	79,6 %	43	11	54
State.java	81,0 %	47	11	58
es.udc.vvs.dfa.util	64,6 %	175	96	271
GenList.java	64,6 %	175	96	271
src/test/java	88,3 %	1.857	247	2.104

C: Se puede generar el informe con "make doc". En la carpeta doc, se encontrará una carpeta llamada coverage la cual contiene el informe de cobertura de las pruebas:


LCOV - code coverage report

Current view: top level - source
Test: coverage.info
Date: 2016-12-22 19:11:32

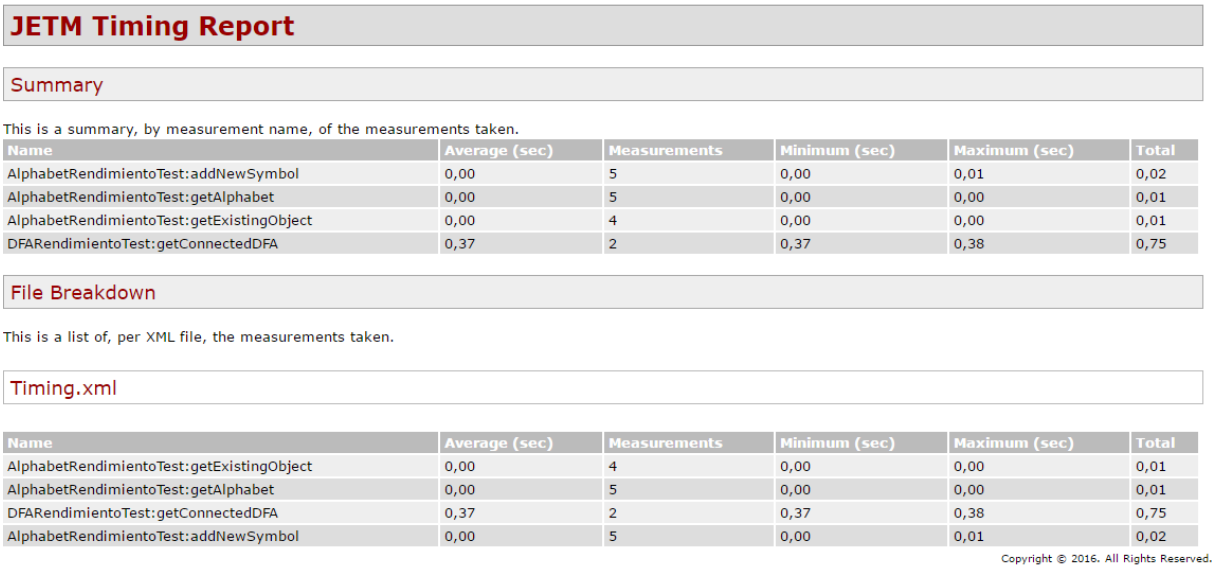
	Hit	Total	Coverage
Lines:	131	139	94.2 %
Functions:	20	20	100.0 %

Filename	Line Coverage		Functions	
GenList.c	96.4 %	53 / 55	100.0 %	8 / 8
State.c	92.9 %	39 / 42	100.0 %	6 / 6
Symbol.c	92.9 %	39 / 42	100.0 %	6 / 6

Generated by: LCOV version 1.12

5.4. Pruebas no funcionales: JETM

Resultado del rendimiento de JETM:



JETM Timing Report

Summary

This is a summary, by measurement name, of the measurements taken.

Name	Average (sec)	Measurements	Minimum (sec)	Maximum (sec)	Total
AlphabetRendimientoTest:addNewSymbol	0,00	5	0,00	0,01	0,02
AlphabetRendimientoTest:getAlphabet	0,00	5	0,00	0,00	0,01
AlphabetRendimientoTest:getExistingObject	0,00	4	0,00	0,00	0,01
DFARendimientoTest:getConnectedDFA	0,37	2	0,37	0,38	0,75

File Breakdown

This is a list of, per XML file, the measurements taken.

Timing.xml

Name	Average (sec)	Measurements	Minimum (sec)	Maximum (sec)	Total
AlphabetRendimientoTest:getExistingObject	0,00	4	0,00	0,00	0,01
AlphabetRendimientoTest:getAlphabet	0,00	5	0,00	0,00	0,01
DFARendimientoTest:getConnectedDFA	0,37	2	0,37	0,38	0,75
AlphabetRendimientoTest:addNewSymbol	0,00	5	0,00	0,01	0,02

Con JETM hemos tenido problemas a la hora de que nos genera el site todos los resultados de las pruebas. Sí genera los resultados guardandolos en un xml con todos los resultados, pero no los muestra en la página de resultados del site.

5.5. Validación de calidad de las pruebas (mutation testing): PIT y Milu

Realizado el mutation testing, resultados:
Informe PIT

C: No se logró usar Milu para hacer mutation testing.

5.6. Pruebas estructurales: CheckStyle

Al pasar la herramienta de CheckStyle descubrimos que nuestra aplicación tiene muchos errores de estilo, es decir, que no cumple el estándar de programación java.

Resumen de errores encontrados:

- Faltan comentarios: En la mayoría de clases faltan comentarios javadoc. Se añaden.
- Mala indexación código y espacios: Se reestructura el código para que solventar dichos errores.

Files	Info	Warnings	Errors
9	0	0	46

Files

File	I	W	E
es/udc/vvs/dfa/Main.java	0	0	2
es/udc/vvs/dfa/dfa/State.java	0	0	7
es/udc/vvs/dfa/dfa/Symbol.java	0	0	7
es/udc/vvs/dfa/dfa/Transition.java	0	0	16
es/udc/vvs/dfa/excepciones/MalformedDFAException.java	0	0	1
es/udc/vvs/dfa/util/GenList.java	0	0	7
testautomation/VVS.java	0	0	6

Category	Rule	Violations	Severity
blocks	EmptyBlock	1	Error
	NeedBraces	6	Error
	RightCurly	5	Error
design	HideUtilityClassConstructor	1	Error
javadoc	JavadocMethod	2	Error
	JavadocType	1	Error
misc	FinalParameters	14	Error
	NewlineAtEndOfFile	1	Error
naming	MethodName	2	Error
whitespace	WhitespaceAfter	3	Error
	WhitespaceAround	10	Error

es/udc/vvs/dfa/Main.java

Severity	Category	Rule	Message	Line
Error	design	HideUtilityClassConstructor	Utility classes should not have a public or default constructor.	19
Error	misc	FinalParameters	Parameter filePath should be final.	28
Error	misc	FinalParameters	Parameter dfa should be final.	47
Error	misc	FinalParameters	Parameter args should be final.	113

es/udc/vvs/dfa/dfa/Alphabet.java

Severity	Category	Rule	Message	Line
Error	misc	FinalParameters	Parameter n should be final.	26
Error	misc	FinalParameters	Parameter symbol should be final.	34
Error	misc	FinalParameters	Parameter symbol should be final.	45

es/udc/vvs/dfa/dfa/DFA.java

Severity	Category	Rule	Message	Line
Error	misc	FinalParameters	Parameter states should be final.	44
Error	misc	FinalParameters	Parameter alphabet should be final.	44
Error	misc	FinalParameters	Parameter initialState should be final.	44
Error	misc	FinalParameters	Parameter finalStates should be final.	45
Error	misc	FinalParameters	Parameter transitions should be final.	45
Error	coding	InnerAssignment	Inner assignments should be avoided.	77

es/udc/vvs/dfa/dfa/State.java

Severity	Category	Rule	Message	Line
Error	misc	FinalParameters	Parameter state should be final.	20
Error	misc	FinalParameters	Parameter obj should be final.	45

es/udc/vvs/dfa/dfa/Symbol.java

Severity	Category	Rule	Message	Line
Error	misc	FinalParameters	Parameter symbol should be final.	20
Error	misc	FinalParameters	Parameter obj should be final.	40

es/udc/vvs/dfa/dfa/Transition.java

Severity	Category	Rule	Message	Line
Error	misc	FinalParameters	Parameter startState should be final.	30
Error	misc	FinalParameters	Parameter endState should be final.	30
Error	misc	FinalParameters	Parameter symbol should be final.	30
Error	misc	FinalParameters	Parameter obj should be final.	67

es/udc/vvs/dfa/excepciones/MalformedDFAException.java

Severity	Category	Rule	Message	Line
Error	misc	FinalParameters	Parameter message should be final.	15

es/udc/vvs/dfa/util/GenList.java

Severity	Category	Rule	Message	Line
Error	misc	FinalParameters	Parameter buffer should be final.	38
Error	blocks	NeedBraces	'if' construct must use '{}'s.	39
Error	misc	FinalParameters	Parameter obj should be final.	66
Error	misc	FinalParameters	Parameter n should be final.	87
Error	misc	FinalParameters	Parameter n should be final.	119
Error	misc	FinalParameters	Parameter obj should be final.	148
Error	blocks	EmptyBlock	Must have at least one statement.	163

Después de revisar los errores encontrados, resolvemos los errores de estilo encontrados y este es el resultado: Informe CheckStyle

5.7. Pruebas estáticas/estructurales: FindBugs y CppCheck

FindBugs es un programa que utiliza el análisis estático para buscar errores en el código de Java.

Informe Find bugs

Errores encontrados inicialmente:

Summary

Classes	Bugs	Errors	Missing Classes
8	4	0	0

Files

Class	Bugs
es.udc.vvs.dfa.Main	1
es.udc.vvs.dfa.dfa.State	1
es.udc.vvs.dfa.dfa.Symbol	1
es.udc.vvs.dfa.util.GenList	1

es.udc.vvs.dfa.Main

Bug	Category	Details	Line	Priority
Found reliance on default encoding in es.udc.vvs.dfa.Main.getStringAutomataFromFile(String): new java.io.FileReader(String)	I18N	DM_DEFAULT_ENCODING	33	High

es.udc.vvs.dfa.dfa.State

Bug	Category	Details	Line	Priority
es.udc.vvs.dfa.dfa.State.toString() invokes toString() method on a String	PERFORMANCE	DM_STRING_TOSTRING	34	Low

es.udc.vvs.dfa.dfa.Symbol

Bug	Category	Details	Line	Priority
es.udc.vvs.dfa.dfa.Symbol.toString() invokes toString() method on a String	PERFORMANCE	DM_STRING_TOSTRING	58	Low

es.udc.vvs.dfa.util.GenList

Bug	Category	Details	Line	Priority
es.udc.vvs.dfa.util.GenList.getArray() may expose internal representation by returning GenList.list	MALICIOUS_CODE	EI_EXPOSE_REP	130	Medium

Después de identificar los errores, resolvemos los problemas mencionados y comprobamos el resultado:

Summary

Classes	Bugs	Errors	Missing Classes
9	1	0	0

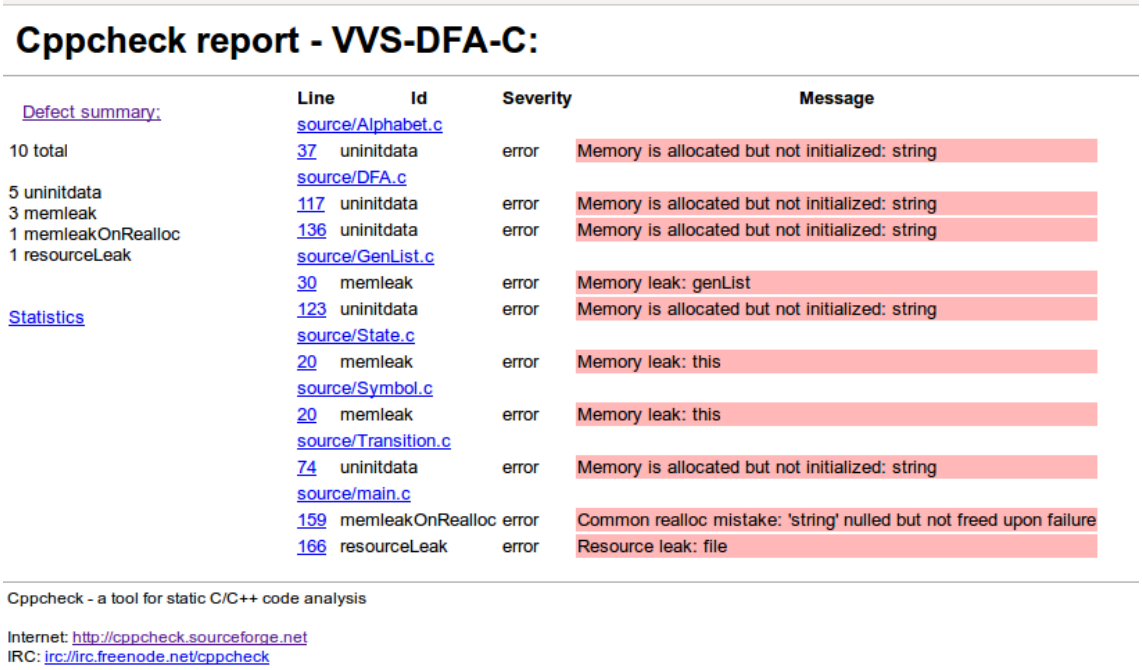
Files

Class	Bugs
es.udc.vvs.dfa.util.GenList	1

es.udc.vvs.dfa.util.GenList

Bug	Category	Details	Line	Priority
es.udc.vvs.dfa.util.GenList.getArray() may expose internal representation by returning GenList.list	MALICIOUS_CODE	EI_EXPOSE_REP	130	Medium

C: El informe inicial de CppCheck tenía este aspecto:



Cppcheck report - VVS-DFA-C:

Defect summary:

10 total

5 uninitdata
3 memleak
1 memleakOnRealloc
1 resourceLeak

Statistics

Line	Id	Severity	Message
source/Alphabet.c			
37	uninitdata	error	Memory is allocated but not initialized: string
source/DFA.c			
117	uninitdata	error	Memory is allocated but not initialized: string
136	uninitdata	error	Memory is allocated but not initialized: string
source/GenList.c			
30	memleak	error	Memory leak: genList
123	uninitdata	error	Memory is allocated but not initialized: string
source/State.c			
20	memleak	error	Memory leak: this
source/Symbol.c			
20	memleak	error	Memory leak: this
source/Transition.c			
74	uninitdata	error	Memory is allocated but not initialized: string
source/main.c			
159	memleakOnRealloc	error	Common realloc mistake: 'string' nulled but not freed upon failure
166	resourceLeak	error	Resource leak: file

Cppcheck - a tool for static C/C++ code analysis

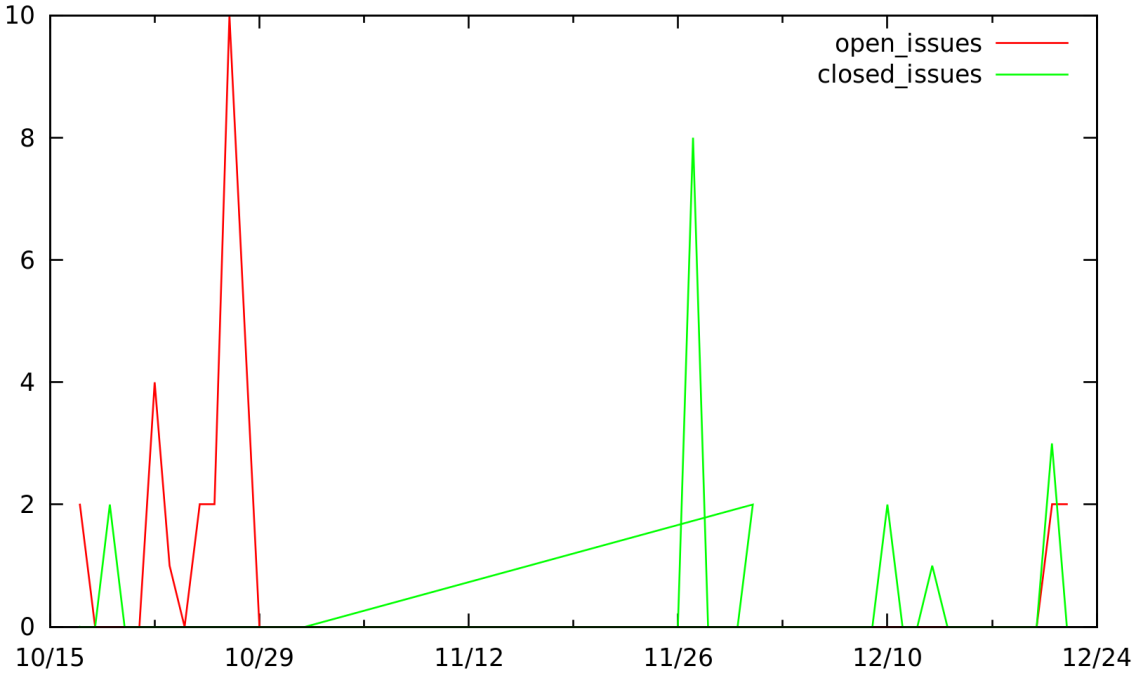
Internet: http://cppcheck.sourceforge.net
IRC: irc://irc.freenode.net/cppcheck

El informe actual se puede generar usando "make doc". Sólo queda el problema de *resourceLeak*, el cual está referenciado en la issue 6 en el repositorio de GitHub.

6. Estatísticas

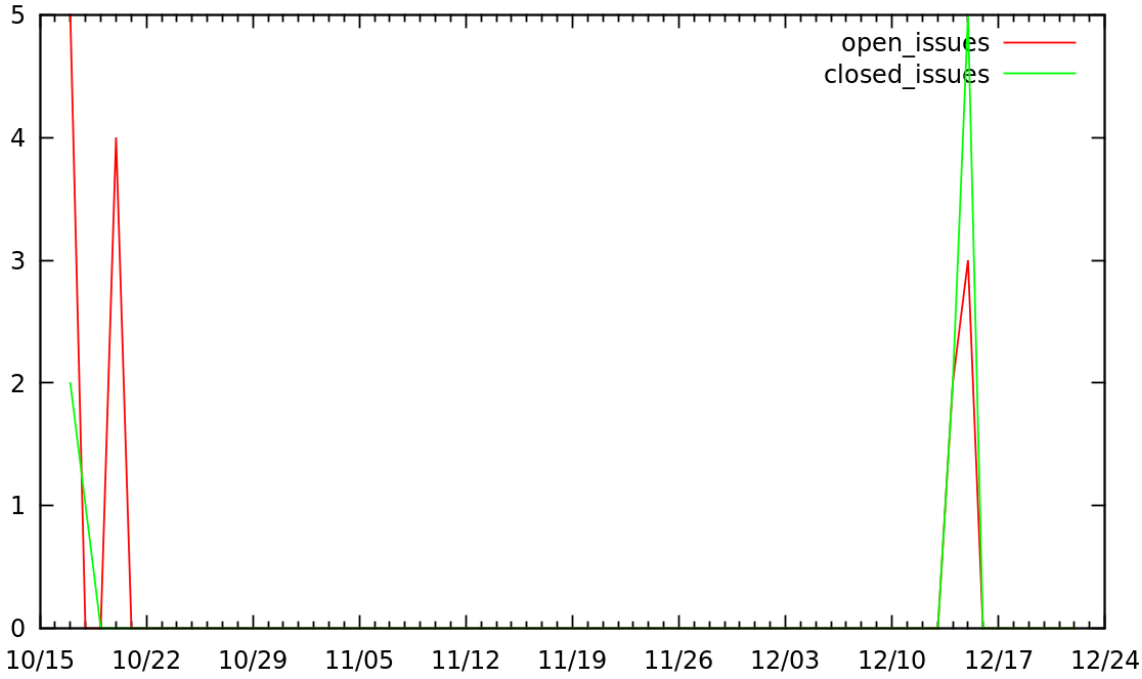
6.1. Número de errores encontrados diariamente y semanalmente.

Java: A partir de las pruebas de unidad con la herramienta junit, el mayor número de errores encontrados son después de realizar las pruebas de análisis de caja blanca (checkstyle y findbugs) y mutación de código (PITest). La tendencia después del punto álgido del 29/10/2016 (observar gráfica) a ir cerrando poco a poco las incidencias abiertas.



Sin embargo, se puede observar que en las últimas semanas se volvieron a abrir issues debido a que se detectó que las pruebas realizadas no eran válidas según la cobertura. Por lo que se abren dos issues de herramientas y test pendientes de implementar para dar por completo por finalizado de testar la aplicación.

C: Por parte de este lenguaje, el ritmo de localización de pruebas empezó fuerte con la aplicación de las herramientas de CUnit, QuickCheck, y CppCheck, los cuales detectaron una cantidad sustancial de errores. Sin embargo, cometimos la equivocación de no empezar a cerrar issues hasta tener todos los tests implementados junto con las herramientas, lo cual considerando los problemas que hubo no fue la mejor de las ideas. Las issues se cerraron en Diciembre sin demasiadas complicaciones, el mes de distancia no es relevante con respecto a la dificultad de cerrarlas.



6.2. Nivel de progreso de ejecución de las pruebas

Java: Las pruebas diseñadas a priori fueron las pruebas iniciales de unidad, mockito y quickcheck. A posteriori se fueron añadiendo las pruebas obtenidas de ejecutar las herramientas de análisis estático de caja blanca (CheckStyle), Herramienta de mutación de código (PITest).

Podemos comprobar que con cobertura qué está al 68 por ciento y hemos diseñado pruebas para la clase DFA para mejorar la cobertura de nuestras pruebas. En una decisión inicial habíamos pensado en que estaban contenidas dentro de GenList pero por el resultado se comprueba qué es necesario realizar pruebas de las mismas. Faltarían por ejecutar.

C: Las pruebas diseñadas se pudieron implementar en su totalidad para los módulos que no necesitaron de un mock, siendo State, Symbol y GenList, los cuales tienen el 100 % de sus pruebas implementadas. El resto de módulos, sin embargo, no han implementado ninguna prueba debido a la imposibilidad de hacer funcionar los mocks en C.

La cobertura fue satisfactoria para estos primeros módulos, alcanzando el 99.75 % del código cubierto. Se incluyeron pruebas para QuickCheck para comprobar que State y Symbol podían ser creados con cualquier cadena de caracteres sin problemas, se definieron funciones para liberación de memoria necesarios para poder hacer una cantidad significativa de tests repetidos con QuickCheck, y se incluyeron varias comprobaciones extra detectadas necesarias por las pruebas que diseñamos.

6.3. Análisis del perfil de detección de errores

Java: La mayoría de errores encontraros se encontraron sobre la clase GenList. Problemas con la lista de objetos y el buffer de objetos.

C: Los errores encontrados se concentraron principalmente en State y Symbol, dada la similitud de sus módulos y debido a su uso extensivo de cadenas de texto y reserva dinámica de memoria.

6.4. Evaluación global del estado y calidad y estabilidad actual

Actualmente se puede comprobar que el código es estable y las pruebas realizadas son correctas, sin embargo, eso no quiere decir de que no existan errores.

Java: En la definición de las pruebas decidimos no realizar pruebas sobre DFA porque considerábamos que estaban incluidas en GenList, sin embargo al validar con cobertura y con PIT(mutación testing) comprobamos que afecta negativamente a la validación de las pruebas. Se debería realizar más pruebas para una cobertura correcta del programa. Lo ideal sería obtener entre un 80 por ciento y 100 por cien de cobertura de las pruebas.

Decidimos no emplear las herramientas de GraphWalker, porque la ejecución de la aplicación es totalmente lineal y no nos generaría una máquina de estados útil para nuestras pruebas.

Una cobertura del 68 por ciento no nos garantiza que existan bugs en el programa. Se podría considerar que consiguiendo realizar las pruebas de estrés y basadas en modelos y aumentando un poco la cobertura de las pruebas podríamos considerar que el programa está completamente validado y listo para utilizarse en algún entorno de producción.

C: Tuvimos problemas imprevistos con dos herramientas: Cmock y Milu. Al intentar utilizarlas, surgieron errores que no nos vimos capaces de solventar en tiempo, lo cual nos impidió su uso. Fue especialmente problemático lo de CMock, pues sin la posibilidad de hacer mocks nos quedó sin testear la mitad de los módulos.

Por otro lado, las pruebas realizadas con el resto de herramientas dieron resultados satisfactorios, alcanzando el 99.75 % de cobertura en los módulos State y Symbol, y el 100 % con GenList, empleando de manera conjunta Unity y QuickCheck4c.

Por parte de CppCheck, nos ayudó enormemente a localizar fugas de memoria por nuestro programa. Si bien en una ejecución normal no surgiría ningún problema, fue lo que nos permitió aumentar sustancialmente el número de pruebas a realizar con QuickCheck4c, pues no encontrábamos fallos de segmento debidos a la memoria huérfana que dejaban las pruebas.

7. Otros aspectos de interés

El equipo sufrió una baja (Javier López Moure) a mitad del proceso de pruebas, lo cual afectó a nuestra capacidad para realizar todas las pruebas sobre la aplicación.