Inferencia Automática de Invariantes de Representación basada en Modelos de Lenguaje de Gran Escala y Testing

Valentín Gabriel Buttignol

—

Director: Dr. Pablo Ponzio
Codirector: Lic. Agustín Borda

—

Universidad Nacional de Río Cuarto
Facultad de Cs. Exactas Físico-Químicas y Naturales
Departamento de Computación

1 de julio de 2025

Acknowledgement

Índice general

Resumen

La especificación formal de *invariantes de representación* es fundamental para garantizar la corrección funcional del software orientado a objetos, pero su escritura manual resulta compleja y propensa a errores. Este trabajo presenta AtrIA, una herramienta que automatiza la inferencia de invariantes de representación para clases Java mediante el uso combinado de modelos de lenguaje de gran escala (*LLMs*) y la *generación automática de tests* con Randoop. AtrIA emplea estrategias novedosas de *prompting*. En particular, se propone un enfoque modular que descompone el problema de aprender un invariante en subproblemas, que consisten en aprender las propiedades que componen el invariante por separado.

Se evaluaron cinco LLMs de código abierto (con hasta 14 mil millones de parámetros), y un modelo comercial de gran tamaño, Deepseek (con 670 mil millones de parámetros), sobre seis clases Java de complejidad diversa. Los resultados muestran que el mejor de los modelos pequeños, Distill-Qwen-14B, con el enfoque modular de prompting logró inferir correctamente un 46 % (14 sobre 30) de las propiedades de los invariantes de clase, frente a solo un 30 % (9 sobre 30) que pueden inferirse si se requiere al LLM que infiera el invariante completo. Por otro lado, Deepseek pudo inferir el 46 % de las propiedades con la técnica de prompting modular, y el 40 % (12 sobre 30) de las propiedades con la técnica de prompting tradicional. Destacamos que el enfoque propuesto por AtrIA logra que un modelo pequeño (que cuenta con aproximadamente un 2 % de los parámetros que tiene Deepseek), alcanzó resultados competitivos con Deepseek. Esto muestra que un buen diseño de los prompts puede compensar las limitaciones de tamaño en los modelos.

Por último, este trabajo evidencia el potencial de los LLMs, incluso los de tamaño reducido, para asistir en tareas de especificación formal, siempre que se utilicen estrategias de prompting adecuadas y de verificación de los resultados generados por los LLMs (como Randoop en nuestro caso).

Palabras clave: Invariantes de representación, LLMs, Generación automática de tests, Ingeniería de prompts.

Capítulo 1

Introducción

Uno de los desafíos fundamentales en la construcción de software confiable reside en la evaluación de la corrección funcional del mismo, es decir, en determinar en qué medida el software cumple con lo establecido en sus especificaciones [1]. Para lograr esto, contar con especificaciones formales es de gran ayuda. Un tipo particular de especificación formal consiste en la definición de *invariantes de representación*, es decir, aquellas propiedades que deben mantenerse sobre el estado interno de los objetos de una clase para asegurar su validez durante toda su vida útil [2]. Sin embargo, los desarrolladores rara vez los escriben de manera explícita, ya sea porque la definición resulta difícil, ambigua, o porque simplemente se subestima su importancia. Más aún, cuando los invariantes están presentes, muchas veces son incompletos o están mal especificados, limitando su utilidad en herramientas automáticas de análisis o validación.

Por otro lado, la presencia de invariantes bien definidos no sólo contribuye a una mejor comprensión del diseño de una clase [3], sino que también contribuyen con el mejor rendimiento de distintas herramientas de análisis automático de software, como por ejemplo, las herramientas de generación automática de tests [4, 5, 6, 7], de detección de errores [8] y de verificación [9].

Por otro lado, en los últimos años, la *inteligencia artificial* (IA), y en particular los *modelos de lenguaje de gran escala* (LLMs, por sus siglas en inglés), ha evolucionado significativamente, permitiendo importantes avances en tareas de generación de tests, autocompletado de código, entre muchas otras. En la rama de *ingeniería de prompting*, el desarrollo de técnicas como *Zero-Shot* [10, 11, 12, 13] y *Few-Shot Prompting* [14, 15, 16, 17], *Chain of Thought* [17] o *Text-to-SQL* [12, 13], han demostrado que es posible inducir comportamientos complejos sin entrenamiento específico, simplemente mediante el diseño estratégico de ejemplos y descripciones en lenguaje natural, que luego son usadas como *inputs* de los LLMs.

En este trabajo se propone explorar el uso de LLMs, concretamente de pequeño tamaño, como una nueva forma de inferir invariantes de representación.

1.1. AtrIA

En esta tesis presentamos AtrIA, una herramienta desarrollada para asistir en la inferencia automática de invariantes de representación en clases Java, basada en el uso de LLMs para la generación de invariantes candidatos, y la eliminación de candidatos inválidos respecto de la *interfaz de programación de aplicaciones* (API, por sus siglas en inglés) del software usando una herramienta de generación automática de tests (*Randoop* [8]).

La herramienta opera con un módulo de generación que ejecuta un LLM con distintos estilos de prompting parametrizables, produciendo uno o más candidatos de invariantes. Estos resultados son extraídos y procesados mediante técnicas de análisis estático, para generar versiones instrumentadas de la clase original con los invariantes inyectados. Posteriormente, AtrIA verifica de forma automática la validez semántica de cada invariante utilizando Randoop, una herramienta de generación automática de tests basada en aleatoriedad [18]. La verificación se realiza sobre instancias construidas por Randoop ejecutando métodos de la API pública de la clase. De esta forma, los métodos son utilizados como oráculo, y un invariante candidato es aceptado si todos los objetos construidos con la API satisfacen al mismo. En cambio, si al menos un test construye objetos que violan alguna propiedad del candidato, dicho invariante es descartado. La etapa de verificación con Randoop es fundamental para nuestro enfoque, ya que como se reporta en nuestra sección experimental 4, los LLMs frecuentemente generan invariantes inválidos. Usando Randoop, AtrIA evita reportar invariantes inválidos al programador, que resultan una carga para el mismo porque debe analizarlos y corregirlos de forma manual.

Este trabajo explora distintas estrategias de interacción con los modelos de lenguaje, y evalúa su efectividad en la generación de invariantes. Particularmente se proponen dos estrategias generales de prompting, y a su vez cada estrategia se arma de múltiples componentes (ver más adelante). La primera estrategia, con un enfoque más tradicional, solicita al LLM un invariante como un único método que exprese todas las propiedades que deben cumplir los objetos. La segunda estrategia es una propuesta novedosa de este trabajo, que parte de la observación de que típicamente un invariante de representación es una conjunción de varias propiedades, que pueden inferirse por separado. De esta manera, la estrategia consiste en pedirle al LLM que liste las propiedades del invariante particular, y luego intentar inferir el código de cada propiedad por separado. El resultado es un método que verifica la conjunción de las propiedades válidas inferidas por el LLM.

Los prompts utilizados en ambas estrategias, como se mencionó anteriormente, están conformados por distintos componentes que suelen ser de utilidad para guiar el comportamiento del modelo. Así, el primer componente de todos los prompts, denominado el prompt base, define el rol general del asistente, describe la tarea principal y define el formato esperado de salida. El segundo componente, consiste en agregar "pistas" (*hints*) al prompt base, con el propósito de eliminar sesgos en los modelos para la tarea específica (por ejemplo, para pedirle al modelo que evite la generación de explicaciones innecesarias y genere sólo código). El tercer componente, consiste en agregar ejemplos de soluciones ya conocidas para el mismo problema (ej. invariantes de representación para alguna implementación existente de `LinkedList`, cuando se quiere aprender invariantes de una implementación de árboles). Esto se denomina

few-shot learning, y suele mejorar significativamente la precisión de los resultados, haciendo que el modelo interprete con más exactitud el problema a resolver y el formato esperado de salida [14, 15]. Finalmente, estas tres componentes pueden combinarse, por ejemplo, dando lugar a prompts que incluyan el prompt base, más las pistas, más los ejemplos conocidos.

En nuestra evaluación experimental analizamos la cantidad de invariantes inferidos que se pudieron lograr con 6 clases Java extraídas de la literatura de verificación y análisis de software. Para esto, usamos 5 LLMs diferentes de código abierto y de tamaño pequeño (no más de 14 mil millones de parámetros); junto a 4 técnicas de prompting.

Nuestros resultados muestran que:

- Las estrategias de prompting propuestas en este trabajo, centradas en la descomposición de invariantes en propiedades, superan ampliamente a aquellas que buscan generar directamente un único invariante de clase con todas las propiedades. En particular la mejor variante del enfoque de descomposición, logró inferir 25 invariantes sobre un total de 30, mientras que la mejor variante sobre el enfoque tradicional logró 10.

- A nivel de modelos, se observó que el tamaño del modelo no es el único factor que influye en su rendimiento. La forma en la que fue entrenado, su capacidad para interpretar distintos tipos de prompt, los prompts que recibe como entrada, tienen tanta o más influencia que el tamaño del modelo.

Complementariamente, los resultados también muestran que la combinación de LLMs con verificación dinámica mediante Randoop permite filtrar invariantes incorrectos, actuando como un mecanismo automático de validación. En promedio, el 25 % de los invariantes generados fueron descartados por no cumplir con los tests generados, mientras que otro 24 % contenía errores de compilación. Sin embargo, los porcentajes de invariantes válidos se incrementan significativamente cuando se emplean estrategias de prompting basadas en descomposición de propiedades, donde más del 80 % de las inferencias fueron válidas. Esto refuerza la efectividad de este tipo de prompts, y también evidencia que una buena ingeniería de prompts reduce la cantidad de alucinaciones sintácticas y semánticas.

Además, comparamos los tres modelos de menor tamaño que obtuvieron mejores resultados (Distill-Qwen-14B, Llama3.1-8B y Qwen2.5.1-7B), con uno de gran tamaño a través de un servicio de chat público (Deepseek[1]), usando los dos mejores prompts del experimento anterior (notar que la interacción con Deepseek es manual, ya que su API es paga y no contamos con acceso a la misma, y por lo tanto estos experimentos no pudieron automatizarse). El objetivo de este experimento es analizar la calidad de los invariantes generados por los modelos pequeños con respecto a los inferidos por modelos más grandes, y en teoría más poderosos. Para los experimentos se consideraron un total de 30 propiedades distribuidas entre las 6 clases referidas previamente. Los resultados obtenidos muestran que:

- La generación modular de invariantes también da mejores resultados para el modelo más grande (Deepseek).

[1] `https://chat.deepseek.com/`

- Deepseek obtuvo el mejor desempeño general, infiriendo correctamente 14 propiedades con el enfoque de descomposición y 12 con el enfoque de un único método, seguido muy de cerca por Distill-Qwen-14B, que alcanzó 14 y 9 propiedades completas con las técnicas mencionadas respectivamente.

- Modelos de tamaño pequeño y buen rendimiento, como Distill-Qwen-14B, lograron resultados competitivos frente al modelo más grande, logrando este modelo inferir la misma cantidad de propiedades que Deepseek con la mejor estrategia de prompting propuesta en AtrIA. Esto muestra que un buen diseño de prompts tiene el potencial de compensar las grandes diferencias en tamaño de los LLMs evaluados (Qwen tiene 14B de parámetros, mientras que Deepseek tiene 671B).

El resto de esta tesis se organiza de la siguiente manera. El Capítulo 2 presenta el marco teórico de este trabajo separado en tres secciones. La Sección 2.1 explaya los fundamentos teóricos sobre invariantes de representación y sus beneficios. La Sección 2.2 introduce los Modelos de Lenguaje de Gran Escala. La Sección 2.3 describe brevemente Randoop, la herramienta de testing utilizada para validar los invariantes. El Capítulo 3 detalla la arquitectura y el funcionamiento de AtrIA. El Capítulo 4 expone la evaluación experimental, sus resultados y un análisis de los mismos. Finalmente, el Capítulo 5 ofrece las conclusiones generales y plantea líneas de trabajo futuras.

Capítulo 2

Preliminares

2.1. Invariantes de representación

Una *especificación* es una descripción de lo que debe hacer un programa. Estas son necesarias, debido a la relatividad en la noción de *correctitud* para una pieza de software. Responder si un programa es correcto o no, es una pregunta sin respuesta si no disponemos de la especificación [2].

En la programación orientada a objetos, una *clase* representa un conjunto de *objetos* del mundo real que aparecen en el dominio de un problema. El concepto de clase proporciona una forma estructurada de definir e implementar nuevos tipos de datos que extienden a los tipos predefinidos en un lenguaje de programación, a través de una fusión entre *atributos* como representación del estado interno del objeto, y *métodos* como operaciones que modifican dicho estado. Los nuevos tipos a menudo encapsulan *propiedades* que deben ser válidas a lo largo del ciclo de vida del objeto, reflejando la intención de uso y representación del desarrollador. Una forma de hacer explícitas estas propiedades es mediante los denominados *invariantes de representación* [3] o *invariantes de clase* [2], que una vez especificados formalmente sirven para asegurar la consistencia y correctitud del estado interno los objetos de la clase.

En otras palabras, el invariante de representación de una clase se basa en un conjunto de propiedades que cada instancia de la misma deberá satisfacer:

- En la creación de la instancia, tras el uso de un constructor.
- Antes y después de cada método de la clase, es decir, siempre que el objeto sufra modificaciones (o no) debido a la ejecución de un método.

A pesar de su nombre un invariante puede no satisfacerse todo el tiempo, ya que durante la ejecución de la rutina de una clase, el objeto puede, momentáneamente, entrar en estados intermedios que no cumplan la especificación con el objetivo de modificar el estado de dicho objeto pero que a la larga se reestablezca el invariante.

2.1.1. Invariantes de clase como rutinas imperativas

Formalmente, un invariante de clase es un predicado que determina qué objetos de la clase son legítimos. A raíz de esto podemos expresar un invariante de representación como un método de la clase que retorna un booleano, comunmente denominado, `repOK` [3]. Así, `repOK` retorna verdadero para los objetos válidos; y falso en caso contrario.

La Figura 2.1 presenta un ejemplo de una versión simplificada de la clase `LinkedList`, perteneciente a la biblioteca estándar `java.util`. Esta clase modela una secuencia de elementos mediante una lista doblemente enlazada, y dispone de dos atributos principales: `header`, que referencia el primer nodo de la lista, y `size`, que indica la cantidad de elementos almacenados. Cada nodo de la lista está representado por una instancia de la clase `Entry`, la cual contiene los campos `element` (el valor almacenado), `next` (el nodo siguiente) y `previous` (el nodo anterior). A partir de la mera inspección de estos campos, la intención del desarrollador respecto a ciertas restricciones estructurales de la lista (como su carácter cíclico o acíclico) podría no ser evidente de forma inmediata. No obstante, dicha intención puede, en muchos casos, deducirse a partir del comportamiento de los métodos definidos en la clase. Por ejemplo, si se pretende que la lista sea cíclica, el invariante de clase debe asegurar que la estructura forme un bucle cerrado, manteniendo la coherencia entre los punteros `next` y `previous`, y que el campo `size` refleje con precisión el número total de nodos. En este ejemplo, `repOK` verifica las dos propiedades, la primera, implementada en `isCircular`, que asegura que todos los nodos deben estar correctamente enlazados hacia adelante y hacia atrás, y el último nodo debe apuntar nuevamente a `header`; y además, se verifica que ningún enlace sea `null` en medio del recorrido, lo que garantizaría la integridad de la estructura circular. La segunda propiedad, verificada en `isSizeOk`, asegura que el campo `size` refleje con precisión la cantidad de nodos almacenados en la lista, contando cada uno durante un recorrido desde `header`.

Preservación del invariante

Al diseñar una abstracción de datos, la definición del invariante de representación constituye un aspecto fundamental que debe establecerse antes de implementar cualquier operación. La formulación explícita del invariante de representación ayuda a clarificar los requisitos y restricciones que rigen la estructura de datos. Al hacer explícitas las propiedades del programa, se reduce la probabilidad de errores lógicos. Además, como veremos más abajo, un `repOK` facilita la verificación del comportamiento esperado de la clase.

Toda operación definida sobre la abstracción debe diseñarse e implementarse de manera que preserve el invariante de representación. De lo contrario, la estructura podría alcanzar un estado inconsistente. La implementación del método `remove` en la Figura 2.2 para la clase `LinkedList` permite ver un ejemplo. La operación primero comprueba si el parámetro `e` es igual al atributo `header` de la clase (en tal caso se lanza una excepción), y luego se realizan las asignaciones que permitirían remover a `e` de la instancia. Según el `repOK` correspondiente a la clase, esta implementación es claramente incorrecta ya que el resultado de utilizar esta operación puede provocar que `size` no sea consistente con la cantidad de elementos almacenados en la lista.

```
public class Entry {
    private Entry next, previous;
    private Object element;
    ...
}
public class LinkedList {
    private Entry header;
    private int size;
    ...
    public boolean repOK() {
        if (!isCircular()) {
            return false;
        }
        if (!isSizeOk()) {
            return false;
        }
        return true;
    }

    public boolean isCircular() {
        if (header == null) return false;
        Entry current = header.next;
        while (current != header) {
            if (current == null || current.previous == null || current.previous.next !=
    current)
                return false;
            current = current.next;
        }
        return header.previous != null && header.previous.next == header;
    }
    public boolean isSizeOk() {
        int count = 0;
        for (Entry e = header.next; e != header; e = e.next) {
            count++;
        }
        return count == size;
    }
}
```

Figura 2.1: Declaración de la clase `LinkedList` y su invariante de representación.

```
public class LinkedList {
    ...
    private void remove(Entry e) {
        if (e == header) {
            throw new NoSuchElementException();
        }

        e.previous.next = e.next;
        e.next.previous = e.previous;
        // size--;     BUG! Falta esta linea
    }
    ...
}
```

Figura 2.2: Código del método `remove` con un bug para la clase `LinkedList`.

Si se cuenta con un `repOK` formal, este error puede detectarse fácilmente. Por ejemplo, en el testing manual se puede usar `repOK` en las aserciones de los tests luego de ejecutar el método `remove`. En tal caso, el predicado `isSizeOk` del invariante retornaría `false`, ya que este es el encargado de chequear que el tamaño de la lista sea consistente con respecto a la cantidad de elementos almacenados en la misma. En la Figura 2.3 se ilustra un ejemplo que asume una implementación correcta de `add` y detecta el bug sobre `remove` a través de una aserción sobre el invariante.

```
public void test() {
    LinkedList list = new LinkedList();
    // add entries
    Entry e1 = new Entry();
    Entry e2 = new Entry();
    list.add(e1);
    list.add(e2);
    // size == 2
    list.remove(e1); // size should be 1.
    assert list.repOK() : "Invariant violated after remove!";
}
```

Figura 2.3: Test fallido que viola el invariante de representación de la clase `LinkedList`.

Usos del invariante de representación en el análisis de software

En la perspectiva de la *generación automática de tests*, el `repOK` puede usarse para la generación exhaustiva acotada de objetos, como la implementada por la herramienta *Korat* [4]. Korat genera automáticamente casos de test válidos para estructuras de datos en Java explorando todas las posibles configuraciones de objetos dentro de un tamaño límite, y usando el método `repOK` como filtro, lo que le permite conservar únicamente aquellos objetos que satisfacen el invariante. El objetivo es que cada objeto generado sea estructuralmente correcto. Otros trabajos han demostrado que la integración de contratos ricos y métodos `repOK` mejoran significativamente la eficiencia y la precisión de la generación automática de entradas para tests [5, 6, 7].

En el ámbito de la *detección de errores*, los `repOKs` se comportan como oráculos para identificar violaciones de invariantes durante la ejecución. Estos pueden usarse como oráculos tanto en tests manuales como en tests generados por herramientas automáticas, para detectar errores en el software bajo test. Por ejemplo, Randoop (basado en *generación aleatoria dirigida por retroalimentación* [8]) está diseñado para usar `repOKs` como aserción que debe valer en todos los tests que genera.

En resumen, la incorporación de invariantes de representación como métodos en clases Java no solo favorece la detección de errores en el software, sino que además potencia las capacidades de herramientas automaticas de análisis de software, como pueden ser las herramientas de testing mencionadas anteriormente, como así también otras técnicas de depuración, verificación formal, etc. Su uso representa una práctica clave para elevar el nivel de confiabilidad en el desarrollo de software orientado a objetos, al establecer una frontera clara entre los estados válidos e inválidos de los objetos que componen el sistema.

2.2. Modelos de Lenguaje de Gran Escala

En la última década, la *inteligencia artificial (IA)* ha experimentado avances significativos, permitiendo la creación de modelos de lenguaje capaces de generar, interpretar y manipular texto de manera sofisticada. Dentro del campo de la computación, los *Modelos de Lenguaje de Gran Escala (Large Language Models, LLMs)* representan una de las innovaciones más importantes de los últimos tiempos, con aplicaciones que van desde la asistencia en la programación hasta la optimización de sistemas de búsqueda y recuperación de información, entre muchas otras.

Los LLMs son modelos de redes neuronales profundas entrenados sobre grandes volúmenes de datos, utilizando técnicas avanzadas de aprendizaje automático, como el aprendizaje supervisado y el aprendizaje por refuerzo. Su arquitectura se basa en *transformadores* [19], un tipo de modelo de aprendizaje profundo que permite capturar dependencias de largo alcance en secuencias de texto, mejorando la coherencia y relevancia de las respuestas generadas.

El entrenamiento y la ejecución de un LLM requiere una gran cantidad de recursos computacionales, incluyendo hardware especializado como unidades de procesamiento gráfico (GPUs) o unidades de procesamiento tensorial (TPUs). Además, el proceso de entrenamiento implica la exposición a grandes corpus de datos. Sin embargo, este enfoque también plantea desafíos en términos de veracidad de la información generada, los sesgos que puede generar en las respuestas, y el alto consumo energético asociado al entrenamiento de estos modelos [20].

A pesar de sus capacidades avanzadas, los LLMs no poseen una comprensión semántica real del código que generan o analizan, ya que operan a partir de correlaciones estadísticas en los datos de entrenamiento en lugar de razonamiento lógico o algorítmico. Por ello, su implementación en entornos de desarrollo de software debe considerar mecanismos de verificación y validación que permitan mitigar errores y garantizar la calidad del código producido [21].

2.2.1. Interacción con un LLM

Las distintas formas de interacción con un LLM determinan en gran medida la precisión de los resultados que el LLM produce. Por lo tanto, el estudio de las formas de interactuar con el LLM es clave para el desarrollo de aplicaciones basadas en este tipo de modelos. Entre los elementos que forman parte de las interacciones destacamos: las *inferencias*, los *roles* y los *prompts* [22], cada uno con una función específica en la comunicación entre el usuario y el modelo.

Inferencias

Las inferencias representan las respuestas generadas por un LLM en función de la entrada proporcionada por el usuario. Estas respuestas pueden adoptar diversas formas, desde texto estructurado hasta código fuente, dependiendo del contexto y la naturaleza del modelo utilizado. La calidad de una inferencia depende de múltiples factores, como la arquitectura del modelo, los datos de entrenamiento, los parámetros del modelo, y los prompts empleados.

Prompts

El prompt es la entrada inicial proporcionada a un LLM y juega un papel fundamental en la obtención de respuestas relevantes. La calidad y precisión del prompt determinan en gran medida la efectividad del modelo en la generación de respuestas útiles. Existen distintas estrategias para mejorar la formulación de prompts. En particular, la denominada ingeniería de prompts (Prompt Engineering) busca optimizar la forma en que se presentan las instrucciones al modelo para mejorar su rendimiento [16]. Además, para aquellos prompts que implican una interacción conversacional con el LLM, es posible emplear mensajes con distintos roles; que resultan útiles para, por ejemplo, presentar ejemplos de respuestas similares al LLM, al estilo de few-shot learning. Más adelante, en el Capítulo 3 se presentan ejemplos.

Roles

Los roles en un LLM definen la naturaleza de la interacción entre el usuario y el modelo. Generalmente, se identifican tres roles principales:

- *Sistema*: Un rol que típicamente se usa para proporcionar instrucciones generales sobre el problema a resolver. También puede contener información sobre el tono, estilo o restricciones en la generación de la respuesta esperada.
- *Usuario*: En el contexto de una conversación, sirve para presentarle al modelo una consulta que realiza un usuario. Esta consulta puede ser respondida en el mismo prompt con el rol de asistente (para informarle al LLM como debería ser una respuesta), o bien podría ser la consulta final que el LLM debe responder.
- *Asistente*: En el contexto de una conversación, sirve para informarle al modelo cuál es la respuesta correcta a la consulta previa (realizada con el rol de usuario). En este rol,

el LLM no busca generar una respuesta, sino que "aprende a comportarse" a partir del ejemplo provisto.

En la Figura 2.4 se muestra un ejemplo de cómo es la estructura de una conversación en formato JSON, para interactuar con un LLM. La clave `messages` contiene un arreglo de objetos, en donde se especifica el rol y el mensaje mediante las claves `role` y `content`, respectivamente. Así, el JSON de la Figura consiste de una única consulta, en la que el rol del sistema indica que la tarea del LLM es responder a una pregunta sobre matemáticas. Luego se presentan varios ejemplos de preguntas y respuestas (roles user y assistant) en un formato de conversación, y por último, en la última línea se plantea la consulta al LLM, que debería responder cuál es el resultado de la suma "2+2".

Los roles permiten estructurar la comunicación con el modelo y establecer límites en la generación de texto (por ejemplo, ayudando al modelo a entender con mayor precisión las preguntas, y el formato esperado para las respuestas), asegurando que las respuestas sean coherentes con los objetivos propuestos en la interacción.

2.2.2. Limitaciones

Aunque los LLMs poseen buenas capacidades para generar respuestas coherentes en muchos casos y en ámbitos de aplicación diversos, su implementación efectiva enfrenta múltiples dificultades. Estas incluyen las diferencias entre los modelos abiertos y cerrados, variaciones en el rendimiento según el tamaño y el tipo del modelo, y limitaciones particulares de aquellos con menor capacidad.

El rendimiento de un LLM depende fundamentalmente del número de *parámetros* que posee, que son los valores internos que el modelo ajusta durante el entrenamiento para aprender patrones y relaciones en los datos. Los modelos con más parámetros ofrecen respuestas más precisas, pero requieren más recursos computacionales para su entrenamiento, así como también para su uso posterior para hacer inferencias. Otra de las dificultades es que los LLMs deben entrenarse con grandes volumenes de datos, y que su entrenamiento requiere de importantes recursos de hardware. Típicamente, este tipo de hardware y estas cantidades de datos no estan disponibles para cualquier usuario, por lo que la mayoría de los LLMs disponibles son preentrenados por grandes empresas que cuentan con los recursos necesarios.

```
{"messages":
    [
        {"role": "system", "content": "Please answer the math question."},
        {"role": "user", "content": "1+1=?"},
        {"role": "assistant", "content": "2"},
        {"role": "user", "content": "1+2=?"},
        {"role": "assistant", "content": "3"},
        {"role": "user", "content": "2+2=?"}
    ]
}
```

Figura 2.4: Ejemplo en formato JSON de prompt con roles establecidos.

Respuesta esperada:
```
4
```

Respuesta con alucinaciones:
```
The answer is 5.
```

Figura 2.5: Ejemplos de respuesta esperada y con alucinaciones de un LLM.

Algunas de estas empresas liberan modelos preentrenados, con distintos tamaños. En particular, en este trabajo usaremos modelos preentrenados disponibles libremente para su uso no comercial. Nos centraremos en los modelos pequeños, que cuentan con hasta 14 mil millones de parámetros, y que pueden ejecutarse en una estación de trabajo con hardware modesto (un CPU de 16 núcleos y 32Gb de memoria RAM; sin GPU). Notar que los modelos más grandes, como la última versión de ChatGPT, se estima que tiene más de un trillón de neuronas, esto es, dos órdenes de magnitud más neuronas que los modelos usados en este trabajo.

Derivado de estas diferencias, surgen problemas específicos que afectan la aplicabilidad de los LLMs más pequeños en entornos reales. Entre las principales debilidades este tipo de modelos destacan las *alucinaciones*, es decir, respuestas incorrectas o irrelevantes derivadas de una menor capacidad para interpretar el contexto y desambiguar términos a partir de los prompts (cabe destacar que los modelos más grandes también alucinan, sólo que sus respuestas en general son mucho más precisas, y lo hacen en una menor cantidad de casos). La Figura 2.5 ilustra el ejemplo de un modelo alucinando ante el prompt mostrado en la Figura 2.4, en donde se muestra cuál es la salida esperada por el usuario del LLM (teniendo en cuenta las intenciones del prompt), y a continuación se presenta un ejemplo del modelo alucinando. En el contexto del prompt se simula un seguimiento de un patrón de preguntas matemáticas con un determinado formato de respuestas correctas (únicamente un número), pero en la respuesta final el LLM no es capaz de interpretar correctamente el patrón requerido. Esta deficiencia se acentúa en situaciones complejas o con escasa representación en los datos de entrenamiento.

2.3. Randoop

El *testing de software* es una fase crítica en el desarrollo de sistemas confiables y robustos. Consiste en la ejecución controlada de un programa con el objetivo de detectar fallas (*bugs*), validar su correcto funcionamiento y asegurar que cumple con los requisitos establecidos [23]. Aunque el testing puede realizarse manualmente, este enfoque suele ser costoso en tiempo y propenso a errores humanos, especialmente en sistemas complejos.

El *testing automatizado*, esto es, mediante herramientas de software, se ha propuesto como una alternativa o complemento al testing manual, y tiene el potencial de incrementar significativamente la cobertura y la detección de errores de los tests. Esto se debe a que las herramientas de software pueden explorar un espacio mucho más grande de casos de tests,

incluyendo combinaciones de entradas poco intuitivas o raramente consideradas en tests manuales, lo que incrementa la probabilidad de detectar errores sutiles y vulnerabilidades ocultas [18, 24]. Reducción del esfuerzo manual, debido a la eliminación de la necesidad de diseñar casos de forma individual; mejorías en la cobertura de código [25], y detección temprana de defectos en las etapas iniciales del ciclo de desarrollo; son algunos de los beneficios que estas técnicas automáticas de testing pueden proveer.

Randoop [8] es una herramienta de generación automática aleatoria de tests unitarios, que utiliza la retroalimentación provista por la ejecución para mejorar la efectividad de los tests (*feedback-directed random testing*). Su objetivo principal es crear tests válidos y significativos para clases Java ejercitando la API de las clases de manera aleatoria, pero guiada por los resultados de las ejecuciones de los tests.

El funcionamiento de Randoop se basa en un algoritmo iterativo que, en cada paso, selecciona aleatoriamente un método de la clase bajo test y lo invoca con parámetros elegidos aleatoriamente. Estos parámetros pueden ser valores primitivos aleatorios (como enteros o cadenas) u objetos construidos durante la ejecución de los tests generados previamente [8]. La herramienta evalúa el comportamiento del método, incluyendo las excepciones lanzadas y las violaciones de aserciones, y utiliza esta información para descartar secuencias inválidas o redundantes [8].

Randoop destaca por su capacidad para generar dos tipos de pruebas fundamentales: *tests de regresión* y *tests fallidos*, cada una con un propósito distinto pero complementario. Mientras que los primeras buscan garantizar que las nuevas versiones del código no introduzcan errores en el comportamiento existente (no modificado), los segundos se enfocan en descubrir defectos activamente durante la fase de desarrollo.

2.3.1. Tests de regresión

Son tests diseñados para detectar cambios indebidos en el comportamiento del código a lo largo del tiempo. Estos tests se construyen mediante la observación de las salidas producidas por métodos invocados aleatoriamente durante la fase de generación. Randoop captura los valores de retorno, el estado de los objetos y las excepciones no lanzadas, codificándolos como aserciones en los tests unitarios (en formato JUnit[1]).

El mecanismo detrás de estos tests se basa en el principio de que, si el código bajo test se modifica en el futuro, cualquier desviación en el comportamiento esperado (registrado previamente) será identificada como un error. Consideremos como ejemplo a la clase de la Figura 2.6 que representa un procesador de texto simple que toma un prefijo `prefix`, e incluye un método `process` para concatenar cadenas al prefijo y retornar el resultado. Además, tiene el invariante que verifica que el prefijo no puede ser nulo, ni la cadena vacía.

[1] `https://junit.org/junit5/`

```java
public class TextProcessor {
    private final String prefix;

    public TextProcessor(String prefix) {
        this.prefix = prefix;
    }

    public String process(String text) {
        return prefix.concat(text);
    }

    @CheckRep
    public boolean repOK() {
        return prefix != null && !prefix.isEmpty();
    }
}
```

Figura 2.6: Declaración de la clase `TextProcessor` y su invariante de representación.

Al observar el test de la Figura 2.7 podemos ver que Randoop genera una llamada a `process("hi!")` sobre la instancia con el prefijo `"hi!"`, y posteriormente una aserción que verifica comparando el resultado de la llamada al método con la cadena `"hi!hi!"`, lo que es efectivamente verdadero. Si en una versión posterior del código, el método retorna un valor distinto (como `"hi!hi!hi!"`), la prueba fallará, indicando una posible regresión.

```java
    @Test
    public void test1() throws Throwable {
        if (debug)
            System.out.format("%n%s%n", "RegressionTest0.test1");
        TextProcessor textProcessor1 = new TextProcessor("hi!");
        java.lang.String str1 = textProcessor1.process("hi!");
        org.junit.Assert.assertEquals("'" + str1 + "' != '" + "hi!hi!" + "'", str1, "hi!hi!");
    }
```

Figura 2.7: Test de regresión sobre `TextProcessor`.

2.3.2. Tests fallidos

Son tests que Randoop genera específicamente para intentar revelar defectos en el código actual, como excepciones no manejadas, violaciones de contratos, o comportamientos incorrectos. A diferencia de los tests de regresión, estos no se basan en un comportamiento previamente observado, sino en la detección activa de anomalías durante la generación aleatoria. Randoop identifica fallas mediante dos enfoques principales:

Detección de excepciones no esperadas

Durante los tests, una excepción no siempre indica un error, pues los métodos pueden exigir precondiciones desconocidas para Randoop. Por ejemplo, pasar un valor negativo a un parámetro que se espera que sea positivo puede generar una `IllegalArgumentException`.

Dado que Randoop no infiere precondiciones y usa entradas arbitrarias por defecto trata estas excepciones como comportamientos válidos.

No obstante, Randoop puede generar tests de revelación de errores si detecta otro tipo de excepciones, ya que esto puede sugerir fallos en la implementación o violaciones de contratos implícitos. Por ejemplo, si un método lanza una excepción al acceder a un campo no inicializado retornará una `NullPointerException`. Estas excepciones en la mayoría de los casos representan fallas. Randoop puede configurarse para indicar cuáles son las excepciones que reprsentan fallas y cuáles no. La Figura 2.8 ilustra una situación como la indicada, en donde se produce una excepción que representa una falla al realizar la invocación `process(null)`.

```
@Test
public void test01() throws Throwable {
    if (debug)
        System.out.format("%n%s%n", "ErrorTest0.test01");
    TextProcessor textProcessor1 = new TextProcessor("hi!");
    // during test generation this statement threw an exception of type
java.lang.NullPointerException in error
    java.lang.String str3 = textProcessor1.process(null);
}
```

Figura 2.8: Test fallido generado por Randoop para `TextProcessor`.

Violación de contratos

Randoop puede generar tests fallidos cuando encuentra violaciones de especificaciones provistos por el usuario durante la ejecución de los tests. Una forma simple de proveer especificaciones es anotando un predicado con `@CheckRep`, como se muestra en la Figura 2.6, que indica que el método (provisto por el usuario) debe considerarse como un invariante de representación para la clase. Un ejemplo de violación de un invariante de representación se muestra en la Figura 2.9, en la que se puede observar que la construcción de una instancia con un prefijo vacío no es permitida por el invariante de representación de la clase.

```
@Test
    public void test0() throws Throwable {
        LinkedList linkedlist1 = new LinkedList();
        // Check representation invariant.
        org.junit.Assert.assertTrue(linkedlist1.repOK());
    }
```

Figura 2.9: Test fallido que viola el invariante de representación de la clase `TextProcessor`.

En resumen, el testing automatizado es una técnica fundamental en el aseguramiento de la calidad del software moderno, que si se realiza de forma apropiada puede ser de gran ayuda para la detección de errores en el software. Herramientas como Randoop ejemplifican los avances en esta área al combinar generación aleatoria guiada por el feedback de la ejecución, logrando explorar de forma intensiva una gran cantidad de comportamientos de la API de un programa.

Capítulo 3

AtrIA

En este trabajo se propone AtrIA: una técnica que utiliza generación de código usando LLMs y Randoop como método de verificación para inferir invariantes de representación de clases en Java. La Figura 3.1 muestra el esquema general del flujo de trabajo de AtrIA.

Dado un modelo de gran escala, una clase en Java, y el estilo de prompt que se elija entre diversas opciones que se verán más adelante, AtrIA primero pone en funcionamiento a uno de sus módulos cuyo objetivo es ejecutar al LLM y permitirle la posibilidad de que genere uno o varios métodos booleanos que funcionen como invariante de representación para la clase. El generador de `repOKs` determina qué tipo de prompt pre-armado y qué modelo se emplearán para la generación, mediante los parámetros seleccionados. Se ejecuta el LLM con dicho prompt, y al terminar se parsea la salida del modelo y se crea un archivo en el que se inyecta el invariante procesado junto a sus posibles métodos auxiliares. En caso de ser múltiples invariantes, genera diferentes archivos.

Luego, AtrIA emplea al segundo módulo, el cual utiliza Randoop para verificar cada `repOK` generado y comprobar si la clase cumple con el `repOK`. El verificador primero lee el archivo del invariante producido por el generador y mediante la clase `JavaClassEditor` inyecta el invariante en una copia de la clase original. El editor produce ajustes en la clase, verificando que el invariante y sus métodos auxiliares sean sintácticamente correctos mediante la utilización de la librería JavaParser[1], y prepara a la clase para su posterior verificación. Una vez preparada, el módulo procede a generar tests con Randoop, con el objetivo de verificar el `repOK` inyectado respecto de las estructuras que puedan crearse con los métodos de la clase, chequeando que no se generen instancias que violen el invariante. Puede verificarse positivamente, negativamente, o no compilar por una posible alucinación en la generación del modelo. En el caso de un resultado positivo, se crea o actualiza un `composedRepOK` con el invariante verificado y se agregan los cambios en la clase original.

[1] `https://javaparser.org/`

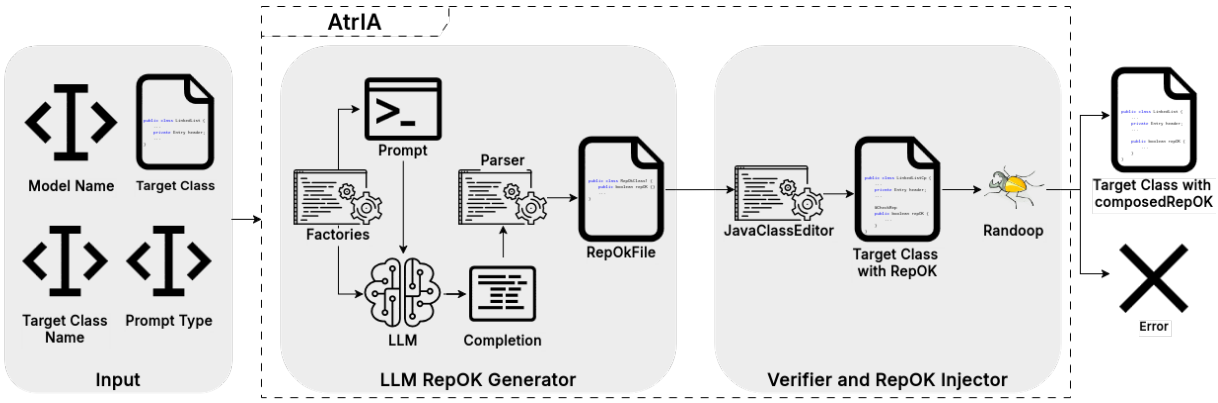

Figura 3.1: Esquema general de AtrIA.

3.1. Generación de invariantes de representación

El primer paso en el flujo de trabajo de AtrIA es la generación de candidatos a invariantes de representación a partir de un modelo de lenguaje de gran escala. Este proceso se basa en la interacción entre el modelo y un prompt preciso y estructurado, diseñado para obtener métodos booleanos que caractericen correctamente las propiedades de la representación interna de una clase en Java.

3.1.1. Construcción de prompts

La aproximación implementada en AtrIA, como se mencionó anteriormente, promueve el uso de diferentes estilos de prompts para la generación de los invariantes de representación. Este trabajo propone que la calidad del invariante generado depende en gran medida de la calidad del prompt utilizado para guiar la generación del LLM, y por lo tanto que un buen diseño de los prompts dará lugar a `repOKs` de mayor calidad.

Dado que múltiples estrategias de *prompting* han demostrado ser efectivas en trabajos recientes, AtrIA incorpora un conjunto de diversas plantillas que permiten experimentar con distintos estilos de construcción de prompts. Estas plantillas están diseñadas para aplicar un estilo específico, o bien una combinación de estilos, según lo determinado por el valor del parámetro de entrada `promptType`. De esta manera, la técnica facilita la exploración sistemática de diferentes enfoques en la formulación de instrucciones al modelo, permitiendo analizar cómo cada enfoque influye en la calidad y precisión de los invariantes generados.

Basic RepOK Prompt (BR)

Esta plantilla propone el uso de *Zero-Shot prompting* [10, 11, 12, 13], en donde la estrategia se caracteriza por dar instrucciones al modelo sin brindarle ejemplos. Es decir, se le solicita directamente al modelo que ejecute una tarea sin proporcionarle ejemplos previos como referencia.

Una buena práctica para los prompts consiste en mantener el contexto lo más simple posible y con la información justa y necesaria para que el LLM pueda resolver la tarea (generar el invariante de la clase) [10, 11, 12, 13]. La idea de este prompt se basa en ese concepto, y un ejemplo se puede ver en la Figura 3.2, en la que se puede observar que el prompt se compone de un mensaje de sistema para introducir el rol del modelo (la primera línea que comienza con ###), junto a la instrucción precisa de la tarea principal a realizar que exige generar el invariante como un método llamado `repOK` (la segunda línea que comienza con ###). Por último, en el mensaje con rol de usuario se incluye la orden final con la tarea de que genere el invariante para la clase recibida como parámetro (encerrada entre ```` '''java ```` y ```` ''' ```` para separar código del texto plano), y debajo de la clase se finaliza con un pie de página (`### repOK:`) para que el modelo complete la inferencia a continuación.

Hints RepOK Prompt (HR)

Este prompt es una continuación del estilo anterior, en donde se amplía levemente el contexto del input con un listado de pistas de ayuda, para que el modelo pueda comprender mejor la tarea a realizar y la respuesta que genere sea más concreta y concisa. El listado de pistas son instrucciones precisas y explícitas para evitar posibles alucinaciones del modelo en la inferencia, que muchas veces surgen de sesgos que tiene el propio modelo, como por ejemplo, proveer explicaciones innecesarias, o dar un formato incorrecto a la respuesta.

Como se puede ver en la Figura 3.3, en el mensaje de sistema las dos primeras líneas son los mismos mensajes que en BR y en la tercer línea (también denotada con ###) se incluyen las pistas sobre las consideraciones que debe tener el modelo (se listan en cuatro items), y el mismo breve mensaje de usuario instruyendo al modelo que debe generar un invariante, seguido del código fuente de la clase. Notar que las primeras dos pistas le indican al modelo qué es un `repOK` y el formato en el que debe generar el mismo (un método booleano de clase). Luego, se proveen pistas para eliminar sesgos típicos de los modelos. La tercera pista indica que el `repOK` debe verificar solo propiedades que el modelo esté seguro que son válidas. El objetivo de esta pista es reducir alucinaciones. Y por último, la cuarta pista indica al modelo que no debe provver ninguna explicación sobre `repOK` (queremos sólo el código del mismo). La mayoría de los modelos generan algún tipo de explicación si no se incluye esta última pista.

Hints Properties Prompt (HP)

Este estilo de prompt combina partes del estilo anterior junto a las técnicas *Chain of Thought prompting* y *Two-Stage design* [17]. El primer método consiste en guiar al modelo para que genere explícitamente los pasos intermedios de razonamiento antes de llegar a una respuesta final, mientras que el segundo consta de formular una interacción múltiple entre el modelo y diversos prompts.

Principalmente, con este procedimiento proponemos la separación explícita del invariante de representación en múltiples propiedades, que pueden aprenderse por separado. La conjunción de estas propiedades conformará el `repOK` generado para la clase. Para aplicar el diseño en dos etapas, elaboramos dos plantillas con instrucciones específicas para cada fase.

System:

```
### You are an expert software engineer with proficiency in the Java programming language.
### Your task is to write representation invariants for Java classes. Answer by giving the
    representation invariant as a Java method called ‘repOK‘.
```

User:

````
### Write a representation invariant for the BinTree class.
```java
public class BinTree {
    private Node root;
    private int size = 0;

    public void add(int x) {
        if (root == null) {
            root = new Node(x);
            size++;
        } else {
            addRecursive(root, x);
        }
    }

    private void addRecursive(Node current, int x) {
        if (x == current.info) {
            return;

        } else if (x < current.info) {
            if (current.left == null) {
                current.left = new Node(x);
                size++;
            } else {
                addRecursive(current.left, x);
            }

        } else {
            if (current.right == null) {
                current.right = new Node(x);
                size++;
            } else {
                addRecursive(current.right, x);
            }
        }
    }

    public static class Node {
        private Node left;
        private Node right;
        private int info;

        public Node(int info) {
            this.info = info;
        }
    }
}
```

### repOK:
````

Figura 3.2: Ejemplo de un *Basic RepOK Prompt* con la clase `BinTree`.

System:

```
### You are an expert software engineer with proficiency in the Java programming language.
### Your task is to write representation invariants for Java classes. Answer by giving the
    representation invariant as a Java method called ‘repOK‘.
### Take into account the following rules for writing the representation invariant:
- The representation invariant is a boolean method of the class.
- The representation invariant must return ‘true‘ if the object is valid, and ‘false‘
    otherwise.
- Only verify properties in the representation invariant that you are completely sure are
    valid.
- Do not provide any explanation.
```

User:

````
### Write a representation invariant for the BinTree class.
```java
public class BinTree {
    ... source code omitted ...
}
```

### repOK:
````

Figura 3.3: Ejemplo de un *Hints RepOK Prompt* con la clase `BinTree`.

La primer fase consta de solicitar al modelo un listado en texto plano de las propiedades que se cumplen en el invariante de representación de la clase en cuestión, permitiéndole al LLM que razone separadamente sobre qué propiedades debe satisfacer la clase, antes de generar el código que las verifica. Una vez obtenido el listado, comienza la segunda etapa en donde para cada propiedad listada en la primer parte, se realizan nuevas consultas al modelo, en donde se le pasa un segundo prompt con la clase y una propiedad específica para la cuál se desea generar código que la verifique. La Figura 3.4 ofrece una ilustración de cómo se realizan las consultas en método presentado en la sección anterior (Single-Stage design), y de la forma en la que funciona el *Two-Stage design* presentado aquí.

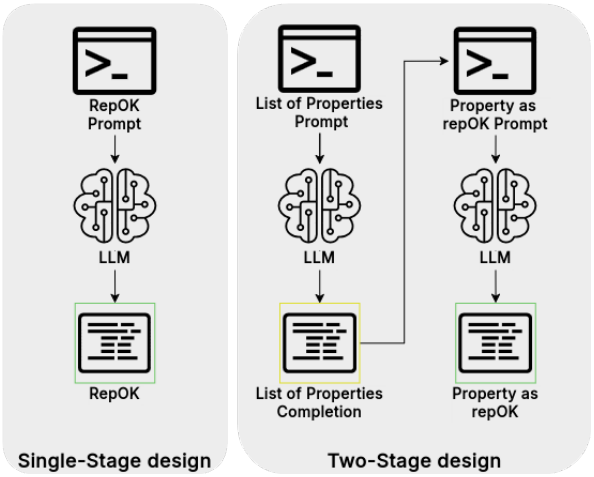

Figura 3.4: Diferencia entre *Single-Stage design* y *Two-Stage Design* prompts.

De esta manera, el prompt de la primera fase del *Two-Stage design* se muestra en la Figura 3.5, en donde se puede observar que, se compone de instrucciones de sistema y pistas similares a las de la técnica HR, con la diferencia de que los mensajes están adaptados para la solicitud del listado de propiedades que la clase satisface. Además en esta ocasión se le encomienda exclusivamente la tarea de armar el listado en un formato específico sin más de cinco propiedades ni explicaciones adicionales.

System:

```
### You are an expert software engineer with proficiency in the Java programming language.
### Your task is to provide a list of properties of representation invariants for Java classes
    as plain text. Answer by giving the list of properties with the format "- Property Name:
    Short Description.".
### Take into account the following rules for writing the properties:
- A representation invariant is a boolean method of the class.
- The representation invariant must return ‘true‘ if the object is valid, and ‘false‘
    otherwise.
- Answer by giving only the list of properties.
- Only list properties that you are completely sure are valid.
- Do not provide more than five properties.
- Do not provide any explanation.
```

User:

```
### Write a list of properties for the BinTree class.
‘‘‘java
public class BinTree {
    ... source code omitted ...
}
‘‘‘

### Properties:
```

Figura 3.5: Ejemplo de la primer etapa de un *Hints Properties Prompt* con la clase `LinkedList`.

Previo a la segunda etapa de prompting, se captura la primera salida, realizando un parseo del listado en texto plano. Se separa cada línea de texto que comience con el formato solicitado en el mensaje de sistema de la primer etapa, y en caso de no coincidir con tal configuración no se realiza el parseo. Luego, se toman las primeras cinco líneas extraídas, ya que por posibles alucinaciones pueden generarse más, y para cada una de ellas se realiza la segunda etapa de prompting. En la Figura 3.6 se muestra un ejemplo del formato esperado para un posible listado de propiedades de la clase `BinTree`.

Como se explicó anteriormente, la nueva fase consiste de realizar una nueva consulta al modelo con un nuevo prompt, solicitando la traducción de la propiedad en texto plano a código fuente en Java (un método booleano). Como se muestra en el ejemplo de la Figura 3.7, el prompt se constituye de la clase para la cual generar el invariante y de la propiedad a codificar que se extraiga de la fase anterior (asumiendo que la Figura 3.6 hubiera sido el primer output, una de las propiedades que se intentaría inferir es la de tamaño consistente). De la misma forma se exige un determinado formato (un método booleano `property`) y se incluyen pistas de ayuda para el modelo, adaptadas a este problema específico.

```
- Consistent size: The size attribute must be equal to the number of nodes in the tree.
- Acyclic structure: The tree structure must be acyclic, i.e., no node should have a path back
    to itself.
- Ordered structure: The tree must maintain the binary search tree property, where the left
    child of a node has a value less than the parent node, and the right child has a value
    greater than the parent node.
- No duplicate values: No two nodes in the tree should have the same value.
```

Figura 3.6: Ejemplo de un posible listado de propiedades en texto plano para la clase `BinTree`.

System:

```
### You are an expert software engineer with proficiency in the Java programming language.
### Your task is to provide the code for a property of the representation invariant for Java
    classes. Answer by giving the property as a Java method called ‘property‘.
### Take into account the following rules for writing the property:
- The property is a boolean method of the class.
- The property must return ‘true‘ if the object satisfies the property, and ‘false‘ otherwise.
- Do not provide any explanation.
```

User:

```
### Write the property of the representation invariant for the BinTree class.
‘‘‘java
public class BinTree {
    ... source code omitted ...
}
‘‘‘

### Property:
- Consistent size: The size attribute must be equal to the number of nodes in the tree.

### Code Property:
```

Figura 3.7: Ejemplo de la segunda etapa de un *System Hints Few-Shot Properties Prompt* con la clase `LinkedList`.

Hints Few-Shot RepOK Prompt (FSR)

El enfoque de esta estrategia es una combinación entre el primer método (HR) y *Few-Shot learning* [14, 15], que consiste en la inclusión de ejemplos en el prompt, similares al problema a resolver para los cuáles la respuesta ya se conoce. Este método de prompting suele dar buenos resultados ya que los ejemplos actúan como elementos de condicionamiento, orientando al modelo en la generación de respuestas más precisas y más adaptadas al contexto del problema particular. En [14, 15], se muestra que los LLMs, logran buenos resultados en una variedad de tareas de análisis automático de software, utilizando únicamente Few-Shot learning, sin necesidad de reentrenar el modelo para la tarea específica.

Este prompt utiliza como base el ejemplo mostrado en HR en la Figura 3.3, pero añade dos ejemplos con clases e invariantes pre-seleccionados, simulando interacciones de mensajes con roles de usuario y asistente para generar una mayor adaptación del modelo al contexto

del problema dado; y después se determina el último mensaje de usuario con la tarea principal a resolver. La Figura 3.8 muestra dicha interacción presentando un primer ejemplo de una versión simplificada de la clase `LinkedList`, que incluye en su declaración una clase privada `Node` para sus elementos y con un método `add`. Para `LinkedList` se simula una respuesta con el `repOK` que verifica propiedades de consistencia de tamaño (`size`), integridad de la lista, y aciclicidad. Cabe recalcar que los mensajes simulados con el rol de asistente también deben cumplir el formato que se exije sobre el método, lo que mejora las probabilidades de que el modelo no produzca alucinaciones que corrompan la generación.

Una de las dificultades principales para la aplicación de Few-shot learning es obtener los ejemplos que se incluirán en los prompts. Aquí usamos una estrategia muy sencilla, usar algunos ejemplos existentes con `repOKs` conocidos, diferentes a los casos de estudio sobre los qué evaluará nuestra técnica. La tarea de definir heuristicas más inteligentes para obtener los ejemplos queda como trabajo futuro.

Hints Few-Shot Properties Prompt (FSP)

En este caso se fusionan las técnicas de la tercer estrategia (HP) con Few-Shot learning (similar a la sección anterior). La combinación es aplicada en ambas fases, ampliando el contexto de aprendizaje durante la ejecución tanto para la generación del listado de propiedades en texto plano como para la generación de código a partir de cada propiedad en texto plano. De la misma manera como se realizó en FSR, se reutilizan los mensajes de sistema para cada fase, los de 3.5 para la fase uno y los de 3.7 para la dos respectivamente, y se agregan las interacciones simuladas para brindarle ejemplos al modelo. En la Figura 3.9 se presenta un ejemplo de la primer fase en donde se muestra un ejemplo de interacción con `LinkedList` y se muestra como respuesta un listado de cuatro propiedades que dicha clase satisface, particularmente, tamaño no negativo, consistencia del atributo `size` respecto con `head`, tamaño consistente respecto de la cantidad total de nodos y aciclicidad. Por otro lado en la Figura 3.10 se ilustra la segunda etapa, en donde se muestra un ejemplo de `LinkedList` con la propiedad de tamaño consistente y la respectiva respuesta con el rol de asistente mediante un método `property`.

System:

```
### You are an expert software engineer
    with proficiency in the Java
    programming language.
### Your task is to write representation
    invariants for Java classes. Answer by
    giving the representation invariant as
    a Java method called ‘repOK‘.
### Take into account the following rules
    for writing the representation
    invariant:
- The representation invariant is a
    boolean method of the class.
- The representation invariant must return
    ‘true‘ if the object is valid, and
    ‘false‘ otherwise.
- Only verify properties in the
    representation invariant that you are
    completely sure are valid.
- Do not provide any explanation.
```

User:

````
### Write a representation invariant for
    the LinkedList class.
```java
public class LinkedList {
    private static class Node {
        int data;
        Node next;
        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }
    private Node head;
    private int size;
    public LinkedList() {
        head = null;
        size = 0;
    }
    public void add(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
        size++;
    }
}
```

### repOK:
````

Assistant:

````
```java
public boolean repOK() {
    if (size < 0)
        return false;
    if (size == 0)
        return head == null;
    if (head == null)
        return false;
    int nodeCount = 0;
    Set<Node> visitedNodes = new
     HashSet<>();
    Node current = head;
    while (current != null) {
        if (!visitedNodes.add(current))
            return false;
        nodeCount++;
        current = current.next;
    }
    return nodeCount == size;
}
```
````

User:

```
(Another class example with the user
    instructions)
```

Assistant:

```
(Another repOK example of the class)
```

User:

````
### Write a representation invariant for
    the BinTree class.
```java
public class BinTree {
    ... source code omitted ...
}
```

### repOK:
````

Figura 3.8: Ejemplo de interacción con roles User/Assistant para *Hints Few-Shot RepOK Prompt* con la clase `BinTree`.

System:

```
### You are an expert software engineer
    with proficiency in the Java
    programming language.
### Your task is to provide a list of
    properties of representation
    invariants for Java classes as plain
    text. Answer by giving the list of
    properties with the format "- Property
    Name: Short Description.".
### Take into account the following rules
    for writing the properties:
- A representation invariant is a boolean
    method of the class.
- The representation invariant must return
    ‘true‘ if the object is valid, and
    ‘false‘ otherwise.
- Answer by giving only the list of
    properties.
- Only list properties that you are
    completely sure are valid.
- Do not provide more than five properties.
- Do not provide any explanation.
```

User:

```
### Write a list of properties for the
    LinkedList class.
‘‘‘java
public class LinkedList {
    private static class Node {
        int data;
        Node next;
        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }
    private Node head;
    private int size;
    public LinkedList() {
        head = null;
        size = 0;
    }
    public void add(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
        size++;
    }
}
‘‘‘

### Properties:
```

Assistant:

```
- No negative size: The size attribute
    must not be negative.
- Size consistency with head: The size
    attribute must be consistent with head
    attribute.
- Consistent size: The size attribute must
    be equal to the number of nodes in the
    tree.
- Acyclic structure: The linked list
    structure must be acyclic, i.e., no
    node should have a path back to itself.
```

User:

```
(Another class example with the user
    instructions)
```

Assistant:

```
(Another list of properties example of the
    class)
```

User:

```
### Write a list of properties for the
    BinTree class.
‘‘‘java
public class BinTree {
    ... source code omitted ...
}
‘‘‘

### Properties:
```

Figura 3.9: Ejemplo de interacción con roles User/Assistant para la primer etapa de *Hints Few-Shot Properties Prompt* con la clase `BinTree`.

System:

```
### You are an expert software engineer
    with proficiency in the Java
    programming language.
### Your task is to provide the code for a
    property of the representation
    invariant for Java classes. Answer by
    giving the property as a Java method
    called ‘property‘.
### Take into account the following rules
    for writing the property:
- The property is a boolean method of the
    class.
- The property must return ‘true‘ if the
    object satisfies the property, and
    ‘false‘ otherwise.
- Do not provide any explanation.
```

User:

```
### Write a list of properties for the
    LinkedList class.
‘‘‘java
public class LinkedList {
    private static class Node {
        int data;
        Node next;
        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }
    private Node head;
    private int size;
    public LinkedList() {
        head = null;
        size = 0;
    }
    public void add(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
        size++;
    }
}
‘‘‘
### Property:
- Consistent size: The size attribute must
    be equal to the number of nodes in the
    tree.

### Code Property:
```

Assistant:

```
‘‘‘java
public boolean property() {
    int nodeCount = 0;
    Node current = head;

    while (current != null) {
        nodeCount++;
        current = current.next;
    }

    return nodeCount == size;
}
‘‘‘
```

User:

```
(Another class example with the user
    instructions)
```

Assistant:

```
(Another list of properties example of the
    class)
```

User:

```
### Write a list of properties for the
    BinTree class.
‘‘‘java
public class BinTree {
    ... source code omitted ...
}
‘‘‘

### Property:
(Parsed property from stage 1 prompt).

### Code Property:
```

Figura 3.10: Ejemplo de interacción con roles User/Assistant para la segunda etapa de *Hints Few-Shot Properties Prompt* con la clase `BinTree`.

3.1.2. Selección y ejecución de modelos

La ejecución de modelos de lenguaje en AtrIA se realiza de manera local utilizando la biblioteca *llama-cpp-python*[2], una implementación ligera y eficiente que permite realizar

[2] `https://github.com/abetlen/llama-cpp-python`

inferencia sobre modelos preentrenados compatibles con el formato *.gguf*. Una ventaja del uso de LLMs de forma local es que se mantiene el sistema autónomo, sin depender de servicios externos, y que se garantiza así la privacidad de los datos.

La selección del modelo a utilizar se configura mediante el parámetro de entrada `modelName`, que identifica el nombre del modelo presente en el repositorio local del sistema. Cada modelo es cargado en tiempo de ejecución, utilizando la biblioteca mencionada, que además permite definir parámetros como el tamaño del contexto, el número de tokens máximos a generar, la temperatura, y otros ajustes que afectan el comportamiento del modelo durante la generación.

Una vez inicializado el modelo, AtrIA construye internamente el prompt de acuerdo con el estilo elegido e invoca al modelo con dicho prompt. Luego la respuesta es enviada a un parser para extraer el método generado por el LLM en el formato final. En caso de seleccionar alguna de las opciones de generación de propiedades en texto plano, este proceso se repite para cada una de ellas.

3.1.3. Parsing y formato de salida

Una vez que el modelo ha generado una salida como respuesta a un prompt que solicita un método booleano en Java, el sistema necesita interpretar dicha salida y extraer de ella los métodos relevantes. Esto se debe a que la definición del `repOK` provista por el LLM puede contener invocaciones a métodos auxiliares. Esta tarea está a cargo de la clase `RepOKParser`, cuya función principal es identificar y aislar correctamente los fragmentos de código que representan métodos `repOK` o `property`, así como cualquier código auxiliar que los acompañe.

El punto de entrada principal es el método `parse`, que utiliza las respuestas del modelo y devuelve una colección de métodos interpretados a partir del texto generado. El código de esta funcionalidad puede verse en la Figura 3.11.

El método `parse` se compone principalmente de la invocación al método `_parse_methods`, que se encarga de procesar los resultados generados por el modelo. En el caso de que se utilice un prompt de tipo de generación de propiedades como texto, puede haber múltiples resultados, uno por cada llamada al modelo. Cada resultado es procesado línea por línea.

Una característica particular que contempla el sistema es que algunos modelos de lenguaje generan una explicación previa al código, marcada explícitamente entre etiquetas especiales. Por este motivo, antes de comenzar a procesar los métodos, se filtran las líneas que se encuentran entre las constantes `OPEN_REASONING_TAG` y `CLOSE_REASONING_TAG`, de forma tal que el parser solo considere el código relevante. Esto se debe a que algunos modelos están entrenados para proveer un texto que explica el razonamiento que realizaron, además de las respuestas.

La detección de los métodos `repOK` o `property` se realiza identificando líneas que comienzan con una de las signaturas esperadas:

- `<access_modifier> boolean repOK() {`
- `<access_modifier> boolean property() {`

En estas definiciones, el componente `access_modifier` representa un modificador de acceso válido, que puede ser `public`, `protected`, `private` o la cadena vacía (este último indica visibilidad de paquete), y se considera parte del patrón de búsqueda. Una vez identificada una línea que coincide con una de estas signaturas, se inicia la captura del cuerpo del método utilizando una variable `snippet`, a la que se le concatenan las líneas subsiguientes. Para determinar el final del método, se lleva un conteo de llaves abiertas y cerradas utilizando la variable `bracket_count`, que permite identificar cuándo el bloque ha sido completamente cerrado.

El parser también asume que, si existen métodos auxiliares definidos por el modelo, estos se encuentran inmediatamente después del método principal. Por tanto, una vez finalizado el bloque de `repOK` o `property`, el sistema continúa analizando las líneas siguientes. Si vuelve a detectar una línea que comienza con un modificador de acceso (por ejemplo, `public`), vuelve a iniciar el conteo de llaves y agrega ese fragmento como parte del código asociado al método principal.

Este enfoque permite capturar tanto el invariante de representación como los métodos auxiliares que el modelo haya generado. Para definir este método primero se observaron manualmente los resultados generados por diversos LLMs, y se concluyó que para la mayoría de los casos era suficiente con considerar un método principal que coincida con el patrón mencionado anteriormente y cero o más métodos auxiliares que se definan debajo del método principal, con el objetivo de capturar la mayor cantidad de inferencias posibles.

Una vez completada la etapa de parsing y obtenidos el código de los invariantes, junto con su posible código auxiliar, el método `parse` continúa su ejecución invocando a la rutina `build_repOK_files`. Esta tiene como objetivo renombrar el invariante generado, para que posteriormente no se generen errores sintácticos al inyectar cada método en la clase objetivo. Además encapsula cada fragmento de código con el nombre del archivo en el que se escribirá. De esta forma, posteriormente, el código de cada invariante se escribe a un archivo de salida denominado `RepOKFileI.txt`, donde `I` representa un identificador numérico incremental asignado a cada invariante parseado, lo que permite mantener un mapeo directo entre cada archivo y su correspondiente código.

```
def parse(self) -> str:
    repok_snippets = self._parse_methods()
    return self._build_repOK_classes(repok_snippets)

def _parse_methods(self) -> list:
    snippets = []
    for completion in self.completions:
        lines = completion.splitlines()
        inside_reasoning = False
        inside_snippet = False
        snippet = ""
        for line in lines:
            stripped = line.strip()
            if stripped == OPEN_REASONING_TAG:
                inside_reasoning = True

            elif stripped == CLOSE_REASONING_TAG:
                inside_reasoning = False

            elif not inside_reasoning and not inside_snippet and
    self._startswith_repoksignature(stripped):
                inside_snippet = True
                bracket_count = 1
                snippet += TAB + line + "\n"

            elif not inside_reasoning and inside_snippet and
    self._startswith_accessmodifier(stripped):
                inside_snippet = True
                bracket_count += 1
                snippet += TAB + line + "\n"

            elif inside_snippet and bracket_count != 0:
                bracket_count += line.count("{") - line.count("}")
                snippet += TAB + line + "\n"
                if bracket_count == 0:
                    snippet += "\n"

        snippets.append(snippet)
    return snippets

def _build_repOK_files(self, repok_snippets):
    files = []
    file_number = 1
    for snippet in repok_snippets:
        modified_snippet = snippet.replace("repOK", f"repOK_{file_number}") \
            .replace("property", f"property_{file_number}")
        files.append((modified_snippet, REPOK_CLASS_FILENAME(file_number)))
        class_number += 1

    return files
```

Figura 3.11: Implementación del método `parse` de la clase `RepOKParser`.

3.2. Verificación de los invariantes mediante Randoop

Una vez obtenidos los candidatos a invariantes por parte del modelo, AtrIA continúa con su segundo módulo, encargado de verificar la validez de cada `repOK` con respecto a la API. Esta etapa tiene como objetivo determinar si las expresiones generadas por el LLM no solo son sintácticamente correctas, sino también son adecuadas para capturar el invariante de representación de la clase original, asumiendo que los métodos provistos al LLM de la clase están correctamente implementados.

Notar que, para cada clase elegimos manualmente unos pocos métodos de la clase para incluir en el prompt. En particular, elegimos un constructor, un método para agregar elementos y un método para eliminar elementos de la estructura. Estos métodos los denominamos métodos generadores de objetos, ya que con ellos pueden construirse todos los métodos de la clase [26]. Esta decisión permite reducir significativamente el tamaño de los prompts, lo que en nuestros experimentos mejora el rendimiento de los LLMs, y sólo requiere asumir que unos pocos métodos de la clase deben ser correctos (por lo que, por ejemplo, pueden testearse estos métodos intensivamente antes de utilizar nuestra herramienta).

Luego del parsing y la compilación del código del `repOK` generado por el LLM, se somete al `repOK` a una verificación con respecto a los métodos generadores de objetos de la API usando la herramienta de generación automática de tests Randoop. Lo que se busca en esta etapa es un *Invariante de Regresión*, donde el criterio de corrección está dado por la capacidad del `repOK` para aceptar todas las instancias generadas a través de invocaciones legítimas a métodos (generadores de objetos) de la clase.

3.2.1. Preparación para la verificación

Una vez obtenidos los candidatos a invariantes por parte del modelo, AtrIA continúa con su segundo módulo, encargado de validar su corrección. Este módulo inicia generando una copia de la clase original, sobre la cual se realizará todo el proceso de verificación, evitando modificar la versión original hasta confirmar la validez del invariante.

El archivo de invariantes generado por el módulo anterior es leído y procesado por una instancia de la clase `JavaClassEditor`, responsable de inyectar los métodos `repOK` directamente en la copia de la clase como si fueran métodos propios. La inyección de código se realiza incorporando el contenido del archivo justo antes de la llave de cierre de la clase, de modo que el nuevo código quede integrado dentro de la definición de la misma. Una vez inyectado el código, se aplican una serie de pasos críticos de preparación antes de la verificación.

En primer lugar, se realiza un análisis sintáctico completo de la clase modificada utilizando JavaParser, una biblioteca para análisis y manipulación de código fuente Java. JavaParser permite representar, recorrer y transformar el árbol de sintaxis abstracta (AST) del programa, facilitando la detección de errores de compilación tras la inyección. Si se detectan errores sintácticos, el invariante se descarta automáticamente.

A continuación, se procede con una evaluación que comprueba si el método inyectado recibe parámetros, si esto sucede el invariante es rechazado. Esto puede surgir debido a alucinaciones del modelo generador. En caso contrario, se actualiza el acceso del método a

`public` y se le añade la anotación `@CheckRep`, lo cual lo habilita para ser interpretado por Randoop como invariante de representación durante la fase de testing.

3.2.2. Verificación con Randoop

Finalizada la etapa de inyección y validación sintáctica de los métodos generados, AtrIA procede a verificar la corrección semántica de cada invariante respecto de la API de la clase utilizando verificación dinámica con Randoop. Este proceso consiste en generar un conjunto de tests que ejercitan los métodos públicos de la clase, a fin de construir objetos legítimos del tipo, y contrastarlos con el invariante aprendido.

A diferencia de una generación arbitraria de estructuras, Randoop produce secuencias de invocaciones a métodos de la API pública de la clase. Esto asegura que los objetos bajo evaluación hayan sido construidos a través de operaciones válidas definidas por el programador. Asumiendo que las rutinas usadas están correctamente implementadas, este enfoque permite utilizar los objetos generados como referencia para validar la adecuación de los invariantes: un buen invariante no debería rechazar ninguna instancia construida mediante el uso correcto de la clase. Como se mencionó anteriormente, sólo utilizamos métodos generadores de objetos (un constructor, un `add` y un `remove`, suficientes para generar todos los objetos de la clase) para la generación de tests con Randoop.

En este contexto, el invariante generado se comporta como un invariante de regresión, cuya corrección se evalúa en función de su capacidad para aceptar todos los objetos que la clase produce. Si durante los tests ocurre una violación de contrato sobre el invariante marcado con `@CheckRep` o alguna excepción causada por el invariante, se interpreta como una señal de inconsistencia, y el método es descartado. En la Figura 3.12 se muestra un ejemplo, asumiendo una implementación simple de la clase `Stack`, con sus métodos `push`, `pop` y un invariante `repOK`.

```
@Test
public void test1() throws Throwable {
    if (debug)
        System.out.format("%n%s%n", "RegressionTest0.test1");
    Stack stack1 = new Stack();
    stack.push(10);
    stack.push(20);
    org.junit.Assert.assertTrue(stack1.repOK());
}
```

Figura 3.12: Ejemplo de test que debe pasar un invariante de regresión para una clase `Stack`

Si esta secuencia genera una instancia válida de `Stack`, y el `repOK` generado por el modelo devuelve `false` sobre ese objeto, entonces el invariante está mal especificado y se descarta. Este tipo de test simboliza el rol del invariante como oráculo de regresión: su validez está sujeta a no contradecir el comportamiento observable de la clase.

3.2.3. Composición de invariantes validados

En caso de superar la verificación dinámica, el invariante se considera válido y se incorpora a un método especial denominado `composedRepOK`, el cual agrupa todos los métodos `repOK_i` (o `property_i`) previamente validados. Este método se construye mediante una manipulación más detallada del árbol de sintaxis usando JavaParser.

Cabe destacar que los enfoques `repOK` y `property` difieren en su granularidad y en la forma en que se generan los invariantes. El enfoque tradicional `repOK` busca obtener un único método que exprese todas las propiedades de la estructura de datos de manera conjunta. Por el contrario, el enfoque basado en `property` descompone el invariante en múltiples propiedades independientes, cada una representada por su propio método. En este contexto, el método `composedRepOK` permite reagrupar automáticamente en un solo método las distintas propiedades validadas tanto por la generación de un único método `repOK` como por múltiples métodos `property`. En otras palabras, `composedRepOK` es el método booleano que representa el invariante de clase generado por AtrIA.

El patrón que sigue `composedRepOK` se muestra en la Figura 3.13, y responde a una lógica condicional acumulativa. Cada invariante validado se transforma en una llamada condicional dentro de este método. Si alguno de ellos falla, `composedRepOK` corta la evaluación y retorna `false`. En caso contrario, se devuelve `true` solo si todos los métodos individuales se cumplen. Esta estructura permite mantener una verificación modular y extensible ante la incorporación de nuevos invariantes, ya que basta con añadir una línea adicional a este método para que se integren automáticamente al chequeo general.

```
public boolean composedRepOK() {
    if (!repOK_1) {
        return false;
    }
    if (!repOK_2) {
        return false;
    }
    ...
    return true;
}
```

Figura 3.13: Formato del método `repOK` compuesto.

Una vez actualizada esta composición, se elimina la anotación `@CheckRep` del método individual previamente evaluado y se consolidan los cambios efectuados en la clase bajo test, trasladándolos a la versión original. Así, se asegura que solo los invariantes validados sean preservados, manteniendo la clase en un estado consistente para continuar con el procesamiento del siguiente candidato.

Capítulo 4

Evaluación

Con el objetivo de evaluar la eficacia de nuestra técnica de generación automática de invariantes de representación, AtrIA, se formularon las siguientes preguntas de investigación:

P1. ¿Cuáles de los enfoques de prompting que dan mejores resultados con LLMs de menor tamaño?

P1 se centra en determinar los prompts más eficaces para modelos de lenguaje de recursos limitados. Para abordar esta cuestión, se llevó a cabo un experimento en el que se midió la proporción de métodos `repOK` generados que lograron superar exitosamente la verificación dinámica automatizada de Randoop. Este análisis permitió comparar distintos enfoques de prompting y modelos, identificando los enfoques de prompting y modelos que dan mejores resultados para esta tarea particular.

P2. ¿En qué medida contribuye la validación dinámica mediante Randoop a la eliminación de invariantes inválidos?

Durante el proceso de inferencia, los modelos pueden generar invariantes que, debido a alucinaciones, pueden no corresponderse con la semántica de la clase o no ser correctos sintácticamente, lo que genera errores de compilación. Estas alucinaciones representan un costo para el programador, quien debería inspeccionarlas manualmente para descartarlos. El objetivo de **P2** es determinar cuántos invariantes no válidos generados por los LLMs son descartados por el módulo de verificación, y en particular Randoop.

P3. ¿Qué tan precisos son los invariantes generados por los LLMs de menor tamaño?

Una vez identificado el esquema de prompting más prometedor y el modelo de mejor desempeño a través de **P1**, para responder **P3** se procedió a una evaluación cualitativa detallada de los resultados. Esta fase del estudio consistió en analizar manualmente las propiedades que los `repOKs` generados verifican, respecto de las propiedades de invariantes de

representación conocidos. El objetivo fue determinar en qué medida los invariantes generados logran capturar los invariantes reales de las clases.

P4. ¿Qué tan precisos son los invariantes generados por LLMs de mayor tamaño, con respecto a los LLMs de menor tamaño?

Por último, esta pregunta explora la capacidad de los LLMs de mayor tamaño para generar invariantes de representación. El propósito de esta tercera etapa fue comparar el rendimiento entre modelos de distinto tamaño, analizando si existe una mejora sustancial en la calidad de los invariantes generados a medida que se incrementa el tamaño de los LLMs.

Estas cuatro preguntas constituyen el núcleo de nuestra evaluación experimental y permiten obtener una visión integral del comportamiento de la herramienta, identificando con mayor precisión las fortalezas y limitaciones actuales de los modelos generativos en el contexto de inferencias de invariantes de representación.

4.1. Diseño Experimental

Para llevar a cabo la evaluación de AtrIA, se definió un entorno de experimentación controlado que permite estudiar el impacto de distintos esquemas de prompting y modelos de lenguaje sobre la generación automática de invariantes de representación. Los experimentos se organizaron en torno a tres dimensiones principales: la variación del modelo de lenguaje, la variación en la estrategia de prompting y la variación de las clases objetivo.

Se seleccionaron cinco modelos de lenguaje de código abierto, de arquitectura diversa y con tamaños que oscilan entre los 7 y 14 mil millones de parámetros. Los modelos utilizados fueron:

- **DeepSeek-R1-Distill-Llama-8B-GGUF** (Distill-Llama-8B)
- **DeepSeek-R1-Distill-Qwen-7B-GGUF** (Distill-Qwen-7B)
- **DeepSeek-R1-Distill-Qwen-14B-GGUF** (Distill-Qwen-14B)
- **Meta-Llama-3.1-8B-Instruct-GGUF** (Llama3.1-8B)
- **Qwen2.5.1-Coder-7B-Instruct-GGUF** (Qwen2.5.1-7B)

Cada modelo fue instanciado localmente, con cuantización de tipo $Q6\_K\_L$, lo cual permitió realizar las pruebas en entornos de cómputo de mediana capacidad sin acceso a GPUs de alto rendimiento.

Dada la imposibilidad de ejecutar localmente modelos de gran escala por restricciones computacionales y presupuestarias, para la alternativa de mayor tamaño, se optó por utilizar un servicio público de chat que proporciona acceso al modelo Deepseek.

Respecto al diseño del prompt, se experimentó con las distintas variantes presentadas en el Capítulo 3. Cada variante fue evaluada de manera sistemática frente al conjunto completo

de clases seleccionadas para el estudio. En los casos que utilizan prompts con Few-Shot Learning, los ejemplos provistos al modelo corresponden a clases distintas a la clase objetivo, sin relación estructural ni semántica directa. Esta decisión busca evitar que el modelo infiera propiedades de la clase objetivo a partir de los ejemplos proporcionados, permitiendo evaluar su capacidad de generalización a partir del conocimiento aprendido durante el entrenamiento. Para los experimentos se seleccionan dos ejemplos entre un conjunto de tres clases que son implementaciones alternativas y más simples de las clases `MinHeap`, `BinTree` y `LinkedList`.

Como casos de estudio se emplearon estructuras de datos ampliamente conocidas y utilizadas, las cuales presentan diferentes características en términos de complejidad de sus invariantes:

- `AvlTree`
- `BinomialHeap`
- `BinTree`
- `LinkedList`
- `NodeCachingLinkedList`
- `TreeMap`

AtrIA se ejecutó para cada combinación de modelo, clase y prompt. La configuración de los parámetros de los LLMs seleccionada para todos los experimentos fueron:

- Temperatura: 0.1
- Cantidad máxima de tokens: 4000
- Cantidad máxima de tokens de contexto: 8192
- Cantidad máxima de tests generados por Randoop: 1500

Por otro lado, los experimentos con Deepseek debieron realizarse de forma manual, por no contar con acceso a la API de la herramienta. Por lo tanto, para reducir la cantidad de casos a ejecutar manualmente, para Deepseek se usaron sólo las técnicas de prompting que demostraron ser más efectivas en **P1**. Posteriormente se empleó el segundo módulo de AtrIA para realizar la verificación de cada resultado y descartar invariantes inválidos de acuerdo a los objetos generados por la API.

4.2. Resultados

4.2.1. P1: Prompts para los modelos de tamaño pequeño

La Tabla 4.1 muestra la cantidad de invariantes verificados correctamente, agrupados por combinación de modelo y técnica de prompting. Cada verificación positiva corresponde a un caso en el que se generó un método `repOK` (o `property`) válidos para uno de los casos de estudio considerados.

La técnica que logró generar más invariantes válidos (propiedades) fue Hints Few-Shot Properties Prompt (FSP), con 25 casos exitosos, seguido de Hints Properties Prompt (HP,

Tabla 4.1: Cantidad de invariantes válidos generados por cada combinación de técnica de prompting y modelo. Cada celda indica el número de clases para las cuales se logró generar un invariante verificado correctamente.

	Distill-Llama-8B	**Distill-Qwen-7B**	**Distill-Qwen-14B**	**Llama3.1-8B**	**Qwen2.5.1-7B**	**Total por prompt**
HR	0	0	1	2	3	6
HP	3	4	5	4	4	20
FSR	1	1	3	2	3	10
FSP	4	4	6	6	5	25
Total por modelo	8	9	15	14	15	

20 casos), Hints Few-Shot RepOK Prompt (FSR, 10 casos) y, en último lugar, Hints RepOK Prompt (HR, 6 casos). Es importante destacar que los invariantes que son inválidos imponen una carga de trabajo importante sobre el programador, ya que este debe analizarlos y corregirlos manualmente antes de poder incorporarlos a la clase en cuestión. Es por eso que la generación de invariantes correctos es de suma importancia en este contexto. Notar que el diseño de los prompts es crucial: los prompts basados en propiedades (HP y FSP) logran siempre aprender más propiedades correctas que los prompts que requieren al LLM un único `repOK` (HR y FSR).

A nivel de modelos, Qwen2.5.1-7B y Distill-Qwen-14B obtuvieron el mayor número de invariantes correctos (15 cada uno), seguidos de cerca por Llama-3.1-8B (14 casos), Distill-Llama-8B (9 casos), y finalmente Distill-Qwen-7B (8 casos). Este resultado sugiere que el tamaño del modelo no es el único factor que influye en su rendimiento. Otros factores que pueden influir en los resultados son: la capacidad de los modelos de adaptarse a distintos tipos de prompt, la forma en que se realiza el entrenamiento del modelo, y muchas otras variables (que no están bajo el control directo de los usuarios de modelos preentrenados).

En conjunto, estos resultados ofrecen un primer panorama sobre el desempeño de diferentes técnicas de prompting y arquitecturas de modelos en la tarea de generación de invariantes.

4.2.2. P2: Invariantes inválidos eliminados

La Tabla 4.2 muestra los porcentajes de validez de los invariantes generados por distintos tipos de prompts: HR, HP, FSR y FSP. Cada fila indica, para su respectivo tipo de prompt, el porcentaje de invariantes que fueron validados exitosamente, los que fueron invalidados por Randoop y los que fueron descartados por errores de compilación. En el total acumulado, aproximadamente el 50,8 % de los invariantes generados fueron considerados válidos, mientras que el 25 % fueron filtrados por Randoop y un 24,2 % presentaron errores sintácticos (por alucinaciones de los modelos) que impidieron su análisis.

Los resultados muestran una clara ventaja de los prompts FSP, donde más del 80 % de los invariantes generados resultaron válidos, sólo un 6,7 % fueron eliminados por la verificación de Randoop y un 10 % contienen errores de compilación. Le sigue el caso de HP, con un 66,7 % de invariantes válidos, un 13,3 % invalidados por Randoop y un 20 % con errores sintácticos. En contraste, los prompts HR y FSR presentan porcentajes de validación considerablemente

más bajos (por debajo del 35 %), con tasas elevadas de invalidación: 43,3 % y 36,7 % por Randoop, y 36,7 % y 30 % por errores de compilación, respectivamente.

Estos resultados sugieren que los modelos tienden a generar menos alucinaciones y una mayor proporción de invariantes válidos cuando se les solicita a partir de propiedades explícitas del invariante, en particular mediante el uso de prompts del tipo FSP.

Tabla 4.2: Porcentajes de validez de los invariantes generados por prompts.

	% Validados	**% Invalidados por Randoop**	**% Invalidados por errores de compilación**
HR	20 %	43,3 %	36,7 %
HP	66,7 %	13,3 %	20 %
FSR	33,3 %	36,7 %	30 %
FSP	83,3 %	6,7 %	10 %
% Total	50,8 %	25 %	24.2 %

4.2.3. P3 y P4: Calidad de los invariantes generados

Si bien es importante generar invariantes válidos, también es importante analizar la completitud de los invariantes generados. Es decir, qué propiedades de los invariantes de representación reales de las clases (ground truth) son verificadas por los invariantes generados. Para evaluar esto analizamos manualmente las propiedades inferidas con respecto a las propiedades esperadas en el invariante de representación real. Dados los resultados en la tabla anterior, seleccionamos los tres modelos pequeños con mejor rendimiento (Distill-Qwen-14B, Llama3.1-8B y Qwen2.5.1-7B), junto al modelo de mayor tamaño (Deepseek); y escogimos los dos estilos de prompting más efectivos pero que contrastan ideas: FSP, centrado en la inferencia directa de propiedades individuales, y el FSR, que genera un único método con múltiples propiedades. Considerando todas las clases, identificamos un total de 30 propiedades esperadas. Para analizar cada generación, las clasificamos de la siguiente forma:

- Propiedades inferidas completamente (✓): propiedades esperadas del invariante ground truth que son verificadas por el invariante generado por AtrIA.
- Propiedades inferidas parcialmente (~): propiedades esperadas del ground truth para las que el invariante generado por AtrIA solo captura una parte de la propiedad (por ejemplo, el invariante inferido omite algún componente relevante) .
- Propiedades no inferidas (×): aquellas propiedades del ground truth que no fueron capturadas por el invariante generado por AtrIA.
- Invariante no generado (-): se utiliza en los casos en los que AtrIA no genera ninguna propiedad válida para la clase. Esto puede deberse a que el LLM no genera invariantes para el caso de estudio, o que ninguno de los invariantes generados por el LLM pasa la verificación de Randoop.

La Tabla 4.3 resume los resultados obtenidos.

Tabla 4.3: Propiedades inferidas por cada propuesta ante cada modelo. Propiedades inferidas completamente (✓), propiedades inferidas parcialmente (~), propiedades no inferidas (×), e invariante no generado (-).

Propiedad esperada	Distill-Qwen-14B		Llama3.1-8B		Qwen2.5.1-7B		Deepseek	
	FSP	FSR	FSP	FSR	FSP	FSR	FSP	FSR
AvlTree								
Es un árbol binario acíclico	✓	-	~	-	-	-	-	-
El tamaño es consistente	×	-	×	-	-	-	-	-
Claves ordenadas	×	-	×	-	-	-	-	-
Árbol balanceado	×	-	×	-	-	-	-	-
Los campos de altura son correctos	×	-	×	-	-	-	-	-
BinomialHeap								
Es un árbol binario acíclico	×	-	✓	-	×	-	×	-
El padre de la raíz apunta a un valor nulo	×	-	×	-	×	-	×	-
Las referencias al padre coinciden	✓	-	×	-	×	-	×	-
El tamaño es consistente	✓	-	×	-	~	-	✓	-
Claves ordenadas	~	-	×	-	✓	-	×	-
Es un BinomialHeap	~	-	~	-	~	-	✓	-
BinTree								
Es un árbol binario acíclico	✓	✓	~	-	~	✓	✓	✓
Claves ordenadas	×	✓	×	-	~	✓	✓	✓
El tamaño es consistente	×	✓	✓	-	✓	~	✓	✓
LinkedList								
Header ≠ null	×	✓	×	✓	×	×	✓	✓
Coinciden anterior y siguiente	✓	✓	×	✓	✓	✓	✓	✓
La lista es una DLL circular	✓	✓	~	✓	✓	✓	✓	✓
El tamaño es consistente	✓	✓	×	×	✓	✓	✓	✓
NodeCachingLinkedList								
Header ≠ null	×	-	×	~	×	-	×	✓
El Header es una DLL circular	✓	-	~	✓	✓	-	~	~
El tamaño es consistente	✓	-	✓	✓	✓	-	×	~
La caché es acíclica	×	-	×	✓	×	-	✓	✓
El tamaño de la caché es consistente	~	-	~	~	~	-	✓	~
No se comparten nodos	~	-	×	×	×	-	×	×
TreeMap								
Es un árbol binario acíclico	✓	~	×	-	×	✓	✓	✓
El padre de la raíz es nulo	✓	✓	×	-	~	~	×	✓
Las referencias del padre coinciden	✓	×	×	-	×	~	×	×
El tamaño es consistente	✓	✓	×	-	×	~	✓	~
Árbol balanceado	~	~	~	-	~	✓	~	~
Claves ordenadas	✓	×	×	-	×	~	✓	✓

Distill-Qwen-14B es el modelo pequeño que más propiedades inferidas completamente logra, con la ejecución de FSP omite algunas propiedades, pero logra cubrir parcial o completamente una buena parte de ellas, sobretodo para `TreeMap` que es una de las clases más complejas; obteniendo un total de 14 propiedades cubiertas en su totalidad. Mientras que con FSR llega a inferir todas las propiedades de `BinTree` y `LinkedList`, junto a algunas de `TreeMap` consiguiendo un total de 9 condiciones verificadas completamente.

Llama3.1-8B, con FSP, en pocos logra inferir algunas propiedades de forma parcial, y a veces logra una inferencia completa. Con FSR infiere algunas propiedades correctamente de `LinkedList` y `NodeCachingLinkedList`, pero en general los resultados de Distill-Qwen-14B son muy superiores.

Para Qwen2.5.1-7B no es muy diferente, aunque es capaz de generar invariantes ligeramente más fuertes en comparación del modelo anterior, debido a que infiere más cantidad de propiedades parciales y completas en ambos estilos de prompt.

Por último, Deepseek es el mejor modelo superando por poco a Distill-Qwen-14B en cantidad de propiedades inferidas completamente, logrando un total de 14 y 12 propiedades correctas para FSP y FSR respectivamente, con una verificación total de las propiedades para `BinTree` y `LinkedList` en ambos prompts.

Estos resultados preliminares muestran que el diseño de prompts es crucial. Con la técnica de prompting propuesta basada en dividir los invariantes en propiedades, AtrIA obtiene resultados con un modelo pequeño (Distill-Qwen-14B) que son comparables a los resultados que se obtienen con un modelo enorme (según la página web oficial Deepseek tiene 671B de parámetros, vs. la versión de Qwen empleada que sólo posee 14B.). Por otro lado, Deepseek logra obtener buenos resultados para el estilo de prompting más tradicional de requerir todo el código del `repOK` de una vez (verificando 12 propiedades en lugar de 14). Creemos que la enorme cantidad de parámetros que posee hacen que este modelo sea capaz de procesar mejor los prompts que le requieren realizar tareas más complejas (aprender el invariante completo en lugar de propiedades más simples por separado).

Observamos que las estructuras con invariantes de mayor complejidad, como `BinomialHeap`, `AvlTree`, `NodeCachingLinkedList` tienden a presentar mayores dificultades para los modelos. `TreeMap` es también una estructura compleja, pero a diferencia de las anteriores es muy utilizada en Java, ya que es parte de la biblioteca estándar `java.util`. Por esta razón, no podemos descartar que haya sido usada en el entrenamiento de los modelos, y que esto resulte en mejores resultados para esta clase en los modelos más grandes.

En contraste, clases como `LinkedList` o `BinTree` presentan invariantes más simples, lo que facilita su inferencia incluso para modelos menos potentes. En general, FSP otorga mejores resultados que FSR, lo que indica que la inferencia de propiedades individualmente es más efectiva que la de un solo invariante, sobre todo para los modelos más pequeños.

Capítulo 5

Conclusiones

La especificación formal de invariantes de representación es una tarea crucial para mejorar la corrección de programas orientados a objetos, pero su automatización efectiva sigue siendo un desafío. En respuesta a esta problemática, presentamos AtrIA, una herramienta automática de inferencia de invariantes de representación que combina el poder expresivo de los LLMs con la verificación mediante generación de test usando Randoop. AtrIA se apoya en la definición de estrategias novedosas de prompting, que guían a los modelos en la generación de mejores invariantes candidatos, como lo muestran los experimentos realizados en el Capítulo 4.

Para evaluar la efectividad de AtrIA en la inferencia automática de invariantes de representación, se llevó a cabo un estudio empírico utilizando 6 clases Java extraídas de la literatura, abarcando distintas estructuras de datos y grados de complejidad interna. La experimentación contempló 4 estrategias de prompting cuidadosamente diseñadas, y su ejecución sobre 5 modelos de lenguaje de acceso abierto, ejecutados localmente como alternativas de menor capacidad, junto a un modelo comercial de gran tamaño accesible vía un servicio público de chat.

Los resultados muestran que las estrategias de prompting propuestas en este trabajo, basadas en la descomposición de invariantes en múltiples propiedades generan invariantes significativamente más precisos y completos que los prompts que solicitan el invariante como un único método. En particular, usando el mejor de los modelos pequeños (Distill-Qwen-14B), AtrIA con la técnica de prompting Hints Few-Shot Properties (FSP) logró inferir 14 propiedades (de un total de 30), contra 9 propiedades inferidas usando la técnica de prompting Hints Few-Shot RepOK (FSR). Adicionalmente, el LLM comercial de gran tamaño Deepseek logró inferir 14 propiedades con FSP, y 12 propiedades con FSR.

Estos resultados muestran que el diseño de prompts es crucial. Notablemente, AtrIA obtiene resultados con un modelo pequeño (Distill-Qwen-14B, con 14B de parámetros) que son comparables a los resultados que se obtienen con un modelo enorme (671B de parámetros). Por otro lado, Deepseek logra obtener buenos resultados para el estilo de prompting más tradicional de requerir todo el código del `repOK` de una vez, verificando 12 propiedades en lugar de 14. Es decir, Deepseek logra compensar mediante su número de parámetros mucho mayor la dificultad de resolver prompts que le requieren realizar tareas más complejas

(aprender el invariante completo en lugar de propiedades más simples por separado).

Por otro lado, otro resultado relevante del estudio es el rol que cumple la etapa de verificación mediante Randoop, como filtro final y para reducir la carga sobre el programador. Los resultados indican que un 25 % de los invariantes generados fueron invalidados dinámicamente por Randoop, lo cual evidencia la presencia significativa de alucinaciones o afirmaciones incorrectas por parte de los modelos. Sin embargo, este porcentaje fue notablemente menor cuando se utilizaron prompts FSP, lo que refuerza la idea de que descomponer la especificación en propiedades simples no solo mejora la calidad de los invariantes generados, sino que también reduce la cantidad de resultados inválidos.

Asimismo, se observó que la dificultad de la inferencia se correlaciona con la complejidad algorítmica de las clases: estructuras simples y con invariantes de menor complejidad como `BinTree` o `LinkedList` resultaron más accesibles para la generación de invariantes usando LLMs. En cambio, estructuras con invariantes más complejos, como por ejemplo, estructuras que involucran propidades de balance de nodos, como `AvlTree` o `BinomialHeap`, presentaron mayores desafíos para los modelos.

Bibliografía

[1] Ian Sommerville. *Software Engineering*. 10th. Pearson, 2015. ISBN: 0133943038.

[2] Bertrand Meyer. *Object-oriented software construction*. en. Old Tappan, NJ: Prentice Hall, 1997.

[3] Barbara Liskov y John Guttag. *Program development in java*. en. Boston, MA: Addison Wesley, jun. de 2000.

[4] Chandrasekhar Boyapati, Sarfraz Khurshid y Darko Marinov. «Korat: automated testing based on Java predicates». En: *SIGSOFT Softw. Eng. Notes* 27.4 (jul. de 2002), págs. 123-133. ISSN: 0163-5948. DOI: 10.1145/566171.566191. URL: https://doi.org/10.1145/566171.566191.

[5] Pablo Abad et al. «Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT Solving». En: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, págs. 21-30. DOI: 10.1109/ICST.2013.46.

[6] Marcelo d'Amorim et al. «An Empirical Comparison of Automated Generation and Classification Techniques for Object-Oriented Unit Testing». En: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 2006, págs. 59-68. DOI: 10.1109/ASE.2006.13.

[7] Lisa Ling Liu, Bertrand Meyer y Bernd Schoeller. «Using contracts and Boolean queries to improve the quality of automatic test generation». En: *Proceedings of the 1st International Conference on Tests and Proofs*. TAP'07. Zurich, Switzerland: Springer-Verlag, 2007, págs. 114-130. ISBN: 3540737693.

[8] Carlos Pacheco y Michael D. Ernst. «Randoop: Feedback-directed Random Testing for Java». En: *OOPSLA 2007 Companion, Montreal, Canada*. ACM. Oct. de 2007.

[9] Gary T. Leavens et al. «How the design of JML accommodates both runtime assertion checking and formal verification». En: *Science of Computer Programming* 55.1 (2005). Formal Methods for Components and Objects: Pragmatic aspects and applications, págs. 185-208. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2004.05.015. URL: https://www.sciencedirect.com/science/article/pii/S0167642304001509.

[10] Jason Wei et al. *Finetuned Language Models Are Zero-Shot Learners*. 2022. arXiv: 2109.01652 [cs.CL]. URL: https://arxiv.org/abs/2109.01652.

[11] Xuemei Dong et al. *C3: Zero-shot Text-to-SQL with ChatGPT*. 2023. arXiv: 2307.07306 [cs.CL]. URL: https://arxiv.org/abs/2307.07306.

[12] Dawei Gao et al. *Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation*. 2023. arXiv: 2308.15363 [cs.DB]. URL: https://arxiv.org/abs/2308.15363.

[13] Zhishuai Li et al. *PET-SQL: A Prompt-Enhanced Two-Round Refinement of Text-to-SQL with Cross-consistency*. 2024. arXiv: 2403.09732 [cs.CL]. URL: https://arxiv.org/abs/2403.09732.

[14] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[15] Patrick Bareiß et al. *Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code*. 2022. arXiv: 2206.01335 [cs.SE]. URL: https://arxiv.org/abs/2206.01335.

[16] Laria Reynolds y Kyle McDonell. *Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm*. 2021. arXiv: 2102.07350 [cs.CL]. URL: https://arxiv.org/abs/2102.07350.

[17] Vasudev Vikram et al. *Can Large Language Models Write Good Property-Based Tests?* 2024. arXiv: 2307.04346 [cs.SE]. URL: https://arxiv.org/abs/2307.04346.

[18] Carlos Pacheco et al. «Feedback-directed random test generation». En: *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*. Minneapolis, MN, USA, mayo de 2007, págs. 75-84.

[19] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[20] Emily M. Bender et al. «On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?» En: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. FAccT '21. Virtual Event, Canada: Association for Computing Machinery, 2021, págs. 610-623. ISBN: 9781450383097. DOI: 10.1145/3442188.3445922. URL: https://doi.org/10.1145/3442188.3445922.

[21] Rishi Bommasani et al. *On the Opportunities and Risks of Foundation Models*. 2022. arXiv: 2108.07258 [cs.LG]. URL: https://arxiv.org/abs/2108.07258.

[22] Meta AI. *LLaMA Documentation: Overview*. 2024. URL: https://www.llama.com/docs/overview/ (visitado 04-04-2025).

[23] Paul Ammann y Jeff Offutt. *Introduction to Software Testing*. 2.ª ed. Cambridge, England: Cambridge University Press, dic. de 2016.

[24] Gordon Fraser y Andrea Arcuri. «EvoSuite: automatic test suite generation for object-oriented software». En: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: Association for Computing Machinery, 2011, págs. 416-419. ISBN: 9781450304436. DOI: 10.1145/2025113.2025179. URL: https://doi.org/10.1145/2025113.2025179.

[25] Paul Dan Marinescu y Cristian Cadar. «KATCH: high-coverage testing of software patches». En: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: Association for Computing Machinery, 2013, págs. 235-245. ISBN: 9781450322379. DOI: 10.1145/2491411.2491438. URL: https://doi.org/10.1145/2491411.2491438.

[26] Pablo Ponzio et al. «Automatically Identifying Sufficient Object Builders from Module APIs». En: *Fundamental Approaches to Software Engineering*. 2019. URL: https://api.semanticscholar.org/CorpusID:92998053.