DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

MENTAT
Una herramienta de validación dinámica para código Java

Tesis de Licenciatura en Ciencias de la Computación

Diciembre de 2015

Alumnos

María Antonia Bonfiglio
mbonfigl@dc.uba.ar
LU 523/94

Pablo Nussembaum
baunax@gmail.com
LU 211/96

Directores

Dr. Esteban Pavese
Dr. Hernán Czemerinski

Resumen

Por lo general, un componente de software en un lenguaje orientado a objetos tiene requerimientos que no son simples de especificar cuando se trata del orden en que sus métodos pueden ser invocados. Generalmente sólo se provee documentación para cada método sin especificar cuál es el protocolo esperado de uso y la interacción entre los métodos, lo que puede llevar a usos inválidos. La minería de especificaciones, si provee información apropiada al programador, puede ayudar a mitigar estos problemas. Esto ha incentivado varias investigaciones de técnicas de inferencias de modelos que infieren un modelo basado en Finite State Machines (FSMs).

Actualmente existen herramientas capaces de analizar el código fuente en forma estática generando modelos que ayudan a contrastar la implementación con su especificación. Herramientas como Contractor crean un Labeled Transition System (LTS) específico llamado Enabledness Preserving Abstraction (EPA) [dCBGU11] a partir de una especificación por contratos o código fuente haciendo análisis estático.

El presente trabajo se propone hacer minería de modelos utilizando una técnica de análisis dinámico, mediante la generación de modelos para aproximar el EPA, basándose en la información obtenida a partir de ejecuciones reales.

Para realizar dicho análisis se creó Mentat, una herramienta que construye dichos modelos a partir de clases Java con anotaciones para especificar su invariante y las precondiciones de sus métodos.

Abstract

Generally, a software component in an object-oriented language has requirements that are not simple to specify, for instance, the order in which its methods can be invoked. Usually only documentation for each method is provided without specifying what the expected use protocol and the interaction between methods, which can lead to invalid usages. If the programmer provides proper information, mining specification can help mitigate these problems. This has encouraged several researches about inference models techniques that builds a model based on Finite State Machines(FSM).

Currently there are tools that perform static analysis on source code that help to contrast an implementation with its specification. Tools such as Contractor builds an specific Labeled Transition System (LTS), known as Enabledness Preserving Abstraction (EPA) [dCBGU11], from a contract specification or performing an static analysis directly from source code.

The current work presents a dynamic analysis technique to perform mining model inference of classes by generating models to approximate the EPA, based on information obtained during actual executions.

To this end Mentat was created, it's a tool that builds models from Java classes with annotations to specify the invariant and the preconditions of his methods.

Agradecimientos

Queremos agradecer primero que nada a nuestras familias por haber estado siempre.

Queremos agradecernos mutuamente, porque después de haber pasado unos cuantos años de haber terminado de cursar, durante los últimos tiempos nos acompañamos y nos fuimos alentando alternativamente para llegar hasta acá y felizmente lograr cerrar esta etapa.

Por supuesto a Hernán y Esteban, por invitarnos y permitirnos hacer este trabajo, acompañarnos, ayudarnos y tenernos paciencia cuando perdíamos ritmo en la producción.

También a los jurados Diego y Fernando, por haber leído la tesis y por haber aportado con sus comentarios.

A lo largo de los años de cursada, hicimos muchos amigos, unos cuantos en común y varios con los que hoy nos seguimos viendo. Todos ellos fueron una parte importante de nuestra experiencia y de los que tenemos muy buenos recuerdos. Gracias a todos ellos.

Además tenemos muy presentes a los profesores que fuimos teniendo, algunos un verdadero honor realmente. Y a la educación pública en general, por permitirnos este lujo académico.

Y a Lari, nuestra hija, que todos los días nos hace muy felices y a quien amamos infinito.

Índice general

Capítulo 1

Introducción

Existe una gran variedad de artefactos de software que tienen requerimientos no triviales en relación al orden en que sus métodos o procedimientos deben ser invocados, es decir, su protocolo de uso. Tal es el caso de muchas APIs[1], que para poder ser utilizadas se debe seguir un cierto protocolo. Por ejemplo, para poder leer de un archivo siempre es necesario abrirlo en primera instancia. Un intento de lectura en un archivo que no fue aún abierto debería resultar en un error o una excepción, dependiendo de la implementación. En la práctica, las descripciones del comportamiento esperado de un software son incompletas e informales, e incluso en ciertos casos inexistentes, lo que dificulta la validación de las implementaciones y del código cliente que lo utiliza. Teniendo en cuenta la falta de documentación, se ha realizado una gran cantidad de trabajos de investigación para obtener técnicas de apoyo para la minería o la síntesis de typestates [2] [SY86] de las implementaciones de las APIs que luego se utilizan para verificar si el código cliente cumple o no con el protocolo de las implementaciones [AČMN05, DKM+10]. Estos enfoques, sin embargo, atacan sólo una parte del problema: asumen que el código del que se extrae el typestate es correcto, es decir, que el protocolo se implementa correctamente.

Esta tesis aborda el problema complementario. Es decir, validar si una implementación es correcta respecto del comportamiento esperado según la especificación, formal o no, de la clase. La validación del comportamiento de una implementación puede redundar en la identificación de errores en el código que introducen un comportamiento no deseado, en la detección de fallas en las especificaciones y en la mejora de la documentación disponible para dicho código.

La hipótesis de este trabajo es que una abstracción gráfica construida de forma automática a partir de una clase anotada con su invariante y precondiciones en los métodos, puede ser útil para (I) la validación contra requerimientos pobremente documentados, (II) la identificación de problemas en el código y en los requerimientos, y (III) mejorar la calidad de su documentación. Dado que la validación es una actividad que requiere la intervención humana, el nivel en que se abstrae una implementación es clave, ya que a diferencia de aquellas utilizadas para verificación automática, éstas deben ser fácilmente legibles e interpretables por humanos [Uri99].

[1] Application Programming Interface: Es el conjunto de funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción. Son usadas generalmente en las bibliotecas.

[2] *Typestates*: es el subconjunto de las operaciones permitidas en un contexto particular o una cadena particular cadena de invocaciones

Con el propósito de poder realizar la tarea de validación, en este trabajo se presenta una técnica dinámica para hacer minería de modelos. Esta técnica, a partir de ejecuciones sobre clases anotadas con precondiciones e invariantes, produce abstracciones llamadas Enabledness Preserving Abstractions (EPAs) [dCBGU11]. Concisamente, los EPAs cocientan el espacio de estados (potencialmente infinito) de una implementación en una cantidad finita de una clase, cada una de las cuales comprende aquellos estados concretos que habilitan exactamente el mismo conjunto de métodos. Se dirá que un modelo es equivalente al EPA si tiene exactamente los estados y las transiciones que representan todas las ejecuciones posibles, ni una más, ni una menos. Dado que el conjunto de todas las ejecuciones posibles es infinita, y que esta técnica se basa en ejecuciones para generar un modelo que se aproxime al EPA, la abstracción resultante puede no contener algunas transiciones que sí forman parte del EPA, por lo tanto, los modelos obtenidos con esta técnica son *sub-aproximaciones*.

La utilidad de los EPAs para tareas de validación ya ha sido estudiada en trabajos previos, como pueden ser [dCBGU11, dCBGU09, ZBdC$^+$11]. Todos estos trabajos se basan en análisis de contratos o análisis estático de código y las abstracciones construidas son *sobre-aproximaciones* del EPA para cada caso. El enfoque dinámico que presenta esta tesis, es complementario a los de [dCBGU11], [dCBGU09] y [ZBdC$^+$11] en diversos sentidos. Por un lado las abstracciones se van construyendo a medida que el código se va ejecutando, y sólo se agrega una transición en caso de que una ejecución de un método sea testigo de la existencia de dicha transición. Por esta razón puede ser que el modelo generado sea una *sub-aproximación* del EPA. Por otro lado, a diferencia de los mencionados trabajos que trabajan con lenguajes como C y C#, la herramienta implementada funciona sobre código Java, uno de los lenguajes más utilizados tanto en ámbitos científico-académicos como industriales .

Para validar la técnica se construyó la herramienta Mentat, que hace minería de EPAs sobre clases Java anotadas con precondiciones e invariantes. A grandes rasgos, Mentat realiza ejecuciones sobre el programa usado como input y chequea el estado alcanzado para ir construyendo las abstracciones. En el caso general, Mentat selecciona el próximo método a ejecutar mediante el azar. Esta forma de selección tiene el problema de que ciertas transiciones profundas no puedan ser "descubiertas", y se debe principalmente a que para ser transitadas se debe previamente ejecutar una secuencia compleja de métodos que difícilmente pueda ser producto de elecciones basadas en el azar. Para atacar este problema, la herramienta construida permite que, además de usar el azar, el usuario pueda definir funciones de selección que guíen a la herramienta para la selección de métodos. Asumiendo que el usuario tiene cierto conocimiento del dominio del programa input, el riesgo de que estados profundos no sean alcanzados disminuye.

En síntesis, en este trabajo se presenta una técnica para hacer minería de modelos que sirve para la validación de implementaciones de software que poseen un protocolo no trivial. Para ello se implementó una herramienta que toma como entrada programas en Java anotados y produce como salida modelos que son *sub-aproximaciones* de sus EPAs. La herramienta, a su vez, permite que el usuario defina funciones de selección que guíen las ejecuciones. Finalmente, se ha comprobado experimentalmente en los casos de estudio, que los modelos producidos resultan de ayuda para la validación del código y detección de errores.

1.1. Estructura de la tesis

El resto de la tesis está organizada del siguiente modo:

En el capítulo 2 se presentan las definiciones formales necesarias, y la motivación para realizar el presente trabajo. También se presenta un ejemplo ilustrativo mostrando a alto nivel el modo de uso de Mentat, junto con una descripción del camino recorrido y con las principales situaciones que fueron haciendo evolucionar las funcionalidades de la herramienta.

En el capítulo 4 se presenta y se explican los detalles de implementación de la herramienta Mentat. Primero se expone una explicación de alto nivel del algoritmo que va recolectando la información necesaria durante la ejecución para posteriormente generar el modelo que aproxima al EPA de la clase. Luego se pone el foco en los principales detalles de implementación, las anotaciones creadas, el lenguaje para la función de selección junto con la generación de parámetros, la generación de las trazas y del modelo resultado, y finalmente el modo de uso.

En el capítulo 5 se discuten algunos ejemplos representativos que fueron de utilidad para evaluar los usos y el rendimiento de la herramienta.

En el capítulo 6 se mencionan brevemente otros trabajos relacionados al presente.

Finalmente, en el capítulo 7 se presentan las conclusiones de esta tesis y se discuten algunas alternativas de trabajo a futuro.

Capítulo 2

Motivación

2.1. Marco teórico

Como ya se ha mencionado anteriormente, el objeto de análisis en este estudio es una clase anotada con invariantes y precondiciones. Por lo tanto es necesario como primer paso definir la interpretación semántica de una clase.

Se puede considerar a una clase C como una estructura de la forma $\langle M, F, R, inv, init \rangle$, donde:

- $M = \{m_1, \ldots, m_n\}$ es el conjunto finito de etiquetas de los métodos públicos.
- F es el conjunto de implementaciones de los métodos indexado por $m \in M$.
- R es el conjunto de precondiciones indexado por $m \in M$.
- inv es el invariante de clase.
- $init$ es la condición inicial establecida por los constructores de la clase.

Se considerará el ejemplo de la clase Bounded Stack, el cual se muestra en el programa 2.1. Dicha clase consta de los métodos `push` para apilar un nuevo elemento y `pop` para desapilar el elemento que se encuentra en el tope de la pila. Su correspondiente interpretación semántica es $Stack = \langle M, F, R, inv, init \rangle$, donde:

- $M = \{push, pop\}$, es el conjunto de métodos que constituye la interfaz pública de la implementación del Bounded Stack.
- $F = \{f_{push}, f_{pop}\}$, es el conjunto de funciones que corresponden a la interpretación semántica del código de los métodos `push` y `pop` respectivamente.
- $R = \{R_{push}, R_{pop}\}$, es el conjunto de predicados que son verdaderos para las instancias en las que puede ejecutarse el método.

  R_{push} es el predicado que es verdadero sólo para las instancias del Bounded Stack que no estén llenas.

  R_{pop} es el predicado que es verdadero sólo para las instancias del Bounded Stack que no estén vacías.
- inv es el predicado que es verdadero sólo para las instancias del Bounded Stack que cumplan $0 \leq count \leq limit$.
- $init$ es el predicado que define el estado inicial del Bounded Stack, donde vale $count = 0$.

A continuación se definirá el espacio de estados posibles el cual se caracteriza mediante un sistema potencialmente infinito y determinístico de transiciones etiquetadas, un Labelled Transition System (LTS). Se define un LTS como una estructura de la forma $\langle \Sigma, S, S_0, \delta \rangle$, donde Σ es el conjunto de etiquetas, S es el conjunto de estados, $S_0 \subseteq S$ es el conjunto de estados iniciales y $\delta : S \times \Sigma \to S$ es una función parcial de transición.

De esta forma, el espacio de estados posibles correspondiente a la interpretación semántica de una clase está compuesto por un estado por cada instancia válida (es decir, que cumple el invariante de clase) de los cuales sólo son iniciales los que además cumplen con la condición inicial. Luego, para cada estado s_i correspondiente a una instancia válida que cumple con la precondición de algún método m existe una transición de etiqueta m a otro estado s_j correspondiente a la instancia resultante luego de aplicar m en caso de ser válida. Es importante notar que el espacio de estados recién definido sólo tiene en cuenta las instancias que cumplen con el invariante de clase. Para una definición formal más detallada ver [dCBGU11].

Una vez que se tiene definido el espacio de estados posibles se debe establecer un nivel de abstracción adecuado para obtener una representación finita del mismo, la cual se pueda generar y manipular. La experiencia indica que agrupar los estados concretos en los cuales se encuentran habilitados el mismo conjunto de métodos es un nivel de abstracción que provee una buena relación entre tamaño y precisión [SY86].

Por lo tanto es necesario formalizar esta noción de equivalencia de instancias. Dada una clase C y dos instancias $c_1, c_2 \in C$, vale que c_1 y c_2 son equivalentes respecto de los métodos que habilitan, es decir *enabledness equivalent* (lo notaremos $c_1 \equiv_e c_2$), sii para cada método $m \in M$ vale $R_m(c_1) \Leftrightarrow R_m(c_2)$.

Dado el LTS del espacio de estados posibles correspondiente a la interpretación semántica de una clase se define un tipo de abstracción denominada Enabledness-Preserving Abstraction (EPA) como una máquina de estados finita no determinística que agrupa las instancias de la clase según los métodos que se encuentran habilitados. Dicha abstracción es capaz de simular cualquier traza del LTS original. Nuevamente remitirse a [dCBGU11] para una definición formal más detallada.

En otras palabras, el conjunto infinito de instancias de una clase particionado mediante la noción de equivalencia $\equiv_e$ antes definida, resulta en un conjunto finito de estados abstractos tales que cada uno de ellos corresponde a un grupo distinto de métodos habilitados. Es decir, cada estado abstracto agrupa todas las instancias que comparten el mismo conjunto de métodos habilitados y pueden ser especificado con un *predicado característico del estado.*

Dicho invariante de estado hace posible la construcción de este tipo de abstracciones, debido a que si bien desde un punto de vista teórico se puede obtener el EPA a partir del LTS utilizando el concepto de equivalencia de instancias descripto en el párrafo anterior, en la práctica esto no es posible dado que el LTS es potencialmente infinito. Por lo tanto es necesario recurrir a algún mecanismo que permita generar EPAs directamente de la ejecución de una implementación existente sin tener que considerar previamente su correspondiente espacio de estados concreto.

Dada una clase $C = \langle M, F, R, inv, init \rangle$ se define el invariante de un estado abstracto dado por un conjunto de métodos $ms \subseteq M$ como el predicado $inv_{ms} : C \to \{true, false\}$,

$$inv_{ms}(c) \overset{def}{\Leftrightarrow} inv(c) \ \wedge \bigwedge_{m \in ms} R_m(c) \ \wedge \bigwedge_{m \notin ms} \neg R_m(c)$$

De esta definición se desprende que un estado abstracto ms es válido si y solo si $\exists c \in C.\ inv_{ms}(c)$. Por lo tanto, un estado abstracto es simplemente un conjunto de métodos, que en caso de ser válido, existe una instancia $c \in C$ en la que se encuentran habilitados. Además tiene que cumplirse que ningún otro método esté habilitado, es decir, todos los otros métodos que no pertenecen a este conjunto se encuentran deshabilitados en esa misma instancia c.

Volviendo al ejemplo del Bounded Stack, se pueden calcular los invariantes de sus estados abstractos de la siguiente forma:

- $inv_{\varnothing}(c) \Leftrightarrow inv(c) \ \wedge \ \neg R_{push}(c) \ \wedge \ \neg R_{pop}(c)$
  $\Leftrightarrow 0 \leq count(c) \leq limit(c) \ \wedge \ count(c) \geq limit(c) \ \wedge \ count(c) \leq 0$
  $\Leftrightarrow limit(c) = 0$
  $\Leftrightarrow false$

- $inv_{\{push\}}(c) \Leftrightarrow inv(c) \ \wedge \ R_{push}(c) \ \wedge \ \neg R_{pop}(c)$
  $\Leftrightarrow 0 \leq count(c) \leq limit(c) \ \wedge \ count(c) < limit(c) \ \wedge \ count(c) \leq 0$
  $\Leftrightarrow count(c) = 0$

- $inv_{\{pop\}}(c) \Leftrightarrow inv(c) \ \wedge \ \neg R_{push}(c) \ \wedge \ R_{pop}(c)$
  $\Leftrightarrow 0 \leq count(c) \leq limit(c) \ \wedge \ count(c) \geq limit(c) \ \wedge \ count(c) > 0$
  $\Leftrightarrow count(c) = limit(c)$

- $inv_{\left\{\substack{push \\ pop}\right\}}(c) \Leftrightarrow inv(c) \ \wedge \ R_{push}(c) \ \wedge \ R_{pop}(c)$
  $\Leftrightarrow 0 \leq count(c) \leq limit(c) \ \wedge \ count(c) < limit(c) \ \wedge \ count(c) > 0$
  $\Leftrightarrow 0 < count(c) < limit(c)$

Notar que en éste ejemplo, según lo explicado anteriormente, el estado abstracto $\varnothing$ no es válido.

Finalmente es posible construir el EPA cuyos estados son los estados abstractos recién caracterizados y cuyas transiciones conectan dos de ellos, ms y ms', con la etiqueta m si existe una instancia c, la cual evoluciona en c' luego de la aplicación del método m; donde c satisface el invariante de ms y la precondición de m, y c' satisface el invariante de ms'.

Formalmente se define un EPA como una estructura $\langle \Sigma, S, S_0, \delta \rangle$ a partir de una clase dada $C = \langle M, F, R, inv, init \rangle$ de la siguiente manera:

1. $\Sigma = M$
2. $S = 2^M$
3. $S_0 = \{ms \in S \mid \exists c \in C.\ inv_{ms}(c) \wedge init(c)\}$
4. Dado $\delta : S \times \Sigma \to S$, para todo $ms \in S$ y $m \in \Sigma$,
   4.1. si $m \notin ms$ entonces $\delta(ms, m) = \varnothing$,
   4.2. si no $\delta(ms, m) = \{ns \in S \mid \exists c \in C.\ inv_{ms}(c) \wedge inv_{ns}(f_m(c))\}$

Notar que el punto 2 define a S como el conjunto de partes del conjunto finito M, con lo cual el EPA caracterizado resulta ser un LTS finito. Además, dada la definición de la función de transición δ, cuyo codominio es el conjunto de partes de S, el LTS resultante puede ser no determinístico.

2.2. Motivación

Existen herramientas que permiten la generación de modelos para aproximar el EPA que usan análisis estático de código con las cuales se obtienen resultados que *sobre-aproximan* el EPA, generando transiciones espurias [dCBGU10]. La generación de modelos basados en las ejecuciones de un programa asegura que todos los estados que se generan son alcanzables, pero por el contrario, se *sub-aproxima* el EPA dado que no puede asegurarse que se cubran todas las ejecuciones posibles.

La motivación de este trabajo es crear y poner a prueba el desempeño de una herramienta que genere modelos que aproximen el EPA de manera dinámica, es decir, basándose en ejecuciones del programa y que funcione como complemento de las herramientas existentes que hacen análisis estático. A su vez, se propone proveer una herramienta de aproximación a EPAs que pueda utilizarse sobre código Java, ya que las existentes, además de hacer análisis estático lo hacen sobre código C y C#.

La herramienta Mentat lleva a cabo la generación de la abstracción, ejecutando una cantidad de veces y en algún orden los métodos de la clase. En cada iteración, la herramienta selecciona un método a ejecutar para ir construyendo la traza de ejecución que se utilizará para aproximar el EPA. Para minimizar diferencias entre el EPA y el modelo que genera Mentat la herramienta provee mecanismos para que cuando sea necesario se *guíen* las ejecuciones y así poder asegurar coberturas razonables de las posibles ejecuciones. Esta funcionalidad es también muy útil para contrastar resultados obtenidos con Contractor, ya que nos permite dirigir la ejecución para intentar verificar si una transición es alcanzable en la implementación real. Más adelante se entrará en detalle sobre las herramientas que provee Mentat para guiar la ejecución, ya que es una característica importante a tener en cuenta.

2.3. Exploración del problema

Con el fin de mostrar las principales decisiones que se fueron tomando durante el diseño de la herramienta, se utilizará un ejemplo sencillo para ir introduciendo Mentat, donde también ya se pueden ir viendo la utilidad y los beneficios de la herramienta. Se mostrarán algunas anotaciones y nociones de Mentat en pseudo-código, sin hacer foco en los detalles técnicos de la sintaxis de Java y del lenguaje de especificación de invariantes y precondiciones.

Se tiene un Bounded Stack de números enteros con capacidad de 5 elementos, y una traza de ejecución generada por 100 invocaciones a métodos seleccionados al azar. Dicha traza es la que se utilizará para generar el modelo.

```java
@ClassDefinition(invariant = ( 5 >= stack.size() ))
public class ArrayStack implements Stack<Integer> {
    private List<Integer> stack = new ArrayList<>();

    @Pre( 5 > stack.size() )
    public void push(Integer n) {
        stack.add(n);
    }

    @Pre( 0 < stack.size() )
    public Integer pop() {
        return stack.remove(stack.size() - 1);
    }
}
```

Programa 2.1: Bounded Stack de enteros

El programa 2.1 muestra la implementación del Bounded Stack, y los métodos expuestos y anotados con sus correspondientes precondiciones son:

- `push`, agrega un elemento al ArrayStack.
- `pop`, saca un elemento del ArrayStack.

La clase está anotada con un invariante que especifica que la cantidad de elementos del Bounded Stack debe ser menor o igual al tamaño configurado. Los métodos están anotados con precondiciones que verifican el estado del Bounded Stack antes de su ejecución. Para ejecutar `pop()` en el Bounded Stack tiene que haber elementos, y para ejecutar `push()`, el Bounded Stack no tiene que estar lleno.

En la figura 2.1 puede observarse el modelo generado por MENTAT, en el cual están cubiertos todos los casos posibles en un Bounded Stack, es decir, coincide con el EPA. Todas las veces que se experimentó este caso con esta configuración (tamaño del Bounded Stack y cantidad de iteraciones), el modelo generado coincidió con el EPA, pero no se debe olvidar que MENTAT no siempre garantiza que el modelo generado sea el EPA de la clase.

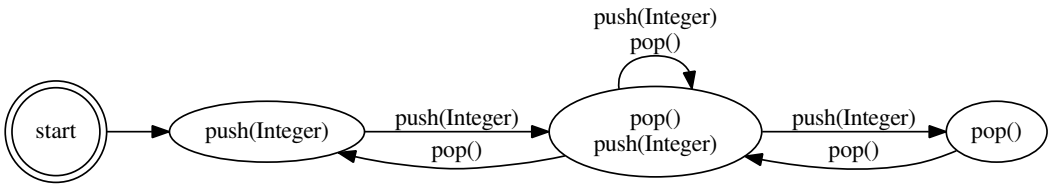

Figura 2.1: Abstracción correspondiente al Bounded Stack de tamaño 5, en una ejecución con 100 invocaciones aleatorias a métodos

Durante la construcción de MENTAT se fueron realizando experimentos adicionales y se encontraron algunas situaciones interesantes que sirvieron para delinear las características de la herramienta y darle más poder.

Como primera situación interesante, se notó que con un Bounded Stack de tamaño 20, también con 100 invocaciones a métodos, el modelo obtenido era diferente. En dicho

modelo puede observarse a primera vista, que nunca es alcanzado el estado donde sólo está habilitado el método `pop()`, en otras palabras, que nunca se logra llenar el Stack. A medida que crece el tamaño del Bounded Stack se hace más improbable que se termine de llenar, y así nunca es alcanzado el estado donde sólo está habilitado el método `pop()`, ya que es necesario que haya una sub-traza donde la diferencia de la cantidad de veces que se llame a `push()` con la cantidad de veces que se llame a `pop()`, sea igual al tamaño del Stack. El modelo obtenido puede verse en la figura 2.2.

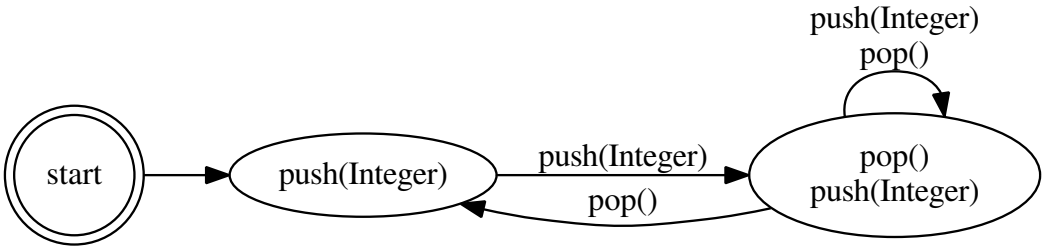

Figura 2.2: Abstracción correspondiente al Bounded Stack de tamaño 20, en una ejecución con 100 invocaciones aleatorias

Dado esto, se buscaron mecanismos para tratar de evitar este tipo de situaciones y se decidió dar la opción al programador de crear una *función de selección* para permitir guiar en cada paso la selección del método durante la ejecución.

Con este objetivo se creó un lenguaje para permitir al usuario crear fácilmente dichas funciones. Dicho lenguaje permite definir una lista de condiciones para seleccionar un método dado el estado actual de la instancia en ejecución.

```
while (stack.size() < limit) execute "push"
while (stack.size() = limit) execute "pop"
```

Listado 2.1: Pseudocódigo de un ejemplo de función de selección.

En esta función de selección 2.1 puede observarse que se ejecuta `push()` hasta que se llegue al límite y luego se ejecuta `pop()` y `push()` alternativamente. La herramienta tiene 2 maneras de generar parámetros, uno usando expresiones `Clojure` y otro utilizando el Solver `Z3`. Más adelante, en la subsección 4.1.3, se explicarán en detalle dichos mecanismos.

Al ejecutar MENTAT con estas condiciones el modelo obtenido puede verse en la figura 2.3. Como puede observarse en este caso nos aseguramos que se complete la pila y se alcanza el estado en el que únicamente es posible hacer `pop()`, pero a la vez se pierde la transición que representa el momento en que la pila se vacía, es decir, la transición que al ejecutar `pop()` deja al Bounded Stack en el estado que tiene como único método habilitado a `push()`.

Dado el poder expresivo del lenguaje para definir las funciones de selección, podría pensarse en definiciones más complejas, pero el objetivo de la función de selección es que guíe la ejecución a ciertos casos especiales sin perder completamente el factor aleatorio como se vio en la figura 2.2.

Para lo cual se modificó la herramienta para que genere el modelo basándose en la unión de las distintas ejecuciones que pueden haber sido generadas utilizado distintas funciones de selección junto con varias ejecuciones aleatorias.

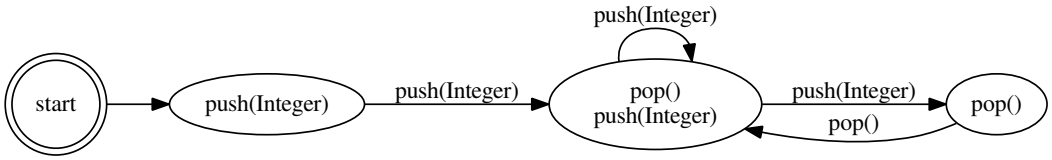

Figura 2.3: Abstracción correspondiente al Bounded Stack de tamaño 20, en una ejecución con 100 invocaciones usando la función de selección ejemplo de la función de selección 2.1

Para continuar con el ejemplo anterior, se hizo la prueba con *2* corridas, cada una con *100* invocaciones, y se definió que la primera corrida elija los métodos aleatoriamente y la segunda usando la función de selección del ejemplo antes expuesto. El modelo obtenido puede verse en la figura 2.4

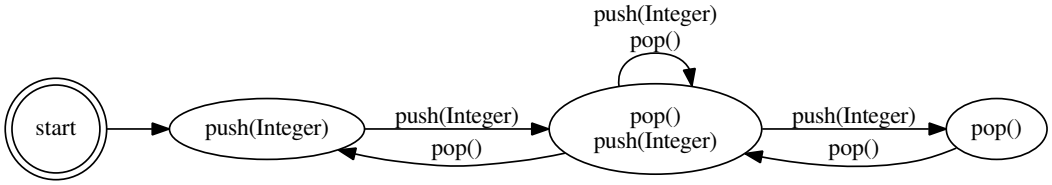

Figura 2.4: Abstracción correspondiente al Bounded Stack de tamaño 20, uniendo los modelos obtenidos en una ejecución con 100 invocaciones aleatorias a métodos, con una ejecución con 100 invocaciones usando la función de selección ejemplo de la función de selección 2.1

A modo de resumen, mediante este sencillo ejemplo se muestra cómo el enfoque adoptado puede asistir a los programadores en el reconocimiento de problemas al momento de utilizar una API en su código fuente. También da una primera impresión de los beneficios que brindan este tipo de abstracciones para validar especificaciones, especialmente las basadas en modelos que cumplen con la propiedad denominada *enabledness preserving*.

Capítulo 3

Definición formal de la técnica

La técnica se basa en ir generando una traza de ejecución para finalmente generar un modelo que aproxime al EPA.

Se comienza evaluando el invariante para verificar que el estado de la instancia actual sea válido. Luego, en cada iteración, se van usando las precondiciones de los métodos y el estado actual de la instancia, para determinar el estado a partir del conjunto de métodos habilitados. De ese conjunto se elige un método, se generan sus parámetros y se ejecuta. El par con el estado con los métodos habilitados y el método ejecutado, se agrega a la traza. Una vez finalizado este ciclo, ya sea porque se alcanzó la cantidad de iteraciones configurada, ocurrió un error o dio timeout algún método, se genera el gráfico del modelo a partir de la traza generada.

En el siguiente gráfico puede observarse un poco más en detalle lo que se hace en cada paso de la iteración de la generación de una traza basándose en la clase anotada. En algunos pasos están especificadas las herramientas o técnicas que se usan para resolver ese paso, como ser `Clojure`, `Z3`, la función de selección y GraphViz. En el capítulo 4 se verá más en detalle.

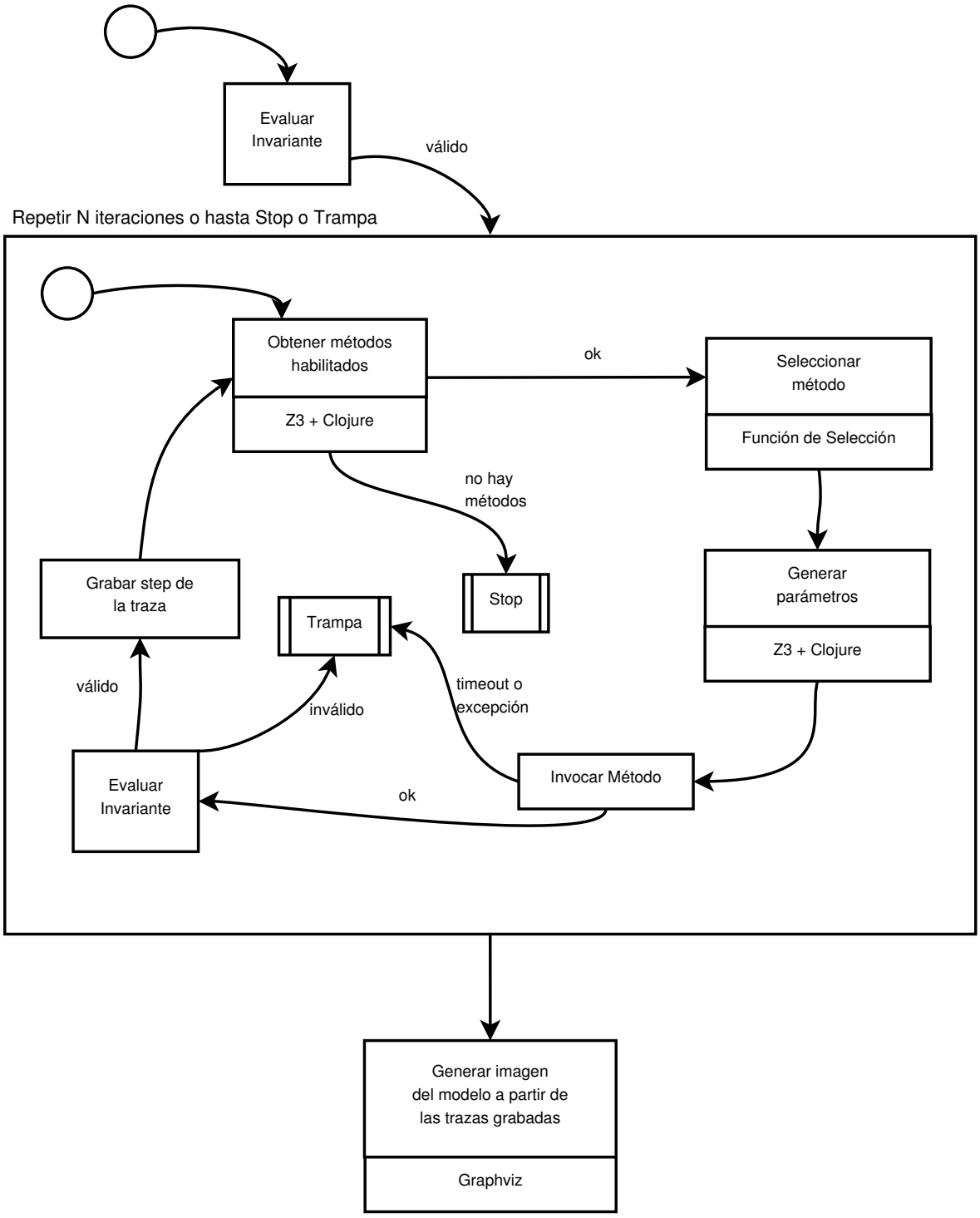

Figura 3.1: Flowchart de MENTAT

3.1. Algoritmo formal

A continuación se mostrará en pseudo código el algoritmo principal de MENTAT, que es el que construye una traza de ejecución. En este algoritmo se van definiendo los estados y las transiciones que se utilizarán posteriormente en la construcción del modelo que genera la herramienta para aproximar al EPA.

```
1: procedure trace-generator(C : ⟨M, F, R, inv, init⟩, n, fs) : T, ⟨Σ, S, S_0, δ⟩
2:     σ ← initialize(C)
3:     Σ ← M
4:     S ← ∅
5:     δ(ms, m) ← ∅      ∀ms, m
6:     S_0 ← ∅
7:     ms_0 ← {m ∈ M | inv(σ) ∧ init(σ) ∧ R_m(σ)}
8:     T ← empty list
9:     if (ms_0 ≠ ∅) then
10:        S_0 ← ∅
11:        i ← 0
12:        continue ← true
13:        ms ← ms_0
14:        while (continue ∧ ms ≠ ∅ ∧ i < n) do
15:            m ← fs(ms)
16:            T ← T + (ms, m)
17:            σ' ← excecute(σ, m)
18:            if (exception(σ') ∨ ¬inv(σ')) then
19:                ms' ← {'TRAP'}
20:                continue ← false
21:            else
22:                ms' ← {m ∈ M | inv(σ') ∧ R_m(σ')}
23:            end if
24:            δ(ms, m) ← (δ(ms, m) ∪ (ms, m) → ms')
25:            S ← S ∪ ms'
26:            σ ← σ'
27:            ms ← ms'
28:            i ← i + 1
29:        end while
30:    end if
31: end procedure
```

Algoritmo 3.1: Construcción de una traza de ejecución en MENTAT

El algoritmo 3.1, recibe como parámetros la clase C sobre la que quiere hacerse la ejecución, n que indica la cantidad de ejecuciones que se realizarán para construir el modelo, y fs que es la función de selección que se utilizará para seleccionar el método a ejecutar en cada iteración. El algoritmo retorna T, que representa la traza de ejecución sobre una instancia de C, con n ejecuciones y con una función de selección fs.

Como es un algoritmo que aproxima EPAs basándose en ejecuciones, puede verse que el primer paso del algoritmo consta de instanciar la clase C en σ que es efectivamente la instancia que se va a utilizar en el resto del algoritmo. Luego se inicializan el resto de las variables y se define el estado ms_0 como el conjunto de métodos en la instancia σ que cumplen con el invariante, las condiciones iniciales y las precondiciones sobre ellos. Este estado ms_0 será el único elemento en el conjunto de estados iniciales S_0, ya que que al estar basándose en la ejecución de una instancia de C, no se tendrán otros estados iniciales que puedan surgir de dicha instancia.

El algoritmo termina cuando ocurre alguna excepción durante la invocación a un método, o no hay ningún método habilitado para ser ejecutado ($ms = \varnothing$), o falla la evaluación del invariante después de invocar un método, o se alcanzaron las n ejecuciones que se establecieron. En cada iteración se determina el método m a ejecutar, utilizando la función de selección fs sobre el conjunto de métodos disponibles ms (son los métodos que cumplen con el invariante y sus precondiciones). Con esa información se va generando la traza T, el conjunto de estados S y la función de transición δ.

Por lo expuesto en la sección 2.1, se sabe que dada una clase C con M métodos públicos, el EPA de dicha clase tiene como máximo S estados, donde $S = 2^M$. En cada paso del algoritmo 3.1 se evalúan el invariante y los métodos habilitados en esa instancia de C. Con esa información se selecciona el método a ejecutar y se agrega a la traza el par formado por un estado s, y por el método utilizado para llegar a s, donde $s \in S$ y está definido por los métodos habilitados. De esta manera puede concluirse por construcción que el modelo generado es una *sub-aproximación* del EPA, ya que todos los estados y transiciones que el algoritmo va incluyendo, son estados existentes y transiciones válidas en el EPA.

Capítulo 4

Implementación

Para desarrollar MENTAT se utilizó `Clojure` [Hic08] un dialecto de `LISP` con la particularidad de que está construido sobre la `Java Virtual Machine`. Se eligió este lenguaje dinámico porque provee muchas facilidades para la generación dinámica de código, lo que permitió una ágil y compacta implementación de la herramienta.

Para la validación de precondiciones y la generación de parámetros válidos en cada invocación, se utilizó el `SMT solver Z3` [DMB08]. A partir de la subsección 4.1.2 se explicará en más detalle la interacción entre MENTAT y `Z3`.

Para generar el gráfico del modelo obtenido, se utilizó GraphViz [EGK+02] que es una herramienta estándar para la visualización de grafos.

4.1. Lenguaje de precondiciones e invariante

Las precondiciones e invariantes se definen con una expresión `Clojure`. Dicha expresión es compilada a una función donde las variables de instancia de la clase son convertidas a parámetros de la función. Aunque las precondiciones y los invariantes comparten la sintaxis por la cual son definidos, su evaluación es muy distinta.

Los invariantes son evaluados como funciones comunes en `Clojure` y se los considera válidos si y sólo si evalúan a un valor verdadero.

Las precondiciones de los métodos se traducen a un script en `Z3` que se construye con la unión del estado de la instancia y la precondición del método a analizar. Esto se verá en detalle en la sección subsección 4.1.2

Para evaluar una precondición se traducen las variables de instancia en su estado actual a comandos `Z3` y la precondición es válida si y sólo si `Z3` puede encontrar un modelo que satisfaga todo el conjunto de `asserts`. Además para facilitar la escritura de las precondiciones se provee la función `eval` que permite evaluar un expresión `Clojure` antes de realizar la traducción a `Z3` (véase por ejemplo su uso en la precondición del método `push` del ArrayStack 2.1, donde `eval` es usada para obtener la cantidad actual de elementos en el Stack).

4.1.1. Anotaciones de Mentat

Las precondiciones y el invariante de la clase son informados a MENTAT a través de dos anotaciones. La anotación `@ClassDefinition`, que como puede verse en el programa 4.1 tiene dos elementos. Uno es el elemento `invariant` para definir el invariante de la clase que tiene como valor por omisión `"true"`. Otro es el elemento `builder` que es una expresión `Clojure` que tiene que evaluar a una instancia válida de la clase que se esté estudiando.

```
public @interface ClassDefinition {
    /** Define el invariante */
    String invariant() default "true";

    /** Define un función para construir las instancias */
    String builder();
}
```

Programa 4.1: Anotación de clase para definir el invariante y la función de construcción.

La anotación `@Pre` se muetra en el programa 4.2. Dicha anotación se utiliza para definir la precondición de un método, y tiene cuatro elementos. El más importante es `value` donde se define la precondición propiamente dicha. En `data` se puede definir una función para generar parámetros para invocar al método (se explica en más detalle en la subsección 4.1.3). El elemento `name` se utiliza para proveer un alias para luego ser utilizado en la función de selección en el caso que la clase tenga método sobrecargados. Y como ayuda al programar se puede utilizar el elemento `enabled` para que MENTAT considere en el análisis la inclusión o no del método.

```
public @interface Pre {
    /** Define la expresión de la precondición */
    String value() default "true";

    /** Define una función para la geneación de parámetros */
    String data() default "";

    /** Define si el método va a ser incluído en el análisis */
    boolean enabled() default true;

    /** Define un alias para el nombre del método */
    String name() default "";
}
```

Programa 4.2: Anotación definir la precondición de un método.

4.1.2. Traducción del estado una clase a Z3

Z3 es un SMT-Solver desarrolado por Microsoft Research que implementa y extiende el estándar SMT-LIB versión 2.0 [BST10] que además provee integración con Java y otros lenguajes de programación.

En cada iteración del algoritmo se necesita traducir el estado de la instancia que se está ejecutando a un script en Z3 para poder evaluar la precondición de cada método de dicha instancia. Básicamente un script en Z3 es lista de definiciones de variables más una lista de afirmaciones o `asserts` sobre ellas. Por ejemplo un `assert` puede afirmar que una variable tiene valor dado o que la variable cumple alguna condición en relación con otras variables. Finalmente se le puede preguntar a Z3 si es posible satisfacer todas las afirmaciones dándole un valor cada variable libre. Para más información sobre el lenguaje, ver `http://rise4fun.com/z3/tutorial/guide`.

MENTAT soporta traducir de Java a Z3 tipos básicos como booleanos, tipos numéricos (tanto los primitivos y sus clases wrapper), `BigInteger`, `BigDecimal`, `AtomicBoolean`, `AtomicInteger`, `AtomicLong`. Para tipos complejos como arreglos y colecciones de los tipos antes mencionados, MENTAT hace todo lo posible para deducir el tipo genérico de la colección pero cuando esto no es posible, presume que es una colección de `Integer`.

Para cada tipo básico se genera una declaración de constante más un `assert` con su valor actual. En el programa 4.3 se muestra, a modo de ejemplo, cómo es traducida una instancia de una clase Java con sólo tipos básicos a su correspondiente script en Z3.

Ejemplo en Java:

```java
public class NumbersToZ3 {
  private int sint = 10;
  private Boolean bTrue = true;
  private boolean bFalse = false;
  private BigInteger bigInteger = BigInteger.TEN;
  private Float everything = 41.9999999999999F;
  ...
}
```

Es traducido al siguiente script en Z3:

```
(declare-const sint Int)
(assert (= sint 10))

(declare-const bTrue Bool)
(assert (= bTrue true))

(declare-const bFalse Bool)
(assert (= bFalse false))

(declare-const bigInteger Int)
(assert (= bigInteger 10))

(declare-const everything Real)
(assert (= everything 41.9999999999999))
```

Programa 4.3: Ejemplo NumbersToZ3 junto con el script generado en Z3.

Para los arreglos nativos o implementaciones de `java.util.Collection` se declara una constante de tipo `Array` en `Z3` con índice entero y elementos del tipo genérico de la colección. Esto se hace siempre y cuando sea posible deducirlo en tiempo de ejecución, ya que por el efecto de la borradura[1] no siempre es posible obtenerlo. En caso de no ser posible, como ya se dijo, se lo presume de tipo entero. En el programa 4.4 se muestra cómo es traducida una instancia de una clase con algunos *Arrays* a su correspondiente script en `Z3`:

[1] `https://docs.oracle.com/javase/tutorial/java/generics/erasure.html`

Ejemplo en Java:

```
public class ArraysToZ3 {
  private List<Boolean> listBool =
    new ArrayList<>(
      asList(new Boolean[] { true, true, false, true }));

  private boolean[] arrBool = { true , false };

  private List<Long> listLong =
    new ArrayList<>(
      asList(new Long[] { 1L, 2L, 3L, 4L }));

  private Set<Integer> setInteger =
    new LinkedHashSet<>(
      asList(new Integer[] { 1, 2 }));
  ...
}
```

Es traducido al siguiente script en Z3:

```
(declare-const listBool (Array Int Bool))
(= true (select listBool 0))
(= true (select listBool 1))
(= false (select listBool 2))
(= true (select listBool 3))

(declare-const arrBool (Array Int Bool))
(= true (select arrBool 0))
(= false (select arrBool 1))

(declare-const listLong (Array Int Int))
(= 1 (select listLong 0))
(= 2 (select listLong 1))
(= 3 (select listLong 2))
(= 4 (select listLong 3))

(declare-const setInteger (Array Int Int))
(= 1 (select setInteger 0))
(= 2 (select setInteger 1))
```

Programa 4.4: Ejemplo ArraysToZ3 junto con el script en Z3 generado.

4.1.3. Generación de parámetros

A diferencia de cómo se evalúa el invariante, las precondiciones de los métodos se traducen a un solo comando `assert` en un script en Z3. Además las expresiones que definen las precondiciones pueden tener variables libres que representan los parámetros del método anotado. Éstas variables son de la forma p_n donde n es la posición en la

lista de parámetros.

En cada iteración del algoritmo es necesario generar una lista de parámetros válidos que cumplan con la precondición del método seleccionado. MENTAT provee dos mecanismos para generar dicha lista. Una forma es mediante una función en `Clojure` que para una instancia devuelva un mapa con un valor para cada parámetro. Esta función de generación de parámetros se define en la anotación `@Pre.data` como se puede ver en el programa 4.5 que genera un valor entero con el tamaño actual del Stack.

```
public class ArrayStack implements Stack<Integer> {
  ...

  @Pre(value = "(> limit (eval (count stack)))",
    data = "{:p0 (count stack)}")
  public void push(Integer n) {
    ...
  }
  ...
}
```

Programa 4.5: Generación de parámetros utilizando una función.

La otra forma es dentro de la precondición dejar variables libres que luego `Z3` genere un modelo proveyendo un valor para ellas. En el programa 4.6 se puede observar el mismo método pero ahora `Z3` generará un valor mayor que 42.

```
public class ArrayStack implements Stack<Integer> {
  ...

  @Pre("(and (> limit (eval (count stack))) (> p0 42))")
  public void push(Integer n) {
    ...
  }
  ...
}
```

Programa 4.6: Generación de parámetros delegada a `Z3`.

Ambos métodos de generación se pueden utilizar en forma simultánea, permitiendo complejas condiciones utilizando `Z3`, y a su vez permitir generar valores de tipos no soportados por la traducción a `Z3` o que no dependen de la clase en estudio, como por ejemplo conexiones a base de datos.

4.2. Función de selección de métodos

Como ya se ha dicho, la herramienta MENTAT permite definir funciones de selección con el fin de guiar, en cada paso del proceso de generación de la traza, la selección de los métodos para poder asegurar que ciertos estados sean alcanzados. En esta sección

se describirá el lenguaje que provee MENTAT para generar funciones de selección de métodos. Una función de selección de métodos es una función que tiene dos parámetros, la instancia del objeto para el cual se está generando la traza junto con la lista de métodos habilitados en ese momento. Dicha función retorna como resultado uno de los métodos habilitados o informa que no pudo elegir ninguno.

MENTAT provee dos implementaciones de las funciones de selección. Una que es la elección al azar del método a ejecutar. Otra, que admite generar una función de selección a partir de un lenguaje que permite especificar predicados basándose en el estado la instancia para elegir el método a invocar.

4.2.1. Lenguaje de generación de funciones de selección

Para definir la función de selección, se genera un archivo de tipo `edn` (Extensible Data Notation[2]). El lenguaje que se está definiendo, consta de una lista de funciones que se ejecutan de forma secuencial, y una vez que se ejecutaron todos los elementos de la lista vuelve a comenzarse, es decir, se ejecutan de manera circular.

Hay 2 tipos de elementos que puede tener esta lista:

1. `#mentat/random`

   Esta directiva se usa para que se ejecute un método al azar y luego sigue con el próximo elemento de la lista. El método que se selecciona al azar, siempre está dentro del conjunto de métodos que en la instancia actual cumplen con el invariante y su precondición.

2. `#mentat/while ("methodName" (Clojure expression))`

   Esta directiva se usa para que mientras se cumpla la expresión, se ejecute el método que se designó en el parámetro, siempre y cuando en la instancia actual se cumpla el invariante y la precondición de dicho método. La expresión, es una expresión `Clojure` que genera una función que tiene como parámetro las variables de instancia de la clase.

Por ejemplo, en el caso del Bounded Stack, podríamos pensar en un ejemplo de función de selección que ejecute el método `push` hasta que falte un elemento para llenar la pila, que 3 veces elija métodos al azar, y por último, si a la pila le falta un elemento, fuerce la ejecución del método `pop`.

```
[#mentat/while ("push" (< (count stack) (dec limit)))
 #mentat/random
 #mentat/random
 #mentat/random
 #mentat/while ("pop" (= (count stack) (dec limit)))]
```

Programa 4.7: Ejemplo de función de selección

[2]Para más información ver `http://edn-format.org`

4.3. Evaluación de la precondiciones

Las precondiciones son evaluadas en cada iteración del algoritmo luego de ejecutar el método elegido. Para evaluarla se traduce a `Z3` la precondición, y se suma al script que representa el estado de la instancia, cuya traducción se mostró en la subsección 4.1.2. La precondición se transforma a un `assert` en `Z3`, reemplazando primero las variables de la precondición que hacen referencia a las variables de instancia de la clase por su valor actual, dejando libres las variables que representan parámetros. Luego se reemplazan los bloques *eval* por el valor de su evaluación. Finalmente el `assert` de la precondición se evalúa en conjunto con el script del estado de la instancia y se lo considera válido si `Z3` puede encontrar un modelo.

La lista de operaciones soportadas de `Z3` son:

Operadores booleanos	
Función	Descripción
`=`	Comparación por igualdad entre dos expresiones
`==>`	Implicación
`iff`	Si y sólo si
`<`	Comparación por menor
`>`	Comparación por mayor
`<=`	Comparación por menor o igual
`=>`	Comparación por mayor o igual
`and`	Conjunción entre dos expresiones
`or`	Disyunción entre dos expresiones
`not`	Negación de la expresión
`xor`	Disyunción exclusiva entre dos expresiones

Operadores matemáticos	
Función	Descripción
`+`	Operación de suma
`-`	Operación de resta
`*`	Operación de multiplicación
`/`	Operación de división
`mod`	Operación resto de la división

4.4. Construcción de la traza

Ahora que se ha mostrado cómo se traduce el estado de una instancia a `Z3` y cómo se evalúan las precondiciones e invariantes, se puede pasar a ver cómo se genera la traza a medida que se ejecuta la clase que se está estudiando. Una traza es un lista arbitrariamente larga de pares: método ejecutado y estado del EPA formado por los métodos cuyas precondiciones son válidas luego de haberlo ejecutado. Para generar la traza MENTAT primero crea una nueva instancia de la clase que se está estudiando, y valida el invariante para verificar que se puede comenzar a generar la traza. A continuación MENTAT construye una lista de ejecuciones de métodos. Cada vez que se necesita construir un nuevo elemento de la lista, se selecciona un nuevo método a ejecutar utilizando la función de selección provista. Al ejecutarlo, si el método no devuelve un valor en un tiempo razonable, o tira una excepción u ocurre una falla en la evaluación del invariante, se termina la lista, marcando que se llegó al estado

trampa ejecutando este último método. En caso contrario, se asume que el método fue evaluado exitosamente y se evalúan todas las precondiciones para definir a qué estado se ha llegado y retorna el nuevo par.

El primer elemento de una traza siempre tiene la forma $[\varnothing, E_1]$, donde E_1 es el estado inmediatamente después de construir la clase. El segundo elemento tiene la forma $[m_1, E_2]$, donde m_1 es el primer método ejecutado y E_2 es el estado al que se llega después de ejecutar m_1. Así se sigue sucesivamente hasta alcanzar el número de ejecuciones definidas por el usuario, o bien hasta que ocurra un error en la ejecución. Como ya se dijo, en el caso de que ocurra un error la ejecución se detiene, y el elemento que se agrega tiene la forma $[m_i, TRAP]$, donde m_i es el método que originó el error y $TRAP$ es el estado *trampa* en el que terminó la ejecución.

4.5. Construcción del modelo

La construcción del modelo que aproxima al EPA se basa en la o las trazas generadas durante la o las ejecuciones realizadas. Utilizando esta información, se va generando un *set* cuyos elementos son triplas de la forma $[E_1, m, E_2]$, donde E_1 es el estado origen, m es el método que se ejecutó y E_2 es el estado al que se llega luego de ejecutar m. Para generar este *set* se utilizan de a pares los elementos de las trazas para poder determinar los elementos de la tripla.

Mentat genera el gráfico del modelo construido utilizando GraphViz [EGK+02], que tiene como input para hacer el gráfico, triplas de la forma recién descripta. Siempre se agrega un estado inicial $START$ con una transición al estado E_1 y a partir de ahí se va armando el gráfico del modelo con los métodos como transiciones y los estados a los que se llega. En el caso de que el modelo se construya basándose en más de una ejecución, y el E_1 no sea el mismo en todos los casos, el gráfico del modelo tendrá una transición desde $START$ hasta cada uno de estos estados.

4.6. Uso de Mentat

En esta sección se explicará cómo utilizar Mentat asumiendo que el lector tiene conocimientos de `Clojure` *REPL* [3]. Se recomienda iniciar el REPL utilizando Leiningen [4].

El primer paso que se realizar es ingresar al *REPL* desde la consola. Si el sistema operativo es *GNU/Linux*, se debe también debe indicar la ubicación de las librerías de `Z3`.

```
1 $ LD_LIBRARY_PATH=./native/ lein repl
2 nREPL server started on port 48214 on host 127.0.0.1 - nrepl://127.0.0.1:48214
3 REPL-y 0.3.5, nREPL 0.2.6
4 Clojure 1.6.0
5 Java HotSpot(TM) 64-Bit Server VM 1.7.0_72-b14
6     Docs: (doc function-name-here)
7           (find-doc "part-of-name-here")
8   Source: (source function-name-here)
9  Javadoc: (javadoc java-object-or-class-here)
```

[3] Read-Eval-Print Loop

[4] Más información en `http://leiningen.org/`

```
10      Exit: Control+D or (exit) or (quit)
11   Results: Stored in vars *1, *2, *3, an exception in *e
12
13 user=>
```

Luego debe importarse la clase Java anotada.

```
1 user=> (import '(ar.com.maba.tesis.arrayList ArrayList$ListItr))
2 ar.com.maba.tesis.arrayList.ArrayList$ListItr
```

A continuación se crean dos trazas de ejecución utilizando la función de selección random.

```
1 user=> (def randomTrace1 (t/trace-gen ArrayList$ListItr t/random-sel))
2 user=> (def randomTrace2 (t/trace-gen ArrayList$ListItr t/random-sel))
```

Y Finalmente se genera el modelo

```
1 user=> (def g (g/build-dot-file [randomTrace1 randomTrace2]))
2 #'user/g
3 user=> (-> g d/dot (d/save! "example.png" {:format :png}))
4 nil
```

Capítulo 5

Experimentación

En este capítulo se describirán los aspectos involucrados en la validación de la herramienta MENTAT. Para ésto se presentarán algunos ejemplos que se utilizarán para analizar la efectividad de la misma. Mediante ejemplos concretos, se evaluarán principalmente los siguientes aspectos de la herramienta:

- ¿Permite descubrir errores de implementación?
- ¿Ayuda a identificar problemas en los requerimientos?

Para realizar el análisis se han seleccionado los siguientes ejemplos: `ListIterator`, la implementación de Oracle Java 7, `ArrayStack`, `Microwave` y `ATM` también usados en [Zop12].

Cabe aclarar que dichos programas no intentan dar solución a los problemas reales asociados a cada uno de ellos, sino que constituyen una versión reducida y simplificada de los mismos, con el único objetivo de ser claros y sencillos para facilitar su entendimiento y posterior análisis, evitando así el grado de complejidad que conlleva cada uno de ellos en la realidad. Por este mismo motivo elegimos ejemplos que modelan objetos de uso común y conocidos. De esta forma, se facilita su comprensión y de esta manera se hace foco en la utilización de la herramienta propiamente dicha, sin profundizar demasiado en los detalles del funcionamiento de cada uno de ellos. Todos los programas aquí presentados se encuentran disponibles para descargar desde la página web de GitHub[1].

La siguiente tabla comparativa resume información relevante sobre los ejemplos previamente mencionados. Para cada uno de ellos, la columna *Líneas* contiene la cantidad total de líneas de código[2], mientras que la columna *Métodos* contiene la cantidad total de métodos públicos. A su vez, la columna *Precondiciones* contiene la cantidad total de métodos públicos anotados para analizar, incluyendo la anotación del invariante de clase. Por ejemplo en el caso del `ListIterator` no se incluyeron los métodos públicos como `hasNext` ó `hasPrevious` que se pueden invocar en cualquier momento para evitar generar un modelo complejo de interpretar.

[1] Disponible en: `https://github.com/bauna/Mentat`.

[2] Contadas utilizando la herramienta `http://cloc.sourceforge.net/`

Nombre	Líneas	Métodos	Precondiciones
`ArrayStack`	36	2	3
`Microwave`	88	12	8
`ATM`	65	6	7
`ListIterator`	315	14	7

En las próximas secciones se profundizará en cada uno de los ejemplos por separado para realizar un breve análisis de los mismos. Se omitirá la sección correspondiente al ejemplo `ArrayStack` debido a que es equivalente al Bounded Stack que ya fue analizado en profundidad en los capítulos anteriores.

5.1. Microwave

Esta clase encapsula las operaciones básicas de un horno microondas. A continuación puede verse la clase, junto con su invariante y precondiciones para los métodos a analizar.

```
@ClassDefinition(
  builder = "(new ar.com.maba.tesis.microwave.MicrowaveImpl)",
  invariant = "(and (==> on (not doorOpened)) " +
                  "(==> on (and (> power 0) (> time 0))) " +
                  "(==> (not on) (= power 0)))")
public class MicrowaveImpl implements Microwave {

...

  @Pre(value = "(and (not doorOpened) (not on))",
       name="start")
  public void start(){...}

  @Pre(value = "(and (not doorOpened) (not on) (> p0 0))",
       name="start1")
  public void start(Integer time){...}

  @Pre(value = "(and (not doorOpened) (not on) (> p0 0) (> p1 0))",
       name="start2")
  public void start(Integer time, Integer power){...}

  @Pre("(and on (> time 0) (> power 0))")
  public void stop(){...}

  @Pre("(and on (> time 0) (> power 0))")
  public void pause(){...}

  @Pre("(not doorOpened)")
```

```
  public void openDoor(){...}

  @Pre("(doorOpened)")
  public void closeDoor(){...}

  public boolean isDoorOpened(){...}

  public long getPower(){...}

  public long getTime(){...}

  public boolean isOn(){...}
}
```

Invocando alguna de las versiones del método `start` se inicia la cocción, la cual, se puede finalizar invocando a `stop` o pausar invocando a `pause`. A su vez, por cuestiones de seguridad, la cocción también se debe detener cuando se abre la puerta y se puede volver a reanudar sólo si la puerta está cerrada. Dichas acciones son capturadas por los métodos `openDoor` y `closeDoor` respectivamente.

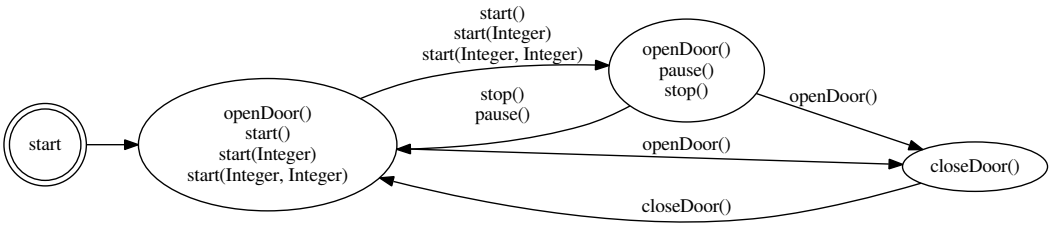

Figura 5.1: Abstracción correspondiente al controlador de un horno microondas básico

La figura 5.1 muestra la abstracción generada por MENTAT utilizando la función de selección aleatoria con cien invocaciones.

Frente a la presencia de errores, eventualmente podrían verse reflejados al observar el modelo generado por MENTAT. Mediante la inspección del modelo, puede identificarse qué método originó la transición al estado *TRAP*, y allí es donde hay que revisar su precondición y su implementación. Si por ejemplo, se omite en la precondición el chequeo de que la puerta esté cerrada para poder hacer alguno de los `start`, tendríamos un código como el que sigue

```
...

  @Pre(value = "(not on)",
       name="start")
  public void start(){...}

  @Pre(value = "(and (not on) (> p0 0))",
       name="start1")
  public void start(Integer time){...}

  @Pre(value = "(and (not on) (> p0 0) (> p1 0))",
```

```
        name="start2")
  public void start(Integer time, Integer power){...}

...
```

y el modelo obtenido en este caso es el siguiente:

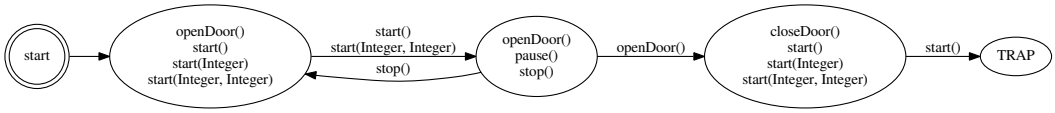

Figura 5.2: Modelo del microondas con las precondiciones modificadas

Aquí se ve fácilmente que al no chequear en las precondiciones que la puerta está cerrada para llamar a alguno de los métodos `start`, sucede que teniendo la puerta abierta permite llamarlo. Como en la implementación sí se está haciendo esta verificación, al intentar llamar a `start` con la puerta abierta se lanza una excepción. Esto se refleja en el modelo obtenido con una transición al estado *TRAP*.

5.2. ATM

Esta clase modela las operaciones principales de un cajero automático básico. A continuación puede verse la clase, junto con su invariante y precondiciones para los métodos a analizar.

```
@ClassDefinition(
    builder = "(new ar.com.maba.tesis.atm.AtmImpl)",
    invariant = "(or cardInside (not authenticated)) ")
public class AtmImpl implements Atm {

...

  @Pre("(and cardInside (not authenticated))")
  public void authenticate(){...}

  @Pre("(authenticated)")
  public void finish(){...}

  @Pre("(not cardInside)")
  public void insertCard(){...}

  @Pre("(and (not authenticated) cardInside)")
  public void removeCard(){...}

  @Pre("(authenticated)")
  public void operate(){...}

  @Pre("(authenticated)")
```

```
  public void printTicket(){...}
}
```

Cada vez que se inserta una tarjeta en el cajero invocando al método `insertCard`, cambia el estado interno de la instancia para permitir al usuario autenticarse. Luego el usuario puede comenzar a operar con el `ATM`, para lo cual en el ejemplo está representado por el método `operate`. Adicionalmente, mientras esté la tarjeta dentro del `ATM` puede imprimir un ticket y finalmente puede finalizar la sesión retirando la tarjeta invocando al método `removeCard`. La figura 5.3 muestra el modelo generado para el `ATM` utilizando MENTAT.

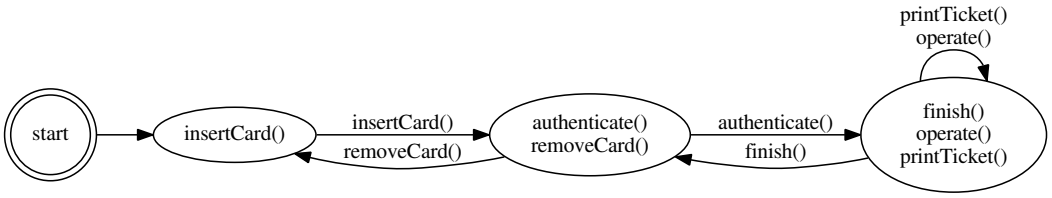

Figura 5.3: Modelo correspondiente al controlador de un cajero automático básico

En el ejemplo del `ATM` se modificará la implementación del método `printTicket` para mostrar como MENTAT encuentra la falla. Luego de hacer una corrida de MENTAT con la función de selección random vemos en la figura 5.4 que se llega al estado *TRAP* habiendo ejecutado `printTicket`.

```
...

  @Pre("(authenticated)")
  public void printTicket() {
    // correcta: checkAuthenticated();
    checkNotAuthenticated();
  }

...
```

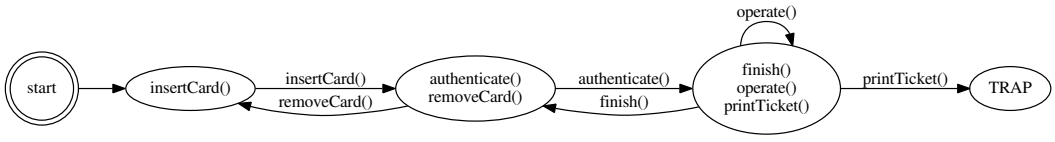

Figura 5.4: Modelo generado con la validación errónea de usuario.

5.3. Java ListIterator

En este ejemplo, a diferencia de los anteriores, se trabajó con `java.util.ListIterator` que es una interface previamente existente y de uso muy extendido en `Java`, y en particular se usó la implementación de

`java.util.ArrayList`. Para poder agregar las anotaciones se copió la implementación de `java.util.ArrayList` que contiene la implementación de `java.util.ListIterator`.

Como en `Java` no es posible en todos los casos obtener el tipo genérico de una instancia en particular, se agregaron dos métodos proxy, `addNumber` y `setNumber`, para evitar este problema. Como ya se aclaró previamente no se incluyen en el modelo de la figura 5.5 los métodos que no alteran el estado de la instancia.

A continuación puede verse la clase, junto con su invariante y precondiciones para los métodos a analizar.

```
@ClassDefinition(builder="(let [a (doto (ar.com.maba.tesis.arrayList.ArrayList.)
                            (.add 1) (.add 2) (.add 3))] (.listIterator a))",
                 invariant = "(and " +
                             "(or (= lastRet -1) (>= lastRet 0)) " +
                             "(and (<= 0 cursor) (<= cursor (.size this$0))))")
public class ListItr extends Itr implements ListIterator<E> {

...

  ListItr(int index){...}

  public boolean hasPrevious(){...}

  public int nextIndex(){...}

  @Pre("(> cursor 0)")
  public int previousIndex(){...}

  public boolean hasNext(){...}

  @Pre("(< cursor (eval(.size this$0)))")
  public E next(){...}

  @Pre("(>= lastRet 0)")
  public void remove(){...}

  @Pre("(> cursor 0)")
  public E previous(){...}

  @Pre("(and (>= lastRet 0) (> p0 10))")
  public void setNumber(Integer integer){...}

  public void set(E e){...}

  @Pre("(> p0 0)")
  public void addNumber(Integer integer){...}

  public void add(E e){...}
```

```
}
```

Esta implementación de `ListIterator` es una reimplementación de la clase real de `Java` pero fue reimplementada para poder aplicar las anotaciones necesarias tomando el código directamente del `Oracle JDK`.

En el caso del `ListIterator` es interesante ver como la herramienta se comporta al escribir una precondición errónea. Para lo cual se cambió la precondición del `next` para que este método esté habilitado cuando el iterador haya alcanzado el último elemento de la lista subyacente. Vemos en la figura 5.6 como ahora se llega al estado *TRAP* luego de ejecutar el método `next`.

```
@Pre("(<= cursor (eval(.size this$0)))")
// correcta: @Pre("(< cursor (eval (.size this$0)))")
public E next(){...}
```

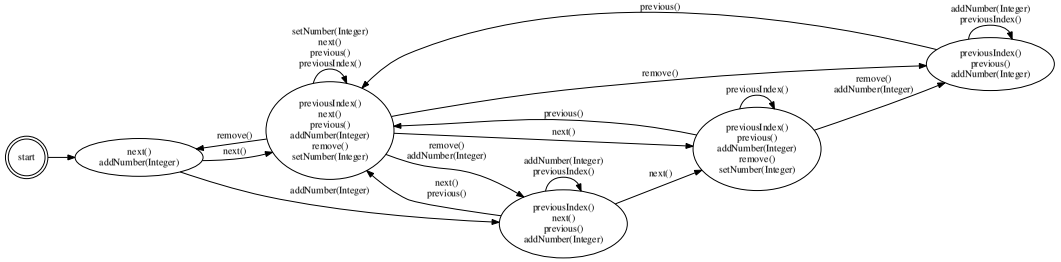

Figura 5.5: Modelo correspondiente al `java.util.ListIterator`

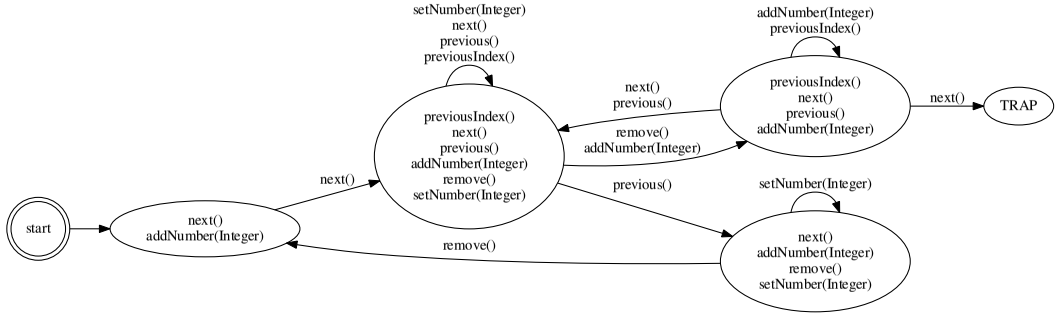

Figura 5.6: Modelo con la precondición del método `next` modificada para fallar.

Capítulo 6

Trabajo relacionado

En [dCBGU09] los modelos para aproximar el EPA se construyen para validar especificaciones basadas en contratos utilizando un demostrador de teoremas de primer orden, y si ni una prueba ni un contraejemplo se pueden encontrar para determinar si existe una transición o no, la herramienta la añade simplemente porque podría estar presente. De manera similar (aunque en este caso sí se validan implementaciones en C) en [dCBGU11] la abstracción se arma usando un software model checker, y cuando no se obtiene respuesta sobre la alcanzabilidad de algunas sentencias la herramienta también agrega transiciones. En [ZBdC+11] se muestra una idea similar a las presentadas en [dCBGU11] y [dCBGU09] pero para .Net. Es decir, en los tres casos mencionados los modelos producidos pueden contener transiciones espurias.

La principal diferencia entre Mentat y esos trabajos reside en la manera en la que se construyen los modelos para aproximar EPAs. En el presente trabajo dicha construcción se basa en la ejecución del código, teniendo en cuenta tanto al invariante como a las precondiciones de los métodos de la clase.

En el trabajo [WZ08] se hace minería de la secuencia de operaciones que modifican el estado de las variables para llamar a un método particular de una API. Esta técnica tiene dos etapas; en la primera se hace mining de las secuencias comunes en diferentes usos de la misma API; y en la segunda se comparan las secuencias comunes con un uso particular para detectar posibles errores o violaciones de uso. En contraposición a Mentat la minería de secuencia de operaciones hace foco en los clientes de la API tratando de detectar patrones comunes de uso para luego detectar violaciones de dichos patrones.

Un enfoque similar al presente trabajo se encontró en [KBM14] y en [LMP08]. En ambos trabajos se utilizan trazas obtenidas a partir de la ejecución del código fuente de la clase bajo estudio. La principal diferencia con el presente trabajo, es que Mentat provee un mecanismo explicito de selección de los métodos a ejecutar, lo que permite una exploración profunda del modelo de comportamiento.

El trabajo [GS08] es similar en el sentido de que utiliza ejecuciones para inferencia. Sin embargo, sólo infiere propiedades aisladas del sistema en vez de un modelo que abarque todo el comportamiento. En particular presenta un algoritmo mejorado para buscar patrones de la forma $(ab^+c)^*$, y así detectar este patrón que es importante ya que representa por ejemplo como un recurso que deber ser obtenido, usado y luego liberado.

El trabajo [DKM+10] es similar pero tiene como objetivo la inferencia de suites de

test para verificación en lugar de apuntar a la validación de especificaciones como lo hace MENTAT.

Capítulo 7

Conclusiones

En este trabajo se presenta la herramienta MENTAT que construye aproximaciones al EPA, utilizando como entrada clases Java anotadas para especificar el invariante y las precondiciones de sus métodos, y generando de manera dinámica una traza de ejecución.

Por las características de la herramienta y por cómo se construye la aproximación al EPA, el modelo obtenido es una *sub-aproximación* al EPA de la clase. Por lo tanto, MENTAT es un buen complemento a CONTRACTOR en los casos en que haya transiciones que CONTRACTOR no pueda asegurar su existencia, para así ir aproximándose al EPA por ambos lados.

MENTAT también puede utilizarse como soporte al programador a la hora de hacer pruebas, utilizando el gráfico del modelo obtenido y permitiéndole detectar errores y validar de una manera visual que la clase con la que está trabajando se comporta de la manera esperada.

7.1. Trabajo futuro

Hay varias extensiones que se le podrían hacer a la herramienta actual, considerando más situaciones de uso. Por ejemplo, dependiendo del tipo de parámetros que necesiten generarse en `Z3`, puede necesitarse ampliar los tipos soportados actualmente.

Otra extensión que podría pensarse, es generar los modelos en base a trazas de ejecuciones reales, para lo cual se podría extraer información de logs de ejecuciones del programa, para luego utilizarse como entrada para generar el modelo. Para ejercitar el código a estudiar una técnica interesante es utilizar herramientas como Randoop[PE07] o EvoSuite[FA11] para generar tests de unidad y obtener las trazas a partir de su ejecución.

Para generalizar el uso de este tipo de herramientas en la industria del software sería útil contar con plugins para los IDEs de programación más usados en Java como ser Eclipse e IntelliJ IDEA. Las principales funcionalidades que debe proveer el plugin deben ser la validación de la sintaxis del lenguaje del invariante de la clase y las precondiciones de cada método y permitir la configuración de ejecuciones de MENTAT.

Con respecto a la decisión tomada de evaluar de manera circular la lista de funciones del archivo de función de selección, podría pensarse en alguna otra estrategia una vez

que se consumió toda la lista.

Desde un enfoque más teórico sería interesante realizar un estudio que compare los modelos obtenidos con técnicas de validación de código estáticas contra técnicas dinámicas.

Otro camino a explorar podría ser cómo se puede extender la herramienta para generar modelos más expresivos, por ejemplo buscar algún mecanismo para reducir el grado de no determinismo en los EPAs.

Bibliografía

[AČMN05] R. Alur, P. Černỳ, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL '05*, pages 98–109, 2005.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[dCBGU09] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Validation of contracts using enabledness preserving finite state abstractions. In *ICSE '09*, pages 452–462, 2009.

[dCBGU10] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Enabledness-based program abstractions. Submitted to ISSTA'10, 2010.

[dCBGU11] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Program abstractions for behaviour validation. In *ICSE 2011*, pages 381–390, 2011.

[DKM+10] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 85–96. ACM, 2010.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[EGK+02] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz-open source graph drawing tools. In *Graph Drawing*, pages 483–484. Springer, 2002.

[FA11] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

[GS08] Mark Gabel and Zhendong Su. Symbolic mining of temporal specifications. In *Proceedings of the 30th international conference on Software engineering*, pages 51–60. ACM, 2008.

[Hic08] R. Hickey. The Clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*. ACM New York, NY, USA, 2008.

[KBM14] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 178–189. ACM, 2014.

[LMP08] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, pages 501–510. ACM, 2008.

[PE07] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

[SY86] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, 12(1):157–171, 1986.

[Uri99] T. Uribe. *Abstraction-based Deductive-algorithmic Verification of Reactive Systems*. Stanford University, Dept. of Computer Science, 1999.

[WZ08] Andrzej Wasylkowski and Andreas Zeller. Mining operational preconditions, 2008.

[ZBdC+11] Edgardo Zoppi, Víctor Braberman, Guido de Caso, Diego Garbervetsky, and Sebastián Uchitel. Contractor. net: inferring typestate properties to enrich code contracts. In *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*, pages 44–47. ACM, 2011.

[Zop12] Edgardo Julio Zoppi. *Enriqueciendo Code Contracts con Typestates*. PhD thesis, DC, FCEyN, UBA, http://www.dependex.dc.uba.ar/ diegog/licTesis/2012-12-18-Zoppi-Edgardo.pdf, Dicember 2012.