# Hacia la construccion de drivers eficientes en bounded model checking mediante deteccion automatica de builders

Mariano Politano[1,2], Valeria S. Bengolea[1], Pablo Ponzio[1,2], and Nazareno Aguirre[1,2]

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

**Resumen** Las técnicas que permiten mejorar la calidad del software producido son de vital importancia, sobre todo en sistemas críticos. Entre ellas, contamos con técnicas de verificación acotada de software, como el model checking de software, que permiten explorar exhaustivamente todas las ejecuciones posibles del software con entradas de tamaño acotado, y reportar fallas encontradas durante el proceso.
Para llevar a cabo la verificación acotada, los model checkers de software se basan en la definición de *drivers*: combinaciones de métodos que permiten construir las entradas con las que se ejecutará el programa. En este trabajo se observa que la selección de los métodos empleados en la definición del *driver* es de vital importancia para la verificación. Intuitivamente, es deseable seleccionar un conjunto de métodos tan pequeño como sea posible (para mayor eficiencia en el análisis), cuyas combinaciones permitan construir todas las estructuras acotadas para el módulo (para analizar el software con todas las entradas 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.
En este trabajo se propone utilizar una herramienta automática para seleccionar un subconjunto de métodos relevantes de un módulo para la construcción de *drivers* eficientes para *bounded model checking*. Además, se evalúa el enfoque propuesto en el análisis de una propiedad particular del modulo Apache NodeCachingLinkedList, empleando el *model checker* Java PathFinder (JPF). Los resultados muestran que el enfoque de construcción de drivers presentado permite incrementar la eficiencia y la escalabilidad a estructuras de mayor tamaño en el análisis usando JPF.

## 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. Esto es central sobre todo en los denominados sistemas críticos, como lo son los sistemas de equipamiento médico, de control de aviones,

etc, cuyas fallas pueden tener consecuencias fatales. 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 [4,1], la verificación de software [3,10], y los análisis estáticos [2], 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 [10]. 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*). 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 5 de resultados experimentales, esto puede tener un impacto importante en el análisis posterior. 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 [6], que nunca modifican las estructuras sobre las que operan. Por otro lado, cuanto mayor es el número 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 APIs 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 (*Builders*).

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 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 aplicando solamente los métodos llamados *builders* a los *drivers* utilizados por JPF se obtiene una ganacia en tiempo y en la cantidad de las estructuras generadas para la verificacion del software.

## 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) [11]. 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. Además cuenta con una *cache*, que es una lista simplemente encadenada que se forma con los nodos que se remueven 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, luego se puede observar que 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 para crear cualquier estructura posible. También, son minimales dado que cualquiera 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. En la API de NLC hay varios métodos que son observadores, como por

| 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() |

**Tabla 1.** API de Apache NodeCachingLinkedList

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

**Figure 1.1.** driverNLC

ejemplo el método `size` que no modifica la estructura del módulo. Por lo tanto, los observadores son siempre superfluos y nunca deben incluirse en un conjunto de constructores mínimos. El enfoque presentado en [8] trata de reconocerlos de antemano y los descarta de la búsqueda para reducir significativamente el espacio de búsqueda.

Se observa 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 obtener los generadores de objetos. Por ejemplo, se puede lograr utilizando diferentes versiones de `add` o `remove`. El enfoque implementado en el trabajo previo le da prioridad a aquellos métodos que contienen menos cantidad de parámetros y menos complejidad de los mismo. Por ejemplo, existen diferentes versiones de metodos `add` que se encuentran desde la posicion 3 a 8 en la tabla 1. La herramienta seleccionará aquellos métodos que reciben menos parámetros o con menor complejidad de los mismos.

Una vez que alimentamos con toda la API de NCL a la herramienta, nos brindó el subconjunto de métodos, mínimo y suficiente, que se encuentra en la figura 1.1, que serán necesarios para brindarle asistencia a la técnica de *bounded model checking* explicada en la sección 4

## 3. Generación automática de objectos

En esta sección resumiremos la herramienta implementada en un trabajo previo [8] y que utilizaremos para obtener los métodos generadores de objectos a partir de una API.

Para realizar esto, utilizaremos Algoritmos Genéticos que son algoritmos de búsqueda no exhaustivo basado en la idea de hill climbing [9]. 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 más detalles sobre estos algoritmos, recomendamos la lectura de [5].

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 número de métodos de la API. Por ende, la posición $i$-th será 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 1.1), 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 más objetos pueda construir un cromosoma, más alto será el número que retorne la 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 combinación de rutinas y el número máximo de valores primitivos disponibles (por ejemplo, enteros de 0 a k - 1). Para este propósito, se desarollo un prototipo que modifica la herramienta Randoop. Primero, se modifico 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 $k$. Para mas información acerca de la herramienta Randoop, recomendamos leer [7]. Para obtener buenos conjuntos de generadores de objetos, se realizó dos mejoras a la función de ajuste, por un lado, cuando hay dos conjuntos suficientes de generadores, siempre se elige el conjunto con menor cantidad de métodos y de esta manera evitamos incorporar métodos superfluos. Por otro lado, aquellos generadores de objectos con más parámetros, o con parámetros más complejos tienen un impacto negativo respecto de los generadores con parámetros más simples. Luego de esta explicación, la función de ajuste queda definida de la siguiente manera:

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

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. *cantidadDeObjetos* hace referencia a las cantidad de objetos que son generados por la herramienta Randoop personalizada para el enfoque planteado en el trabajo previo. La parte derecha de la sumatoria retorna un valor entre 0 y 1 que permite penalizar a los métodos que generan igual número de objetos de acuerdo a lo explicado anteriormente. En el dividendo de la fórmula general, el primer sumando penaliza los conjuntos que contengan un mayor número de método. Esto lo realiza restándole a 1 el resultado de la división entre el número de métodos en $MT$ y el número de métodos en $M$. La constante *w1* permite aumentar/disminuir el peso de este sumando con respecto al otro. En el segundo sumando se penaliza los conjuntos de métodos con parámetros más complejos. Similar a *w1*, la constante *w2* sirve para darle más o menos peso a esta suma. *PP()* es el número de parámetros primitivos en los métodos que contiene el subconjunto pasado como parámetro de esta función. También, cada parámetro de tipo referencia se le agrega una constante *w3*. *RP()* es el número 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 número de parámetros en $M$ (con el peso agregado de los parámetros de referencia) con el numero de parámetros en $MT$ (también ponderado). El resultado de todo esto, se resta a 1. Finalmente, dividimos esto entre *w1* + *w2* para obtener el número deseado en el intervalo [0,1].

En la evaluación experimental realizada en el trabajo previo de esta herrramienta, establecimos $w1 = 2, w2 = 1, w3 = 2$. Estos valores fueron lo suficientemente buenos para que la herramienta produjera conjuntos de constructores suficientes y mínimos en todos los casos de estudios de la herramienta.

## 4. Construcción de drivers eficientes usando *builders*

Como se discutió anteriormente, Java PathFinder (JPF) [10] es una herramienta de verificación automática de programas, que permite implementar diversos tipos análisis. En este trabajo la utilizaremos para realizar (bounded) model checking de software, es decir, para explorar todas las posibles ejecuciones de un programa con entradas de tamaño acotado. Las propiedades a verificar en JPF pueden especificarse usando la construcción `assert` de Java. `assert` evalúa una condición lógica y produce una falla en el programa en caso de que esta sea falsa en la ejecución corriente. En caso de que la condición sea verdadera, `assert` no tiene ningún efecto sobre la ejecución. Por ejemplo, el siguiente código especifica la propiedad de que para cualquier lista de entrada `t`, insertar al inicio un valor entero `v` (`addFirst(v)`, con `v` entre 0 y `b`, con `b` el límite máximo en la cantidad de enteros a analizar, ver más adelante) que no pertenece a la lista (ver explicación de `Verify.ignoreIf` abajo), y luego eliminar el primer elemento de la lista (el valor agregado recientemente, ejecutando `removeFirst`), resulta en una lista con una cantidad de elementos igual a la original (antes de ejecutar `addFirst` y `removeFirst`; notar que la cantidad de elementos previos se almacena en `oldSize` al inicio).

**Algorithm 1** Propiedad a chequear

```
oldSize ←t.size()
value ←Verify.getInt(0, b)
Verify.ignoreIf(t.contains(v))
t.addFirst(value)
t.removeFirst()
assert oldSize = t.size() : "different size";
```

Este código utiliza dos directivas muy importantes de JPF:

- `Verify.ignoreIf(condicion)`: Evita explorar ejecuciones cuyo estado no cumplen con el predicado `condicion`. En el programa anterior, permite ignorar todas las ejecuciones en las que la lista contiene el valor `v` a insertar. Esto es similar a agregar una precondición a la propiedad a verificar.
- `int i = Verify.getInt(min,max)`: Esta construcción de JPF explora todas las posibles ejecuciones del programa que resultan de asignar a `i` cualquier valor entre `min` y `max`. Esto significa que JPF ejecutará el código luego de esta instrucción con `i=min`, con `i=min+1`, ..., `i=max`. Esto introduce no determinismo, ya que en principio no hay garantías del orden en que se asignarán valores a `i` (aunque JPF tiene opciones para elejir un orden de ejecución particular). En nuestro código previo, esto permite explorar todas las ejecuciones de nuestra propiedad con valores de `v` entre 0 y `b`.

Para poder realizar el análisis de esta propiedad, es necesario proveer a JPF los mecanismos necesarios para generar las listas de entrada para el parámetro `t`. Esto es necesario para cualquier tipo estructurado; notar que los valores enteros pueden generarse fácilmente con `Verify.getInt`. Nuestro objetivo es construir todas las listas posibles de tamaño máximo `b`, que almacenan elementos enteros entre `0` y `b` (verificación exhaustiva acotada). Notar que `b` es un parámetro del algoritmo, dado por el usuario. La generación exhaustiva acotada de estructuras para el parámetro `t` se implementa en el *driver* del algortimo 3. Al inicio, el *driver* selecciona la cantidad de métodos a ejecutar, `nExec`, un número entre 0 y `b`. Como los métodos considerados agregan a lo sumo un elemento a la lista, ejecutar un máximo de `b` métodos resulta en listas de tamaño a lo sumo `b` (ejemplo, ejecutar `b` veces `addFirst`). Cada iteración del ciclo corresponde a la ejecución de un único método, seleccionado también de manera no determinísitica entre todos los disponibles. En el caso en que el usuario no conoce el conjunto de *builders* (y no desea realizar el difícil trabajo de seleeccionarlos manualmente), la solución más segura para evitar descartar métodos importantes es utilizar todos los métodos disponibles en el módulo, como se muestra en el algortimo 2. En el cuerpo del ciclo cada método tiene asignado un único entero, y se elige no determinísiticamente un entero entre 0 y la cantidad de métodos para seleccionar el método a ejecutar la iteracción corriente. Por ejemplo, si `methodN=3` se ejecuta el método `add`. Es fácil ver que la cantidad de ejecuciones posibles a explorar por JPF crece exponencialmente con la cantidad de métodos disponibles (por cada método que se ejecuta en una iteración, hay `m` posibles métodos para ejecutar en la iteración siguiente).

Si bien evitar este crecimiento exponencial no es posible en muchos casos, en este trabajo se propone utilizar sólo los builders detectados por el algoritmo genético explicado anteriormente para construir *drivers*, para acelerar la verificación de propiedades en JPF y mejorar su escalabilidad en casos que ocurren típicamente en la práctica (como el caso de NodeCachingLinkedList presentado aquí). Como se muestra en la figura 1.1, sólo 3 métodos conforman un conjunto suficiente y minimal para la construcción de `drivers` para NodeCachingLinkedList. Utilizando sólo esos métodos, obtenemos el *driver* del algoritmo 3, que genera exactamente los mismos objetos de NodeCachingLinkedList que el anterior (porque los builders son suficientes y minimales), y por lo tanto JPF explora las mismas ejecuciones de la propiedad con ambos *drivers*. Sin embargo, como se discute en la sección que sigue, eliminar métodos innecesarios del *driver* produce ganancias sustanciales en tiempo y escalabilidad en el análisis de nuestra propiedad de ejemplo usando JPF.

**Algorithm 2** Driver API

```
t ← new NLC()
nExec ← Verify.getInt(0, bound)
for i = 0...nExec do
   methodN ← Verify.getInt(0, nMet -1);
   switch ( methodN )
      case 0:
         t.clear()
      case 1:
         t.removeFirst();
      case 2:
         t.addLast(Obj)
      case 3:
         t.add(t)
      ...
      case 28:
         t.getLast()
      case 29:
         t.get(Verify.getInt(0, bound))
      case 30:
         t.set(Verify.getInt(0, bound),Obj)
      end switch
end for
```

**Algorithm 3** Driver con Herramienta

```
t ← new NLC()
nExec ← Verify.getInt(0, bound)
for i = 0...nExec do
   methodN ← Verify.getInt(0, nMet -1);
   switch ( methodN )
      case 0:
         t.addFirst(Obj)
      case 1:
         t.removeFirst()
      end switch
end for
```

## 5. Resultado Experimental

En esta sección nosotros realizaremos una evaluación de cómo se comporta la herramienta, implementada en el trabajo previo ya citado, para calcular los generadores de objetos en la estructura Apache NodeCachingLinkedList. Luego, realizaremos una comparación para demostrar como se puede mejorar (a partir de la obtención de métodos generadores) la eficiencia de una técnica de *bounded model cheking*, más precisamente JPF. Todos los experimentos fueron corridos en máquinas con cuatro core Intel i7-6700 3.4GHz con 8GB de RAM, con el sistema operativo GNU/Linux. La primera parte del análisis consiste en chequear

la herramienta comentada en la sección 3. Pudimos examinar que los constructores identificados fueron los ya mencionados en la figura 1.1. La API usada como ejemplo contiene un total de 34 métodos, y el tiempo promedio de las 5 corridas fue de 1744 segundos. Inspeccionamos manualmente los resultados y descubrimos que el conjunto de generadores identificados automáticamente (*buiilders*) 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 basa en la utilización de los métodos generadores obtenidos (*builders*), en el contexto de un análisis de programa, mas específicamente en la herramienta Java PathFinder. Estas rutinas generadoras pueden usarse, por ejemplo, para la creación de *drivers* útiles para guiar el análisis. Como ya se mostro anteriormente, el primer gran aporte a JPF es se encuentra en la escritura de los *drivers*. Queda claro que programar el *driver* a partir de todos los métodos de la API es una tárea compleja que requiere mas conocimento del modulo bajo análisis como muestra el algoritmo 2. Para compensar esto, mostramos que el algoritmo que se utiliza como *driver* en caso de obtener previamente los *builders* es más sencillo y entendible para el usuario.

Luego, se realizo una comparación para determinar el tiempo (en segundos) que le lleva al verificador de software en comprobar una propiedad deseada, como la que se detalla en el algoritmo 1. En la tabla 2 se expone el tiempo que consume el verificador de software para diferentes *scopes* (tamaño maximo de las listas). Recordemos que en *bounded model checking* es necesario brindarle una cota, que cuanto más grande es, mas en profundidad podremos analizar las estructura y por ende, permite tener más posiblidad de detectar *bugs*, tal cual lo explicamos en secciones anteriores. Se observa en la tabla 2 que luego de un cota de tamaño 8, el *driver* sin la previa obtencion de los *builders* se queda sin tiempo (**TO**), con un presupuesto de tiempo de 1800 segundos, mientras que el *driver* creado a partir de la obtencion de los *builders* verifica la propiedad en menor tiempo.

| Scope | Driver S/Builders | Driver C/Builders |
|---|---|---|
| 3 | 9 | 1 |
| 5 | 250 | 3 |
| 8 | **TO** | 23 |
| 11 | **TO** | 185 |
| 13 | **TO** | 881 |

**Tabla 2.** Resultados de verificar la propiedad con JPF (en segundos)

## 6. Conclusiones

En este trabajo, utilizamos una herramienta que dectecta automaticamente conjuntos de métodos generadores de objectos desde una la API para construir *drivers* que mejoren la eficiencia en técnicas de *bounded model checking*. La técnica propuesta fue analizada en un caso de estudio utilizando una implementación de listas con caché, y muestra resultados muy promisorios en cuanto a los ganancias en la eficiencia y escalabilidad obtenidos del análisis de una propiedad usando JPF.

En este trabajo los *drivers* se construyeron manualmente a partir de los builders que nos dió la herramienta presentada en [8]. Como un desafios a resolver es la idea de automatización de todo el proceso de la construccion de drivers. Esto quiere decir, lograr generar de manera automática los *drivers* a partir de la herramienta , la cual, de manera automatica, genera los *builders* de un modulo. Lo que resta resolver, es realizar un trabajo y una inspección en la tecnica de *bounded model checking* mas precisamente en JPF para lograr unir ambas herramientas. Vale destacar, que dentro de los desafios que se propone está el de realizar pruebas en nuevos casos de estudios.

## Referencias


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. 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)
3. 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)
4. 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)
5. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edn. (1989)
6. Huang, W., Milanova, A., Dietl, W., Ernst, M.D.: Reim & reiminfer: Checking and inference of reference immutability and method purity. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 879–896. OOPSLA '12, ACM, New York, NY, USA (2012)
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. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edn. (2009)
10. 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)
11. Website of the Apache Collections library. `https://commons.apache.org/proper/commons-collections/`