Detección asutomática de métodos suficientes y no superfluos para la construcción de drivers eficientes en bounded model checking

Mariano Politano[1,3], Valeria S. Bengolea[1], Pablo Ponzio[1,3], and Nazareno Aguirre[1,3]

[1] Universidad Nacional de Río Cuarto, Río Cuarto, Argentina.
{pponzio,vbengolea,mpolitano,naguirre}@dc.exa.unrc.edu.ar
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

Abstract. Todos sabemos que los software contienen bugs. Dado que Java es el lenguaje de programación mas popular, es esencial tener herramientas que puedan detectar errores en códigos Java. Testing es la técnica mas usada para detectar bugs, pero esta tiene limitaciones, especialmente cuando existe código no-determinista. Para encontrar errores en estos códigos, testing necesita ser complementado con otras técnicas como *modelchecking*. Java PathFinder (JPF) [9] es uno de los mas populares model checker para códigos Java. Esta herramienta, se basa en la definición de controladores (*drivers*), que son métodos encargados de guiar la ejecución del model checker. De esta forma, se logra construir todas las listas, siempre de manera acotado debido a la explosión del espacio de búsqueda.
En un trabajo previo, nosotros logramos construir estos drivers de manera automática [8]. Esto quiere decir, conseguir el subconjunto mínimo de métodos de una API, que me permitan generar todos los objetos. (por ejemplo, para la API LinkedList, con el metodo add, es suficiente). A partir de esto, luego se puede realizar model checking. En este trabajo se propone realizar un análisis experimental para poder observar cuanto ayuda nuestra herramienta previa para para que combinada con JPF pueda realizar un análisis de un modulo de manera mas eficiente.
Para entender mejor el problema, nosotros elegimos la forma de generación aleatoria, para demostrar que podemos elegir el mejor individuo para el objetivo dado. Gracias a a este avance, podemos establecer una relación para mejorar la técnica y as poder mejorar nuestro imput.

1 Introducción

El uso de software está en continuo crecimiento gracias al avance de la tecnología, lo que significa que garantizar el correcto funcionamiento del software es más crucial que nunca. Por lo tanto, un área de investigación de creciente importancia es la de análisis automático de software, cuyo objetivo es ayudar a los desarrolladores de software a detectar fallas en el software mediante herramientas que funcionan de manera autónoma, es decir, sin asistencia por parte del desarrollador. La generación automática de tests [5, 1], la verificación de software [4, 11],

y los análisis estáticos [3], entre muchos otros, son enfoques destacados en esta línea de investigación.

Java PathFinder (JPF) es una herramienta de verificación automática de programas, que implementa diversas técnicas de análisis [11]. Uno de las posibles aplicaciones de JPF, en la que estamos interesados en este artículo, es realizar *bounded model checking*. Esto es explorar sistemáticamente el conjunto de todas las ejecuciones del software con entradas de tamaño acotado en busca de fallas. Los límites en los tamaños de las entradas deben ser proporcionados por el usuario de la herramienta (ej. cantidad máxima de nodos en una lista, rango de enteros a considerar, etc.). En caso de encontrar una falla, JPF la reporta al usuario para facilitar el debugging.

Para llevar a cabo la verificación acotada, JPF se basa en la definición de *drivers*: combinaciones de métodos que permiten construir todas las entradas acotadas para ejecutar el código bajo análisis (que usualmente requieren del uso de construcciones no deterministas del *model checker*, ver sección **??**). Para tipos de datos complejos (ej. estructuras dinámicas alocadas en el *heap* como listas doblemente encadenadas) usualmente la creación de estructuras requiere de la utilización de varias de las rutinas de la API del módulo que implementa la estructura (ej. add, remove, etc.), y por lo tanto, la construcción del *driver* depende de estas rutinas.

Sin embargo, para módulos ricos con respecto a la cantidad de métodos, la selección de métodos para construir el *driver* es una tarea no trivial, ya que como se detalla en la sección de resultados experimentales **??**, esto puede tener un impacto importante en el análisis posterior empleando el *driver*. La intuición manifiesta que es deseable seleccionar un conjunto de métodos tan pequeño como sea posible cuyas combinaciones permitan construir todas las estructuras acotadas para el módulo. Por un lado, es importante no incluir rutinas superfluas con respecto a la construcción de objetos, ya que la cantidad de combinaciones posibles de un conjunto de métodos crece exponencialmente con el número de métodos en el conjunto (por lo tanto, utilizar un subconjunto más pequeño de métodos implica una mejora de la eficiencia del análisis). El ejemplo más simple de rutinas superfluas son los métodos puros [], que nunca modifican las estructuras sobre las que operan. Por otro lado, cuanto mayor es el numero de estructuras diferentes que se pueda crear con el subconjunto de rutinas seleccionado, mayor será la probabilidad de encontrar errores en el código. Es decir, es importante seleccionar subconjuntos de métodos suficientes, que pueden combinarse para construir todas las instancias acotadas posibles. Esta selección de métodos, que usualmente se lleva a cabo de forma manual, no es una tarea fácil: requiere un análisis exhaustivo de las rutinas disponibles en el módulo y una comprensión profunda de la semántica de las mismas. Esta tarea es muy tediosa sobre todo para módulos con API ricas, donde hay muchas rutinas y mucha redundancia entre ellas. En un trabajo previo [8], se propuso un enfoque automático para seleccionar un subconjunto de rutinas suficientes y no superfluas de la API de un módulo, de manera automática.

No.	Return type	Method name	No.	Return type	Method name
0		NCL()	17	boolean	isEmpty()
1		NCL(int)	18	Iterator	iterator()
2		NCL(Collection)	19	int	lastIndexOf(Object)
3	boolean	add(Object)	20	ListIterator	listIterator()
4	void	add(int,Object)	21	ListIterator	listIterator(int)
5	boolean	addAll(Collection)	22	Object	remove(int)
6	boolean	addAll(int,Collection)	23	boolean	remove(Object)
7	boolean	addFirst(Object)	24	boolean	removeAll(Collection)
8	boolean	addLast(Object)	25	Object	removeFirst()
9	void	clear()	26	Object	removeLast()
10	boolean	contains(Object)	27	boolean	retainAll(Collection)
11	boolean	containsAll(Collection)	28	Object	set(int,Object)
12	boolean	equals(Object)	29	int	size()
13	Object	get(int)	30	List	subList(int,int)
14	Object	getFirst()	31	Object[]	toArray()
15	Object	getLast()	32	Object[]	toArray(Object[])
16	int	indexOf(Object)	33	String	toString()

Table 1. API de Apache NodeCachingLinkedList

En este trabajo, se observa que el enfoque mencionado puede ser de utilidad para la construcción de *drivers* eficientes para *bounded model checking*, ya que permite reducir la cantidad de métodos a utilizar en los *drivers* (evitando métodos superfluos), mejorando su eficiencia, y manteniendo la capacidad de construir todos los objetos acotados posibles (por la suficiencia de los métodos elegidos). Todo esto, sin requerir trabajo adicional al usuario de la herramienta. Así, se propone una técnica de construcción de *drivers* que utilice los métodos suficientes y no superfluos seleccionados por la herramienta, y se lo compara experimentalmente con el enfoque de utilizar todos los métodos de la API (que representa el caso en que no se tiene información adicional de cuáles métodos son importantes para la construcción del *driver*). Se evaluaron ambos drivers en un caso de estudio (Apache NodeCachingLinkedList) para el análisis de una propiedad particular de este módulo. Los resultados muestran que (COMPLETAR: ganamos en tiempo y ganamos en tamaño de las estructuras).

2 Ejemplo motivador

En esta sección describiremos un ejemplo que nos ayudara a explicar el trabajo realizado. Nos centramos en la estructura de datos Apache NodeCachingLinkedList (NLC) [12]. Es una implementación de listas que intenta reducir la creación de objetos y el posterior uso del *garbage collector*, manteniendo una lista de los nodos borrados en una *cache*. Es una lista doblemente encadenada y circular, esto quiere decir que cada nodo tiene una referencia al nodo siguiente y al anterior. Ademas cuenta con una *cache*, que es una lista simplemente encadenada que se forma con los nodos que tienen que ser removidos de la lista principal. Estos nodos pueden ser reutilizados insertándolos nuevamente en la lista principal.

NCL tiene una gran cantidad de métodos en su API, pero solo unos pocos son suficientes para generar cualquier instancia finita de NCL. En la tabla 1 se encuentran todos los métodos que existen en dicha API. En primer lugar, siempre, se necesita un método constructor de objetos y luego se puede observar que

```
(3) add(Object)
(4) add(int,Object)
(7) addFirst(Object)
(8) addLast(Object)
```

Figure 1.1. Variantes del metodo Add

```
(0)  NodeCachingLinkedList()
(7)  addFirst(Object)
(25) removeFirst()
```

Figure 1.2. driverNLC

al menos una de las variantes del método **add** es suficiente para agregar elementos a la lista *principal*. Sin embargo, si queremos generar instancias donde la lista *cache* es no vacía, debemos utilizar alguna variante del método **remove**. El método **remove** elimina nodos de la lista *principal* y los inserta en la lista *cache*. Estos métodos (**remove y add**) son suficiente dado que para crear cualquier estructura como sea posible, con estos métodos es suficiente. También, son minimales dado que cualquier de estos 3 métodos que no esté implicaría que haya alguna instancia que no se va a lograr construir. Decimos que una rutina es un observador si nunca modifica los parámetros que toma, y nunca genera un valor no primitivo como resultado de su ejecución. En la API de NLC hay varios métodos que son observadores.Por lo tanto, los observadores son siempre superfluos y nunca deben incluirse en un conjunto de constructores mínimos. Nuestro enfoque trata de reconocerlos de antemano y los descarta de la búsqueda para reducir significativamente el espacio de búsqueda.

Nosotros observamos que cuanto más simples son los parámetros de una rutina, más fácil es usar la rutina para generar entradas en el contexto de un análisis de programa. Ademas, existen varios subconjuntos de métodos equivalentes para lograr obtener los generadores de objetos. Por ejemplo, se puede lograr utilizando diferentes versiones de **add** o **remove**. Nuestro enfoque le da prioridad a aquellos métodos que contienen menos cantidad de parámetros y menos complejidad de los mismo. Por ejemplo, en la figura 1.1, **add(int,Object)** recibe mas parámetros que los otros tres, por lo cual, nuestra herramienta lo reemplaza durante la ejecución por alguno de los otros métodos **add**. Una vez que alimentamos con toda la API de NCL a nuestra herramienta, nos brinda el subconjunto de métodos, mínimo y suficiente, que se encuentra en la figura 1.2.

3 Generación automática de objectos

En esta sección describiremos el algoritmo implementado en nuestra herramienta y que utilizaremos para obtener los métodos generadores de objectos a partir de una API.

Para realizar esto, nosotros utilizaremos Algoritmos Genéticos que son algoritmos de búsqueda no exhaustivo basado en la idea de hill climbing [10].

El espacio de búsqueda de este algoritmo esta compuesto de un conjunto muy grande de individuos (posibles soluciones del problema) comúnmente llamados *cromosomas*. Estos cromosomas son representados como vectores donde cada posición se la denomina *gen*. El objetivo del algoritmo es buscar una característica deseada en la población (conjunto de cromosomas). Esta búsqueda siempre es bajo la condición de que el algoritmo mantenga elitismo (guarde siempre al mejor individuo de la población). Para lograr esto se requiere una función de ajuste la cual asigna un valor a cada cromosoma posible. Para mas detalles sobre estos algoritmos, recomendamos la lectura de [6].

En el caso de nuestro problema, los individuos son representados por los métodos de la API a analizar. Estos cromosomas lo representamos como vectores booleanos, que tendrán tamaño n que es el numero de métodos de la API. Por ende, la posición i-th sera verdadera si y solo si el cromosoma contiene el método i-th de la API. De esta manera el cromosoma que representa al subconjunto de métodos suficientes y minimales de esta API (ver figura 1.2), sera un vector que tendrá el valor 1 (valor *true*) en la posición 0, 7 y 25. La función de ajuste de nuestro problema computa la cantidad acotada de objetos que se puede construir usando la combinación de métodos que están presentes en el cromosoma a evaluar. Cuanto mas objectos pueda construir un cromosoma, mas alto sera el numero que retorne nuestra función de ajuste. Para esto necesitamos un generador exhaustivo limitado para el conjunto de métodos. El límite k representa el número máximo de objetos que se pueden crear para cada API y el número máximo de valores primitivos disponibles (por ejemplo, enteros de 0 a k - 1). Para este propósito, desarrollamos un prototipo que modifica la herramienta Randoop. Primero, modificamos Randoop para que funcione con un conjunto fijo de valores primitivos (enteros de 0 a k - 1). Luego, canonizamos los objetos generados por la ejecución de cada secuencia, y descartamos la secuencia si algún objeto tiene un índice igual o mayor que k. Y ademas, extendimos Randoop con extensiones de campo "globales", y cuando la ejecución de una secuencia termina, todos los valores de campo de los objetos generados por la secuencia se agregan a las extensiones de campo. El resultado de la función de ajuste para cada cromosoma es el numero de valores de campo en las extensiones globales computada por nuestra herramienta. Para mas información acerca de la herramienta Randoop, recomendamos leer [7].

Para obtener buenos conjuntos de generadores de objetos, nosotros realizamos dos mejoras a nuestra función de ajuste. Por un lado, cuando hay dos conjuntos suficientes de generadores, siempre se elige el conjunto con menor cantidad de métodos. De esta manera evitamos incorporar métodos superfluos. Por otro lado, aquellos generadores de objectos con mas parámetros, o con parámetros mas complejos tienen un impacto negativo respecto de los generadores con parámetros mas simples. Luego de esta explicación, nuestra función de ajuste queda definida de la siguiente manera:

$$f(M) = \#extensionesDeCampo(M) + \left(\frac{w_1 * \left(1 - \frac{\#M}{\#MT}\right) + w_2 * \left(1 - \frac{(\#PP(M) + w_3 * PR(M))}{(\#PP(MT) + w_3 * PR(MT))}\right)}{w_1 + w_2} \right)$$

Dado un cromosoma, M es el conjunto de métodos disponible en dicho individuo y MT es el conjunto de métodos que existen en la API. *extensionesDeCampo* hace referencia a las extensiones de los valores de campos generados por la herramienta Randoop personalizada para nuestro enfoque. La parte derecha de la sumatoria retorna un valor entre 0 y 1 que permite penalizar a los métodos que generan igual igual numero de extensiones de campo de acuerdo a lo explicado anteriormente. En el dividendo de la formula general, el primer sumando penaliza los conjuntos que contengan un mayor numero de método. Esto lo realiza restándole a 1 el resultado de la división entre el numero de métodos en MT y el numero de métodos en M. La constante *w1* nos permite aumentar/disminuir el peso de este sumando con respecto al otro. En el segundo sumando se penaliza los conjuntos de metodos con parametros mas complejos. Similar a *w1* la constante *w2* sirve para darle mas o menos peso a esta suma. *PP()* es el numero de parámetros primitivos en los métodos que contiene el subconjunto pasado como parámetro de esta función. Se puede observar que se suma la cantidad de parámetros primitivos en los metodos de M. Tambien, cada parametro de tipo referencia agrega una constante *w3*. *RP()* es el numero de parámetros de tipo referencia que hay en los métodos que contiene el subconjunto pasado como parámetro. Intuitivamente, el lado derecho del sumando computa el radio entre el numero de parametros en M (con el peso agregado de los parametros de referncia) con el numbero de paremetros en MT (tambien ponderado). El resultado de todo esto, se resta de 1. Finalmente, nosotros dividimos esto entre *w1* + *w2* para obtener el numero deseado en el intervalo [0,1].

En nuestra evaluación experimental establecimos $w1 = 2, w2 = 1, w3 = 2$. Estos valores fueron lo suficientemente buenos para que nuestro enfoque produjera conjuntos de constructores suficientes y mínimos en todos los casos de estudios de la herramienta.

Para resumir la explicación de la herramienta, dejaremos el pseudocodigo del algoritmo que se utiliza

4 Resultado Experimental

En esta seccion nosotros realizaremos una evaluacion de como se comporta nuestra herramienta para calcular los generadore de objectos en la clase NLC. Luego, realizaremos una comparacion para demostrar el aporte que se puede realizar a JPF luego de tener el conjunto de generadores de objetos.

Todos los experimentos fueron corridos en maquinas con cuatro core Intel i7-6700 3.4GHz con 8GB de RAM, con el sistema opertativo GNU/Linux. La

	Sample Builders	Time
NCL #API: 34	NCLinkedList(int) addFirst(Object) removeFirst()	1744

Table 2. Resultado de la generaracion

evaluacion consiste en brindarle a JPF un driver con todos los metodos de la API de la clase NLC de Apache Collections como muestra la figura 1.2

La primer parte del analisis consiste en evaluar qué tan buenos son los constructores identificados y el tiempo que toma nuestro enfoque para calcularlos. Para cada estudio de caso, ejecutamos nuestro enfoque 5 veces. Los resultados se muestran en la Tabla 2 , incluido el número de rutinas en toda la API, una muestra de constructores identificados (algunos métodos pueden intercambiarse en diferentes ejecuciones, por ejemplo, addFirst y addLast en NCL), y tiempo de ejecución promedio (en segundos) de las 5 carreras. Inspeccionamos manualmente los resultados y descubrimos que los conjuntos de constructores identificados automáticamente eran en todos los casos suficientes (todos los objetos factibles para la estructura pueden construirse usando los constructores) y mínimos (no contienen métodos superfluos).

La segunda parte de la evaluación se refiere a cuán útiles son los generadores identificados en el contexto de un análisis de programa, mas especificamente en la utilizacion de la herramienta Java PathFinder. Estos generadores pueden usarse, por ejemplo, para la creacion de driver para guiar la ejecucion simbolica de dicho analisis. Como primera etapa de esta parte, mostraremos que es mas tedioso escribir el driver a partir de todas las rutinas de la API. Luego de ejecutar nuestra herramienta, podemos escribir el driver solo con los metodos obtenidos. Esto nos mejorara la forma de escribir el driver y seria mas entendible al usuario. Para apliar sobre esto, mostraremos el pseudocodigo de un driver para verificacion de la estructura con todos las rutinas de la API, **??**La otra figura ... es el driver solo con las rutinas generadores que nuestra herramienta nos brindó.

En segunda etapa se realizo fue una comparacion por igualdad (tiene en cuenta el scope que necesita esta verificacion) para determinar cuanto tiempo toma el driver para realizar la verificacion del software utilizando todos los métodos disponibles en la API, y luego calculamos el tiempo que le lleva a un driver utilizando solo los métodos de generacion (BLD) identificados por nuestro enfoque en el experimento anterior.

Figure **??** shows two algorithms side by side: they can be referenced as algorithm **??** and algorithm **??**...

5 Trabajo Relacionado

Como se menciona a lo largo del documento, el problema de identificar suficientes constructores es recurrente en varios análisis de programas, que incluyen, entre otros, la verificación de modelos de software y la generación de pruebas. En el

contexto de la verificación del modelo de software, en el contexto de la generación automatizada de pruebas, y solo por citar algunos, el problema de identificar parte de un API y proporcionarlo para el análisis está presente. Por lo general, el problema se trata manualmente.

El uso de técnicas basadas en la búsqueda para resolver problemas desafiantes de ingeniería de software es una estrategia cada vez más popular, que se ha aplicado con éxito a una serie de problemas, incluida la generación de entrada de prueba, la reparación del programa y muchos otros. Hasta donde sabemos, esta es una aplicación novedosa de computación evolutiva en ingeniería de software. Un enfoque que aborda un problema relacionado, pero diferente, es el asociado con la herramienta SUSHI [2]. El objetivo con SUSHI es alimentar un algoritmo genético con una condición de ruta, producido por un motor de ejecución simbólico, de modo que una entrada que satisfaga la condición de ruta provista pueda reproducirse utilizando la API de un módulo. Este enfoque supone que se proporciona la API (o el subconjunto de métodos relevantes), en oposición a nuestro trabajo, que aborda con precisión la provisión de la API restringida.

6 Conclusiones

En este trabajo, presentamos un algoritmo evolutivo para detectar automáticamente conjuntos de constructores desde la API de un módulo. Evaluamos nuestro algoritmo en la estructura de datos *NodeCachingLinkedList de Apache Commons*, y descubrimos que es capaz de identificar con precisión conjuntos de constructores que son suficientes y mínimos, dentro de tiempos de ejecución razonables. Hasta donde sabemos, este es el primer trabajo que aborda este problema, que generalmente se trata de forma manual. También mostramos que tecnicas como la verificación del modelo de software, pueden beneficiarse mediante el uso del conjunto identificado de constructores para construir automáticamente controladores eficientes. Se necesita más experimentación, pero dados los resultados en este documento, nuestro enfoque parece muy prometedor. Uno de los mayores desafíos de este trabajo fue la construcción de la herramienta y el posterior uso y adaptacion en el verificador de software Java PathFinder. Logramos que la herramienta nos permitiera generar todas las estructuras limitadas, para un número máximo de k dado, a partir de los métodos del programa API. La solución propuesta funcionó lo suficientemente bien, pero sería deseable evitar la aleatoriedad en el proceso.

Algunos aspectos de nuestro algoritmo genético pueden mejorarse aún más. Por ejemplo, se puede definir una clasificación más poderosa para los tipos de argumentos, en la priorización de métodos según sus complejidades. Además, también se pueden incorporar otras dimensiones, como la complejidad del código, para favorecer métodos más simples. Exploraremos esta dirección como trabajo futuro. Además, la implementación de nuestro algoritmo genético es, para la mayoría de las partes, una implementación evolutiva predeterminada de la biblioteca Java JGap [13]. Por supuesto, las mejoras en el algoritmo evolutivo y el ajuste fino de sus parámetros (p. Ej., Tasa de cruce / mutación) pueden generar

tiempos de ejecución más rápidos, por lo que planeamos investigar esto más a fondo en el trabajo futuro

References

1. Abad, P., Aguirre, N., Bengolea, V.S., Ciolek, D., Frias, M.F., Galeotti, J.P., Maibaum, T., Moscato, M.M., Rosner, N., Vissani, I.: Improving test generation under rich contracts by tight bounds and incremental SAT solving. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013. pp. 21–30 (2013)
2. Braione, P., Denaro, G., Mattavelli, A., Pezzè, M.: SUSHI: a test generator for programs with complex structured inputs. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. pp. 21–24. ACM (2018)
3. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26:1–26:66 (Dec 2011)
4. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004)
5. Fraser, G., Arcuri, A.: Evosuite: Automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. pp. 416–419. ESEC/FSE '11, ACM, New York, NY, USA (2011)
6. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edn. (1989)
7. Pacheco, C., Ernst, M.D.: Randoop: Feedback-directed random testing for java. In: Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion. pp. 815–816. OOPSLA '07, ACM, New York, NY, USA (2007)
8. Ponzio, P., Bengolea, V.S., Politano, M., Aguirre, N., Frias, M.F.: Automatically identifying sufficient object builders from module apis. In: Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (2019)
9. Păsăreanu, C.S., Rungta, N.: Symbolic pathfinder: Symbolic execution of java bytecode. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. pp. 179–180. ASE '10, ACM, New York, NY, USA (2010)
10. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edn. (2009)
11. Visser, W., Mehlitz, P.: Model checking programs with java pathfinder. In: Proceedings of the 12th International Conference on Model Checking Software. pp. 27–27. SPIN'05, Springer-Verlag, Berlin, Heidelberg (2005)
12. Website of the Apache Collections library. `https://commons.apache.org/proper/commons-collections/`
13. Website of the Java Genetic Algorithms Package. `http://jgap.sourceforge.net`