DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Tesis de Licenciatura

A*Corral*ando EPAs: acercando el modelo mental al computacional

Abril de 2016

Alumno

Leandro Lera Romero
lleraromero@dc.uba.ar
LU: 187/09

Director

Dr. Diego Garbervetsky
diegog@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Resumen

Por lo general un componente de software tiene requerimientos que no son triviales cuando se trata del orden en que sus operaciones (e.g., métodos o procedimientos) pueden ser ejecutados. Esta situación es común, por ejemplo, en el caso de APIs que implementan protocolos. Este trabajo apunta a proveer herramientas que ayuden en el proceso de validación. En particular, nos interesa facilitar la tarea de determinar si una implementación particular de un componente cumple con el comportamiento esperado en los casos en que su descripción es informal, parcial o directamente inexistente.

Contractor es una herramienta que permite construir abstracciones con información de typestate (una máquina de estados que modela secuencias de operaciones válidas) a partir del uso de especificaciones basadas en contratos. Este nivel de abstracción mostró resultados positivos al momento de validar el modelo, ayudando a la detección de errores, ajustes en los requerimientos y la generación de documentación más completa y detallada.

Tomamos Contractor.NET, un prototipo de Contractor para analizar programas .NET, y le incorporamos a su *engine* un nuevo motor de razonamiento llamado Corral. Corral es un verificador automático que permite resolver preguntas de "alcanzabilidad de código" de forma eficiente y precisa.

Los principales desafíos de esta tesis son a) la codificación del programa y sus contratos en una forma amigable para Corral; b) la correcta interpretación de los resultados de la herramienta, ya que la misma no utiliza el mismo principio de abstracción que en versiones anteriores y c) la implementación de un prototipo eficiente que sea capaz analizar casos de estudio en un tiempo razonable.

Abstract

Code artifacts that have non-trivial requirements with respect to the ordering in which their methods or procedures ought to be called are common and appear, for instance, in the form of API implementations and objects. This work addresses the problem of validating if API implementations provide their intended behaviour when descriptions of this behaviour are informal, partial, or non-existent.

Contractor is a tool that addresses this problem by generating abstract behaviour models which resemble typestates. These models are statically computed, based on contracts embedded in the source code, and encode all admissible sequences of method calls. The level of abstraction at which such models are constructed has shown to be useful for validating code artifacts and identifying findings which led to the discovery of bugs, adjustment of the requirements expected by the engineer to the requirements implicit in the code, and the improvement of available documentation.

We improved Contractor.NET, an existing prototype based on Contractor capable of analysing .NET bytecode, by adding a new decision engine which relies on Corral. Corral is a solver that tackles the reachability-modulo-theories problem in a precise and efficient way.

The main challenges of this thesis are a) finding the proper encoding of a program and its contracts in order to use Corral; b) figuring out the correct interpretation of the results obtained using Corral, since it does not use the same abstraction principle that has been used in previous versions of Contractor.NET and c) improving the performance of the prototype so as to make it capable of analysing larger case studies.

Agradecimientos

Quiero dedicarle algunas líneas para agradecer a aquellos que supieron acompañarme durante toda esta etapa de mi vida.

En primer lugar, me gustaría mencionar a Hernán Melgratti y Juan Pablo Galeotti por haber aceptado ser jurados de mi tesis y por haberse tomado el trabajo de leer y evaluar el trabajo que realizamos.

Además quiero agradecer a todo el hermoso grupo de gente de LaFHIS que supo abrime las puertas. Particularmente, debido a la naturaleza de esta tesis, tuve la suerte de trabajar y compartir mucho tiempo con Edgard, Fernán y Guido. Muchas gracias genios!

Para poder llegar a la tesis tuve que pasar antes por todo el resto de la carrera. Sin lugar a dudas, tuve la enorme fortuna de poder conocer a grandes amigos que supieron hacer muy especial el día a día. Especialmente, quiero reconocer al grupo de gente más colgado de todos, aquellos con los que pude compartir grandes momentos. Gracias Ivi, March, Ari, Juansis, Mati, Nico! A pesar de que muchas veces los odié porque todo tardaba infinito, la pasamos muy bien. Gracias también Manu y Luz, son dos de las personas más nobles que conozco y me alegro mucho haberlos conocido.

Un párrafo aparte para mis amigos de la vida, esos que me vienen bancando desde siempre, que se cansaron de preguntarme "¿todavía seguís con la tesis?". Gracias por todo el aguante todo este tiempo, por escuchar y por estar ahí. A ustedes, Mianco, Flor, Alan, Maru, Zeta, Fefe, Danip, Danis, Nadu, mis infinitas gracias. Los quiero mucho de verdad.

Diego, ¿qué decir?, sos un grande. Para todo aquel que esté leyendo esto y pensando si elegirlo como director de tesis le digo, ni lo dudes, por supuesto que si! Quizá una de las mejores decisiones que tomé en mi vida, tenerte como director. Siempre fuiste súper generoso conmigo y abierto a escuchar mis inquietudes. Sos un excelente director, pero por sobre todo una excelente persona. Gracias, gracias, gracias.

Por último, quizá los más importantes, mi familia. Gracias por ayudarme a recorrer este camino, cada uno a su manera. Siempre estuvieron al lado mío, festejando en los mejores momentos y sosteniéndome en donde el camino se hacía más sinuoso. Mi amor por ustedes siempre va a ser inconmensurable y comparto este éxito con ustedes que por supuesto también se lo merecen. Gracias Mami, Papi, Gon!

Naturalmente, las palabras me quedan cortas para expresar mi profundo agradecimiento a todos los "culpables", a los que nombré y a los que no también, de este logro en mi vida.

Simplemente gracias.

Índice general

1. Introducción **9**
1.1. Objetivo 10
1.2. Contribuciones 10
1.3. Trabajo relacionado 11
1.4. Estructura de la tesis 11

2. Motivación **13**

3. Preliminares **17**
3.1. Enabledness-Preserving Abstractions (EPAs) 17
3.1.1. Modelo formal 17
3.2. Generación de EPAs 20
3.2.1. Algoritmo de exploración bajo demanda 20
3.3. Corral: Reachability Modulo Theories solver 22
3.3.1. Stratified inlining 24

4. Implementación **25**
4.1. Corral como motor de decisión 25

5. Resultados **31**
5.1. Comportamiento de Corral en casos básicos 31
5.2. Corral vs. Code Contracts 47
5.3. Limitaciones conocidas 52

6. Conclusiones **57**

7. Trabajo futuro **59**

8. Bibliografía **61**

Capítulo 1

Introducción

El diseño por contratos [22] es una disciplina de programación que incentiva a los diseñadores de software a definir especificaciones formales, precisas y verificables para las interfaces de los componentes de software. Se lleva a cabo mediante la extensión de la definición estándar de los tipos abstractos de datos con precondiciones, postcondiciones e invariantes. Entre las ventajas que ofrece el uso de este enfoque se encuentran la posibilidad de la verificación del código de forma estática, el uso de la especificación como documentación adicional del proyecto e incluso la mejora en la calidad del software por medio de chequeos en tiempo de ejecución.

Debido a que esta metodología está tomando cada vez más importancia en el ámbito de la ingeniería de software gracias a que permite la verificación automática, en los últimos años se han desarrollado distintas tecnologías que intentan expresar formalmente dicha especificación de manera sencilla. Code Contracts [12] es una de ellas. Este proyecto de Microsoft Research acerca las ventajas de la programación mediante el diseño de contratos [22] a todos los lenguajes basados en .NET, sin requerir anotaciones especiales ni un compilador específico.

A pesar de que Code Contracts permite incorporar el uso de contratos de forma muy intuitiva, es muy frecuente observar que la mayoría de ellos están más enfocados en la especificación de precondiciones en lugar de incluir también postcondiciones e invariantes. Es decir, son pocas las clases que cuentan con especificaciones que describan el estado en el cual una instancia debe encontrarse después de la ejecución de uno de sus métodos. Esto se debe, en gran medida, a que no es una tarea fácil escribir contratos que se refieran a estructuras de datos compartidas y modificadas por varios métodos.

Existe, sin embargo, una alternativa que permite ayudar a la comprensión global del comportamiento de un objeto en tiempo de ejecución: las *typestate specifications* [11]. Estas especificaciones definen el conjunto de estados posibles en los cuales un objeto puede estar a lo largo de su ciclo de vida y codifican todas las secuencias permitidas de invocaciones de sus métodos. Generalmente, se los utiliza para asegurar propiedades que dependen y cambian el estado de los objetos. Como una forma de caracterizar dichas propiedades, se dice que un *typestate* es seguro [1] si no existe una secuencia de invocaciones que viole los invariantes internos de la biblioteca; y es permisivo [17] si contiene a todas y cada una de dichas secuencias.

Con el objetivo de combinar los beneficios del diseño con contratos y los typestates, se diseñó hace unos años la primer técnica de generación de abstracciones que permiten explicar el comportamiento de clases [7] usando *model checkers*. Estas abstracciones, llamadas *enabledness-preserving*, consisten en modelos finitos de comportamiento que agrupan instancias de una clase según los métodos que se encuentran habilitados en ellas. Una de sus características más interesantes, aún proveyendo un nivel de abstracción entendible por personas no involucradas en el proceso de programación, es que permite trazar formalmente su relación con el código fuente. Como consecuencia, mostraron resultados positivos al momento de validar modelos, ayudando a la detección de errores

tanto en los contratos [9], como en el código fuente [8], llevando a un mejor entendimiento de los requerimientos por parte del diseñador de la API.

Naturalmente, para poder generar el typestate de una clase anotada con precondiciones, es necesario entender el comportamiento de la misma. Para esto, se realizan consultas que permiten determinar la posibilidad de que un método esté habilitado para su ejecución para un conjunto de instancias.

Luego de haberse implementado una versión que permite analizar programas escritos en C, se extendió el uso de esta técnica en lenguajes de más alto nivel. Esta nueva implementación, llamada Contractor.NET [25], resuelve mediante el uso de Clousot, las consultas que permiten evaluar los métodos, a través del uso de *abstract interpretation* [5].

Este prototipo demostró la factibilidad del análisis de clases en .NET, acercando la técnica a casos de uso más frecuentes en la industria. Sin embargo, por limitaciones de Clousot, los resultados obtenidos no fueron lo suficientemente satisfactorios, invitando a considerar otras alternativas.

1.1. Objetivo

Nos interesa mejorar la herramienta Contractor.NET para que sea más eficaz a la hora de facilitar la validación de artefactos de software mediante abstracciones. Esperamos que sea capaz de incluir casos de mayor escala y mejorar la capacidad de detección de inconsistencias en los mismos. Además, intentaremos que sea más precisa en los resultados que logra obtener e idealmente que reduzca los tiempos de generación.

Por otro lado, en el contexto de la generación de EPAs, queremos comprender mejor los pros y contras de aplicar técnicas de inferencia tales como la interpretación abstracta, que trabaja con sobre-aproximaciones y cálculo automático de invariantes, versus técnicas de verificación como model checking, que utilizan representaciones más explicitas del espacio explorado.

1.2. Contribuciones

Las principales contribuciones de esta tesis son:

- La codificación del programa y sus contratos en una forma amigable para Corral. Al incorporar el uso de Corral para la resolución de consultas, permitimos la ampliación del espectro de casos que son manejables por la herramienta.
- La modificación de los algoritmos de generación de EPAs para la correcta interpretación de los resultados de la herramienta ya que la misma no utiliza el mismo principio de abstracción que versiones anteriores
  - La incorporación de una mejora en las consultas utilizadas por el algoritmo de forma de poder obtener resultados más precisos
- La implementación de un prototipo eficiente con el cual se pueden analizar casos de estudio en un tiempo razonable
- La evaluación de los resultados obtenidos y contrastados con las dos herramientas implementadas previamente
  - Discutimos algunas posibles mejoras para confirmar transiciones factibles y eliminar transiciones espurias mediante el uso de Corral

1.3. Trabajo relacionado

En [7] se estudian las enabledness-based abstractions y su potencial para validar especificaciones con contratos. Al igual que en nuestro trabajo, se utiliza directamente el código como objeto de estudio haciendo más complejo el análisis. Esto se debe a que las características propias del lenguaje como ciclos, manejo de memoria y llamadas a métodos, suelen ser complejas y no necesariamente tan declarativas. A diferencia de dicho trabajo, en el que se emplea código C para el análisis, nuestro proyecto hace un estudio de bytecode de .NET permitiendo incorporar lenguajes de más alto nivel. Además, para evitar tener que incluir anotaciones especiales, en forma de sobre- y sub-aproximaciones en las precondiciones, utilizamos un método que no necesita anotaciones pero que permite de todas formas utilizar todo el potencial de los model checkers para la construcción de EPAs. Por su parte, en [25] se analiza la construcción de EPAs para lenguajes de mas alto nivel como C#, mediante el uso de abstract interpretation, pero sin poder aprovechar el potencial de los model checkers.

Asimismo, nuestra técnica está relacionada con enfoques que sintetizan typestates [23, 10] o interfaces [1, 14, 17] a partir de un programa. Cualquier secuencia de métodos que no sea aceptada por nuestra abstracción, no será permitida por un programa que la codifique. Sin embargo, en este tipo de propuestas el foco está puesto en la verificación modular [11, 2] más que en la validación. Como consecuencia, los modelos que construyen suelen ser demasiado restrictivos y no están pensados para ser inspeccionados y analizados por personas.

También es posible considerar nuestra propuesta como una forma de abstracción de predicados [24]. Desde este punto de vista, nuestra propuesta está relacionada con técnicas que construyen grafos de estados abstractos finitos a partir de sistemas de estados concretos infinitos [19, 15, 16]. Sin embargo, estas técnicas se enfocan en la utilización de las abstracciones para la verificación o la generación de casos de prueba en lugar de la validación. Por lo tanto, el nivel de abstracción y el tamaño de los modelos obtenidos varían de forma significativa dificultando su inspección y validación. En [20] se utiliza un nivel de abstracción similar al nuestro, pero el modelo no está pensado para representar comportamiento (no define transiciones entre estados) sino que se utiliza para definir criterios de cubrimiento de casos de prueba (test coverage).

Nuestra propuesta también está relacionada con las técnicas de minería de especificaciones temporales [13, 21, 6], las cuales producen a partir de trazas un autómata de estados finitos que describe cómo son utilizadas un conjunto de operaciones. Nuevamente el enfoque está puesto en la generación de casos de prueba y la verificación del código cliente. Más aún, estas técnicas suelen ser dinámicas y dependen fuertemente de la calidad y cantidad de las trazas utilizadas para el análisis.

1.4. Estructura de la tesis

En el capítulo 2 se presenta un ejemplo ilustrativo y completo sobre el modo de uso de la herramienta Contractor.NET, así como también los beneficios obtenidos.

En el capítulo 3 se introduce una definición formal de las abstracciones de tipo *enabledness-preserving* junto con dos versiones del algoritmo de construcción de las mismas. Ambas alternativas consisten en la versión *naïve*, muy simple de entender pero poco efectiva en la práctica, y la segunda basada en una exploración más eficiente.

Adicionalmente, se presenta una noción de Corral como tecnología. Se propone una discusión acerca de su arquitectura, para que lector adquiera una visión general de sus componentes, y luego se hace foco en el módulo de Stratified Inlining.

En el capítulo 4 se pretende dar una visión general de la herramienta Contractor.NET que es una implementación concreta de lo explicado en capítulos anteriores. Se hace hincapié en los

detalles de implementación particulares, tecnologías involucradas y se realiza una comparación con la versión anterior para resaltar las diferencias y similitudes.

Posteriormente, en el capítulo 5 se presentan algunos ejemplos representativos y se analizan los resultados obtenidos nuevamente comparándolos con las implementaciones existentes que permiten generar EPAs. A continuación, se lleva a cabo una comparación entre las implementaciones que soportan código .NET con diversos ejemplos discutiendo acerca de las diferencias en las EPAs obtenidas.

Para finalizar, el capítulo 6 corresponde a las conclusiones derivadas del trabajo realizado y posteriormente, en el último capítulo, se proponen algunas alternativas de trabajo a futuro.

Capítulo 2

Motivación

El objetivo de esta sección es presentar un escenario simple y de fácil entendimiento para mostrar cómo el uso de contratos y abstracciones ayuda a la comprensión de un artefacto de software.

Supongamos que estamos intentando desarrollar el software necesario para controlar la típica máquina expendedora de bebidas como la que se observa en la Figura 2.1.

Figura 2.1: Clásica máquina expendedora

El primer paso para el desarrollo es saber cuáles son los aspectos que nos interesan modelar en este caso.

En un tipo de máquinas como ésta, la expendedora recibe dinero mediante su ranura de billetes para aceptar el pago de los usuarios. Una vez que la cantidad de dinero ingresado es suficiente para realizar la compra, permite seleccionar alguna de las bebidas disponibles. Por último, en aquellos casos en los que el cliente pague con un monto superior al valor de la bebida la máquina devuelve el dinero excedente. Una vez realizados todos estos pasos la máquina se encuentra nuevamente disponible para el próximo cliente.

Una vez que se conocen las acciones que debe poder realizar nuestro programa, podemos pensar en cómo modelar estas operaciones. Para eso, una alternativa podría ser escribir una clase que permita crear instancias que respondan a los siguientes mensajes:

- **IngresarDinero**: que permite modelar el ingreso de dinero a la máquina y así sumarle crédito al cliente.
- **LiberarBotella**: que modela el hecho de que el usuario haya seleccionado la bebida que desea tomar y por lo tanto la máquina debe liberar la botella correspondiente.

- **DarVuelto**: en caso de que el dinero ingresado sea superior al valor del producto (que por simplicidad lo consideramos como único para todas las bebidas), se le debe poder indicar a la máquina que devuelva el excedente, para que el cliente solamente pague el valor correspondiente a la botella.

En la Figura 2.2, se puede ver una posible implementación de dicha clase[1].

Como se puede observar en el código, además de las instrucciones necesarias para representar el progreso en el proceso de la compra, se incluyeron condiciones necesarias para que los métodos puedan ser ejecutados. Estas precondiciones están expresadas en forma de contratos utilizando la biblioteca de Code Contracts provista por el framework de .NET. Algunas de éstas están definidas de forma explícita por nuestro problema mientras que otras surgieron de la forma en que la solución fue modelada.

Por ejemplo, en el caso de **IngresarDinero** se espera que el *monto reconocido* por el lector de billetes sea un entero positivo (consecuencia de modelar el dinero como un *int*) y que a su vez no haya otra compra en curso (ya que no tiene sentido el ingreso de dinero del comprador siguiente si el anterior no terminó).

Es importante destacar que suele ser una tarea difícil determinar el conjunto de condiciones necesarias para restringir el uso de los métodos a los casos en los que deberían ser utilizados. Más aún, poder definir dichos casos es un problema en sí mismo y es muy frecuente el hecho de que se cuente con un conocimiento parcial del problema a resolver mientras se intenta crear una solución.

Otro inconveniente muy frecuente es que quien entiende la realidad que se quiere modelar no necesariamente sabe programar, y viceversa. Es decir, existe una distancia entre quien sabe qué es lo que hay que hacer de quien conoce el cómo. En nuestro caso, es posible que el Sr. Coca-Cola no sepa programar, pero sí sabe lo que se espera del funcionamiento de la expendedora.

Veamos entonces, cómo usando una abstracción podemos intentar salvar esta distancia. Si generamos la abstracción correspondiente al código de nuestro software de control de la máquina expendedora, obtenemos el resultado que se observa en la Figura 2.3.

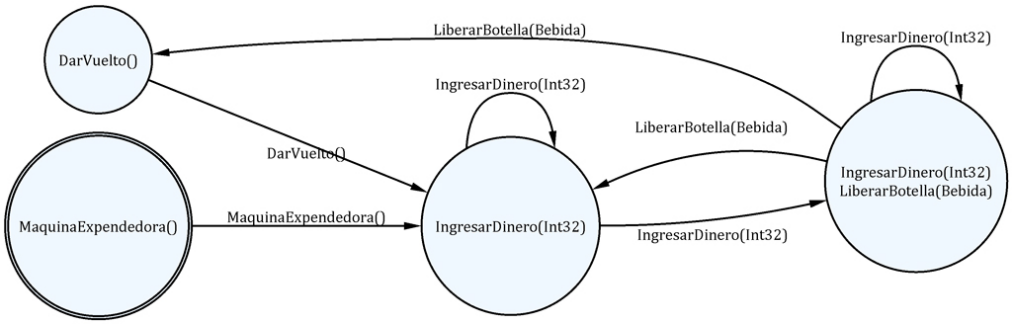

Figura 2.3: Abstracción del comportamiento del modelo de la Máquina Expendedora

A grandes rasgos, podemos decir que la abstracción obtenida permite visualizar gráficamente los posibles órdenes (o trazas) en los cuales se pueden ir llamando los métodos de la clase, de forma tal que siempre se respeten las precondiciones. En otras palabras, dado que las instancias de la clase pretenden ser una representación computable de la realidad que modelan — en este caso la máquina expendedora — y por lo tanto se comportan de la forma en que la máquina física lo hace (ingresar dinero, liberar la botella y dar el vuelto) la Figura 2.3 debería permitirnos "simular" el comportamiento de la máquina expendedora del mundo real.

[1] Vale la pena aclarar, que hay detalles de implementación que no se muestran ya que no aportan información a nuestro estudio y hacen más compleja la comprensión del ejemplo.

```
public enum Bebida { Coca, Light, Zero, Sprite, Agua }
public class MaquinaExpendedora
{
    protected bool ventaEnCurso;
    protected int dineroDisponible;

    [Pure] public bool EstaVendiendo() { return ventaEnCurso; }
    [Pure] public int DineroDisponible() { return dineroDisponible; }
    [Pure] public int PrecioDeLaBotella() { return 15; }

    public MaquinaExpendedora()
    {
        ventaEnCurso = false;
        dineroDisponible = 0;
    }

    public void IngresarDinero(int montoReconocido)
    {
        Contract.Requires(montoReconocido > 0);
        Contract.Requires(!EstaVendiendo());

        dineroDisponible += montoReconocido;
    }

    public void LiberarBotella(Bebida bebidaSeleccionada)
    {
        Contract.Requires(!EstaVendiendo());
        Contract.Requires(DineroDisponible() >= PrecioDeLaBotella());

        ventaEnCurso = true;
        dineroDisponible -= PrecioDeLaBotella();

        // Dejar caer una botella de la bebidaSeleccionada

        if (dineroDisponible == 0)
            ventaEnCurso = false;
    }

    public void DarVuelto()
    {
        Contract.Requires(EstaVendiendo());
        Contract.Requires(DineroDisponible() > 0);

        dineroDisponible = 0;
        ventaEnCurso = false;
    }
}
```

Figura 2.2: Implementación del controlador de la máquina expendedora

Si bien las transiciones en el modelo respetan el nombre que se les otorgó en el código, se puede ver que la mayor parte de la "programación" está oculta dentro de la abstracción. Luego, para simular distintos escenarios que la expendedora debería contemplar (y también los que no), basta con ver si existe algún camino de flechas en el diagrama que representan el comportamiento del escenario.

Por ejemplo, podríamos tener el caso en el que se ingresa una cantidad superior de dinero a la necesaria, y por lo tanto, luego de haber liberado la bebida se debe proceder a devolver el vuelto al cliente, como se aprecia en la siguiente figura.

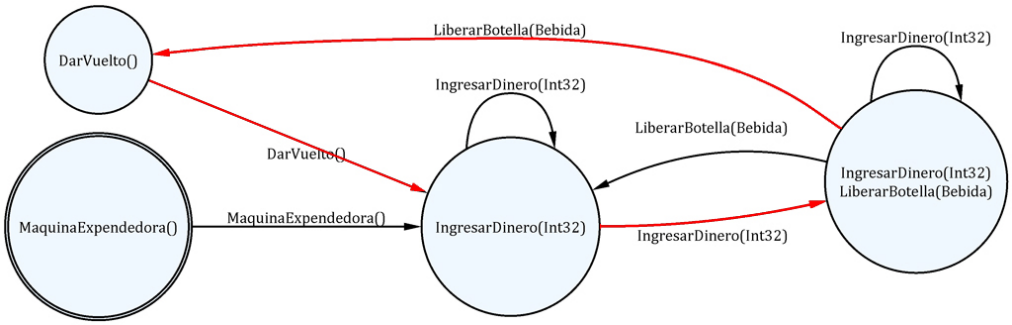

Figura 2.4: Análisis del caso de la devolución del vuelto

A lo largo de la tesis veremos cómo es que se hace para obtener estas abstracciones y algunas de las propiedades que las caracterizan.

Capítulo 3

Preliminares

3.1. Enabledness-Preserving Abstractions (EPAs)

Antes de presentar los algoritmos que permiten construir de forma automática la abstracción, es necesario introducir primero los aspectos formales que los fundamentan así como también establecer una notación clara para los mismos. Para mayor detalle e información al respecto consultar [7], trabajo en el cual nos basamos.

3.1.1. Modelo formal

El objeto de análisis para nuestra propuesta es el código fuente de una clase. Por lo tanto, es necesario como primer paso definir la interpretación semántica del mismo.

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

1. $M = \{m_1, \ldots, m_n\}$ es un conjunto finito de etiquetas de los métodos que saben responder las instancias de la clase
2. F es el conjunto de implementaciones de los métodos (*functions*) indexado por $m \in M$
3. R es el conjunto de precondiciones (*requires clauses*) indexado por $m \in M$
4. inv es el invariante de la clase
5. $init$ es la condición inicial, que indica cuáles son las instancias iniciales de la clase

Ejemplo 1

Veamos como se aplica este modelo al ejemplo presentado previamente. Recordando, los métodos que expone la clase **MaquinaExpendedora** son: *IngresarDinero*, *LiberarBotella*, *DarVuelto*.

Luego, la interpretación semántica del ejemplo bajo el modelo formal sería:

$$MaquinaExpendedora = \langle M, F, R, inv, init\rangle \text{ donde,}$$

$M = \{IngresarDinero, LiberarBotella, DarVuelto\}$

$F = \{f_{IngresarDinero}(montoReconocido), f_{LiberarBotella}, f_{DarVuelto}\}$

$$R = \{R_{IngresarDinero}(montoReconocido), R_{LiberarBotella}, R_{DarVuelto}\}$$

$$\begin{aligned} R_{IngresarDinero}(montoReconocido) &= \text{montoReconocido} > 0 \wedge \text{!EstaVendiendo()} \\ &\equiv \text{montoReconocido} > 0 \wedge \text{!ventaEnCurso} \end{aligned}$$

$$\begin{aligned} R_{LiberarBotella} &= \text{!EstaVendiendo()} \wedge \text{DineroDisponible()} \geq \text{PrecioDeLaBotella()} \\ &\equiv \text{!ventaEnCurso} \wedge \text{dineroDisponible} \geq 15 \end{aligned}$$

$$\begin{aligned} R_{DarVuelto} &= \text{EstaVendiendo()} \wedge \text{DineroDisponible()} > 0 \\ &\equiv \text{ventaEnCurso} \wedge \text{dineroDisponible} > 0 \end{aligned}$$

$$inv = \text{true}$$

$$init = \text{!ventaEnCurso} \wedge \text{dineroDisponible} = 0$$

¿Por qué no aparecen los métodos "constructores" en el modelo? El lector perspicaz pudo haber notado que si bien en la EPA usada como motivación aparecen transiciones con los constructores de la clase estos no aparecen en el modelo formal. Es fácil ver que debido a que los métodos de construcción de instancia no son mensajes que puedan responder las instancias de la clase estos no pertenecen a la EPA de la clase. Sin embargo, en lenguajes como C# o Java en donde existe una mezcla entre el paradigma imperativo y el orientado a objetos, el programador define cómo se crea una instancia y qué mensajes sabe responder en el mismo archivo de texto. Es por este motivo que nos parece apropiado hacer este abuso de notación con el objetivo de hacer más relevante la información que aporta la abstracción.

Más aún, como veremos más adelante los constructores tienen un tratamiento especial en el algoritmo de creación de EPAs.

Lo siguiente que debemos definir es el espacio de estados posibles, el cual se caracteriza mediante un sistema infinito y determinístico de transiciones etiquetadas (*labelled transition system* o 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 de transición parcial.

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) y un estado inicial que cumple 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 ver [8].

Ahora que tenemos definido el espacio de estados posibles (recordar que es infinito) debemos establecer un nivel de abstracción adecuado para obtener una representación finita del mismo, la cual podamos 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. Esto se debe a que brinda buena trazabilidad con el código ya que se puede navegar a los métodos y mirar las precondiciones.

Por lo tanto es necesario formalizar esta noción de equivalencia de instancias. Dada una clase C y dos instancias $c_1, c_2 \in C$ decimos que c_1 y c_2 son equivalentes respecto de los métodos que habilitan, es decir *enabledness equivalent* (lo notaremos $c_1 \equiv_e c_2$), si y solo si 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 podemos definir un tipo de abstracción denominada *enabledness-preserving* (o simplemente EPA) como una máquina de estados 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 [7] para una definición formal.

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 a todas las instancias que comparten el mismo conjunto de métodos habilitados y pueden ser caracterizados con un *invariante de estado*.

Dicho invariante hace posible la construcción de este tipo de abstracciones, debido a que si bien desde un punto de vista teórico podemos obtener el EPA a partir del LTS utilizando el concepto de equivalencia de instancias descripto previamente, en la práctica esto no es posible dado que el LTS es infinito. Por lo tanto, es necesario recurrir a algún mecanismo que permita generar EPAs directamente del código fuente 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 $inv_{ms} : C \to \{true, false\}$,

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

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 habilitadas (y ningún otro método lo está, es decir, todos los otros métodos que no pertenecen a este conjunto se encuentran deshabilitados en esa misma instancia c).

Ejemplo 2

Usando la definición del invariante de estado en el ejemplo de la máquina expendedora, los invariantes para cada estado que aparece en la abstracción serían los siguientes:

$$\begin{aligned}
inv_{\{IngresarDinero\}}(c) &\Leftrightarrow inv(c) \wedge R_{IngresarDinero}(c,m) \wedge \neg R_{LiberarBotella}(c) \\
&\quad \wedge \neg R_{DarVuelto}(c) \\
&\Leftrightarrow true \ \wedge \ (\exists m.\ m > 0 \wedge !c.ventaEnCurso) \wedge \\
&\quad \neg(!c.ventaEnCurso \ \wedge \ c.dineroDisponible \geq 15) \wedge \\
&\quad \neg(c.ventaEnCurso \wedge c.dineroDisponible > 0) \\
&\Leftrightarrow (\exists m.\ m > 0 \wedge !c.ventaEnCurso) \wedge \\
&\quad (c.ventaEnCurso \ \vee \ c.dineroDisponible < 15) \wedge \\
&\quad (!c.ventaEnCurso \vee c.dineroDisponible \leq 0) \\
&\Leftrightarrow (\exists m.\ m > 0 \wedge !c.ventaEnCurso) \ \wedge c.dineroDisponible < 15
\end{aligned}$$

$$\begin{aligned}
inv_{\left\{\begin{smallmatrix} IngresarDinero \\ LiberarBotella \end{smallmatrix}\right\}}(c) &\Leftrightarrow inv(c) \wedge R_{IngresarDinero}(c,m) \wedge R_{LiberarBotella}(c) \\
&\quad \wedge \neg R_{DarVuelto}(c) \\
&\Leftrightarrow (\exists m.\ m > 0 \wedge !c.ventaEnCurso) \ \wedge \ c.dineroDisponible \geq 15
\end{aligned}$$

$$\begin{aligned}
inv_{\{DarVuelto\}}(c) &\Leftrightarrow inv(c) \wedge R_{DarVuelto}(c) \wedge \neg R_{IngresarDinero}(c,m) \wedge \neg R_{LiberarBotella}(c) \\
&\Leftrightarrow c.ventaEnCurso \wedge c.dineroDisponible > 0
\end{aligned}$$

Habiendo caracterizado formalmente los estados abstractos lo único que resta definir son las transiciones de la abstracción.

Vamos a decir que dos estados ms y ms' estarán conectados mediante una transición con etiqueta m si existe una instancia c que evoluciona en c' luego de ejecutar el método m. Además hay que tener en cuenta que debe valer que en ms sea válida la precondición de m y que c' satisfaga el invariante de ms'.

Ahora si estamos en condiciones de caracterizar una EPA. Diremos que una EPA es una estructura $\langle \Sigma, S, S_0, \delta \rangle$ a partir de una clase dada $C = \langle M, F, R, inv, init \rangle$ que cumple las siguientes condiciones:

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. Para todo $ms \in S$ y $m \in \Sigma$,
   a) si $m \notin ms$ entonces $\delta(ms, m) = \emptyset$
   b) 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 la EPA caracterizada resulta ser un LTS finito. Además, la función de transición se define como $\delta : S \times \Sigma \to \mathcal{P}(S)$, cuya imágen es el conjunto de partes de S, por lo tanto, el LTS resultante es no determinístico.

3.2. Generación de EPAs

Ahora que contamos con una caracterización formal de las EPAs es posible definir un algoritmo que nos permita generarlas.

El primer enfoque que uno podría utilizar es el de enumerar todos los posibles estados que puede tener la EPA. Por ejemplo, una clase con n métodos públicos puede tener como máximo 2^n estados abstractos. Como consecuencia, se podría escribir un algoritmo que enumere todo el conjunto $\mathcal{P}(S)$. Luego, se puede determinar cuáles son los estados que cumplen la condición $init$ y a partir de estos, solo nos interesa quedarnos con aquellos estados que son alcanzables.

Naturalmente, esta forma de creación de EPAs es exponencial en la cantidad de métodos y por lo tanto resulta inadecuada para implementar en la práctica. Como veremos a continuación, existe una alternativa considerablemente más eficiente que permite ir "descubriendo" la EPA.

3.2.1. Algoritmo de exploración bajo demanda

Una opción mucho más eficiente es la presentada en [7], cuyo pseudocódigo se exhibe en el Algoritmo 1. La idea detrás de este método es realizar una exploración de búsqueda a lo ancho, o simplemente BFS por sus siglas en inglés, del espacio de estados.

Para construir la EPA vamos a partir con una función de transición (δ) y un conjunto de estados (S) inicializados como vacíos.

El conjunto A^- contiene todos los métodos que no pueden estar habilitados en los estados iniciales, debido a que sus precondiciones nunca se cumplen luego de crear una instancia de la clase. De forma análoga, el conjunto A^+ contiene todos los métodos que siempre están habilitados en todos los estados iniciales.

Habiendo constituido los conjuntos de acciones A^+ y A^- es posible definir el conjunto de estados candidatos para ser iniciales S_0^C. Estos estados se caracterizan por contener todos los métodos de A^+ y ninguno de A^-, notar que no necesariamente vale que $A^+ \cup A^- = A$.

Algoritmo 1 Template para la construcción BFS de una EPA

procedure ConstructEPA(C: $\langle A, F, R, inv, init\rangle$) : $\langle \Sigma, S, S_0, \delta\rangle$
  $\Sigma \leftarrow A$
  $S \leftarrow \emptyset$
  $\delta(as, a) \leftarrow \emptyset\ \forall as, a$
  $A^- \leftarrow \{a \in A \mid \forall\ c.\ init(c) \Rightarrow \neg\exists p.\ R_a(c, p)\}$
  $A^+ \leftarrow \{a \in A \mid \forall\ c.\ init(c) \Rightarrow \exists p.\ R_a(c, p)\}$
  $S_0^C \leftarrow \{as \subseteq A \mid A^+ \subseteq as, A^- \cap as = \emptyset\}$
  $S_0 \leftarrow \{as \in S_0^C \mid \exists c.\ inv_{as}(c) \wedge init(c)\}$
  $W \leftarrow$ queue starting with elements in S_0
  while there is a certain as as the head of W **do**
    $W \leftarrow W - [as]$
    $S \leftarrow S \cup \{as\}$
    for each action $a \in as$ **do**
      $B^- \leftarrow \{b \in A \mid \forall c, p.\ inv_{as}(c) \wedge R_a(c, p) \Rightarrow \neg\exists p'.\ R_b(f_a(c, p), p')\}$
      $B^+ \leftarrow \{b \in A \mid \forall c, p.\ inv_{as}(c) \wedge R_a(c, p) \Rightarrow \exists p'.\ R_b(f_a(c, p), p')\}$
      $S^C \leftarrow \{bs \subseteq A \mid B^+ \subseteq bs, B^- \cap bs = \emptyset\}$
      for each state $bs \in S^C$ **do**
        if $(\exists c.\ inv_{as}(c) \wedge \exists p.\ R_a(c, p) \wedge inv_{bs}(f_a(c, p)))$ **then**
          $\delta(as, a) \leftarrow \delta(as, a) \cup \{bs\}$
          if $bs \notin S \wedge bs \notin W$ **then**
            $W \leftarrow W \cup [bs]$

A partir de los candidatos se definen los estados iniciales como aquellos para los cuales existan instancias que verifiquen la condición inicial (*init*) así como también el correspondiente invariante de estado. De esta forma, conseguimos que el conjunto de estados iniciales cumpla con el punto 3 de la caracterización definida previamente. Naturalmente, mientras más métodos sean clasificados como necesariamente habilitados o deshabilitados menor será la cantidad de estados iniciales candidatos que deberán ser verificados posteriormente.

Una vez determinado S_0 se emplea BFS para continuar con la construcción de la EPA. Para eso se inicializa una cola W con los estados pendientes de ser analizados, en otras palabras con S_0.

Cada vez que un estado as pasa a ser analizado se lo agrega al conjunto de estados de la abstracción, ya que el hecho de estar en la cola lo hace alcanzable. Recordemos que el estado as representa a todas las instancias que únicamente pueden ejecutar los métodos de as. Por lo tanto, vamos a ver a qué nuevos estados podemos llegar al ejecutar alguna de la acciones habilitadas $a \in as$.

De forma similar a la etapa de inicialización del algoritmo, el conjunto B^- contiene todas las acciones que nunca están habilitados luego de ejecutar a a partir del estado as. En otras palabras, aquellos métodos cuyas precondiciones nunca se cumplen luego de ejecutar a. Análogamente, el conjunto B^+ contiene todos los métodos que siempre están habilitados al ejecutar a.

Al igual que antes construimos el conjunto de estados potencialmente alcanzable S^C, los cuales se caracterizan por contener todos los métodos de B^+ y ninguno de B^-.

Posteriormente, cada uno de estos estados candidato es considerado para verificar si puede ser alcanzado realmente al evolucionar instancias de as tras la ejecución del método a. Por último, cada vez que un nuevo estado abstracto bs es alcanzado, se lo encola en W para ser analizado posteriormente.

La demostración de correctitud de este algoritmo fue realizada en el trabajo previo antes citado [7].

Cabe mencionar que la complejidad temporal y espacial en el peor caso sigue siendo exponencial respecto de la cantidad de métodos. Sin embargo, la diferencia radica en que mientras podamos clasificar más métodos como necesariamente habilitados o deshabilitados en un estado en particular, menos estados candidatos deben ser considerados por el algoritmo. En la práctica, esta optimización reduce de forma significativa los tiempos de ejecución.

Sensibilidad de las EPAs ante la falta de información

Tal como está definido el algoritmo estipula *cuáles* son las validaciones que se deben realizar, pero no indica *cómo* resolverlas. Lamentablemente, decidir la validez de una fórmula de primer orden es un problema indecidible en el caso general, lo que nos obliga a analizar el impacto que pueden tener respuestas poco precisas o inciertas en la abstracción resultante. Si miramos atentamente el Algoritmo 1 observaremos que es necesario resolver fórmulas en las líneas 5, 6, 8, 14, 15 y 18.

Analicemos primero la línea 5. Al intentar decidir si una acción a tiene que ser incluida en el conjunto A^-, el resultado de validar la fórmula $\forall c.\ init(c) \Rightarrow \neg\exists p.\ R_a(c,p)$ puede ser incierto. En este caso, es seguro no incluir la acción en el conjunto ya que no hay garantías de que esté necesariamente deshabilitada. Sin embargo, esto no tiene impacto en la EPA resultante ya que el conjunto A^- se utiliza para restringir el conjunto de estados candidatos.

Algo similar ocurre al intentar construir los conjuntos A^+, B^- y B^+. El hecho de servir de optimización en el algoritmo nos permite prescindir de dichos conjuntos parcial o totalmente, sin afectar el resultado.

Distinto es el caso de las otras validaciones (líneas 8 y 18), que son cruciales. La primera afecta el conjunto de estados que efectivamente son catalogados como iniciales. Por su parte, la segunda afecta las transiciones de la abstracción y por lo tanto se ven también afectados los estados alcanzables que deben ser analizados. Ante una eventual incertidumbre la opción conservadora es no restringir el resultado, agregando los estados y transiciones de todas formas (adornadas con el signo ? para distinguirlas). De esta forma se logra obtener en el peor caso un super-conjunto del conjunto de estados iniciales (incertidumbre en la línea 8) así como también del conjunto de transiciones (incertidumbre en la línea 18).

A partir de esta decisión se deduce que la abstracción resultante del algoritmo, en un contexto de incertidumbre, es una simulación de la EPA ideal. Este resultado es importante porque cuenta con la ventaja de que la abstracción obtenida es una sobre-aproximación y por lo tanto es capaz de simular cualquier traza de la abstracción óptima. Como contrapartida, tiene la desventaja de aceptar trazas que puedan llegar a ser no factibles en el programa.

Otra opción podría haber sido generar una sub-aproximación, pero en este caso podría pasar que exista una traza válida que no pueda ser simulada, lo cual pensamos es demasiado restrictivo. Como mínimo todas las trazas válidas deberían poder ser aceptadas por la abstracción. De otra forma, se correría el riesgo de ocultarle al usuario trazas válidas posiblemente no deseadas, pero que pasarían desapercibidas justamente por no mostrarle todo el comportamiento.

3.3. Corral: Reachability Modulo Theories solver

La mayoría de los verificadores automáticos de programas plantean la verificación como un problema de decisión que intenta responder a la siguiente pregunta: *¿existe una prueba que demuestre la ausencia de errores?* Encontrar una respuesta es un problema indecidible, más aún, no es ni siquiera recursivamente enumerable, lo que invita a pensar nuevas alternativas para intentar resolver este complejo desafío.

Por este motivo, el enfoque planteado para desarrollar Corral fue distinto [18]. La propuesta

fue considerar el problema de verificación como un problema de decisión que responda a la pregunta *¿existe una ejecución que muestre la presencia de un error?* Aunque este problema sigue siendo indecidible, es sin embargo recursivamente enumerable [18].

Atacar la verificación bajo este punto de vista presenta algunas ventajas con respecto a la versión clásica. En primer lugar, permite la formulación de problemas acotados y decidibles.

Además, se asemeja bastante a la idea de *bug-finding* y *debugging* que suelen ser las formas más frecuentemente utilizadas por los desarrolladores para justificar que un programa "funciona bien" [1].

Por último, es importante destacar que la búsqueda de contraejemplos no invalida el uso de demostraciones formales. Por el contrario, estas se transforman en una oportunidad de optimización en las búsquedas, más que ser un fin en sí mismas.

Teniendo en cuenta estos lineamientos, se ideó la arquitectura de Corral (Figura 3.1), que permite entender los componentes principales que interactúan dentro del mismo para llevar a cabo el análisis del código.

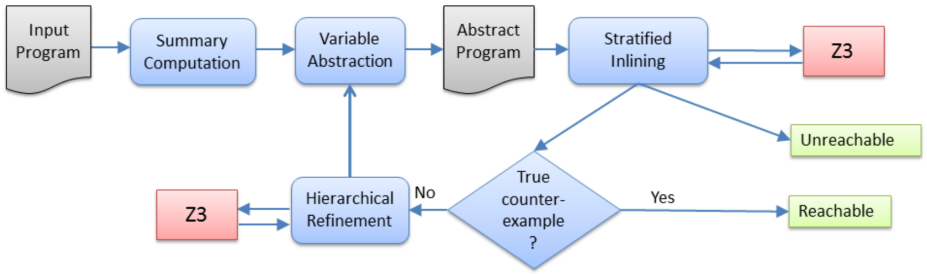

Figura 3.1: Arquitectura de Corral

Es importante notar, que el lenguaje de código fuente que Corral es capaz de analizar es Boogie [3]. Uno de los motivos de esta elección, argumentan sus creadores, es que permite modelar las estructuras de control clásicas, tipos no acotados y operaciones como aritméticas y funciones no interpretadas, que abarcan gran parte de las funcionalidades de lenguajes utilizados en la industria tales como C# o Java. Además, Boogie es capaz de generar *verification conditions*, a partir de porciones de código con un *assert*, que pueden ser resueltas con un *SMT solver* como Z3. Dichas verification conditions serán satisfactibles si y sólo si existe una ejecución del código que viole el assert.

Corral usa un ciclo de *refinamiento de abstracciones guiado por contraejemplos*, conocido como CEGAR loop [4] por sus siglas en inglés, de dos niveles.
El ciclo del primer nivel realiza una abstracción sobre las variables globales del programa. Dado un conjunto de variables que están siendo consideradas en un momento dado, se abstrae el programa utilizando dicho conjunto; el conjunto inicial de variables bajo análisis es vacío. Luego, se pasa esta abstracción al módulo de *Stratified Inlining* para buscar una traza que haga fallar el programa.

En caso de encontrarse una ejecución que lleve a un error puede ser por dos motivos, que surja por no haber considerado alguna variable global, o bien, que efectivamente el programa puede hacer falso un assert. Para poder determinar esto, se chequea esta traza contra el uso de todas las variables globales. Si de todas formas se sigue violando el assert, se concluye que es un bug. En caso contrario, se llama al módulo de *Hierarchical Refinement* que incrementa el conjunto de variables *trackeadas* con el mínimo conjunto de variables necesarias para que el contraejemplo pase a ser espurio.

[1] Aunque E.W. Dijkstra no esté muy de acuerdo: "Program testing may convincingly demonstrate the presence of bugs, but can never demonstrate their absence".

Es importante notar que ambos módulos, Stratified Inlining y Hierarchical Refinement hacen múltiples llamadas al SMT solver para realizar sus tareas.

3.3.1. Stratified inlining

El objetivo del módulo de Stratified Inlining es tratar de demorar la construcción de fórmulas de tamaño exponencial lo más posible. A diferencia de *static inlining*, que reemplaza todas las llamadas a métodos por sus respectivos cuerpos, *stratified inlining* propone ir incluyendo los cuerpos de lo métodos a medida que va encontrando caminos prometedores para alcanzar el objetivo.

Stratified inlining usa sub y sobre-aproximaciones del comportamiento de los métodos para hacer el *inlining* bajo demanda. Para sub-aproximar un procedimiento se bloquean todas las posibles ejecuciones que lo utilicen. Para esto se lo reemplaza con un "assume false", ya que esencialmente esto permite ignorar todas las trazas que sucedan luego de esta instrucción. Por el contrario, para realizar una sobre-aproximación se utiliza un *summary* del comportamiento, es decir una especificación válida, potencialmente sobre-aproximada, de la semántica del método en cuestión.

Por defecto, se utiliza el summary que especifica que el procedimiento potencialmente modifica todas las variables a las que tiene acceso y que no restringe de ninguna manera el output del mismo. Notar que el cálculo de summaries está considerado como un módulo independiente en la arquitectura y por lo tanto puede ser tan sofisticado como se deseé.

El algoritmo comienza con P siendo el **main** del programa a analizar y C es el conjunto de todos los *call-sites* en el **main**. En todo momento, se mantiene un programa P parcialmente *inlined*, junto con un conjunto de *call-sites* C en P que todavía no han sido *inlined*.

Cada iteración se compone de dos etapas. En la primera, cada call-site abierto en P es reemplazados por su sub-aproximación para así obtener un programa cerrado P', que es chequeado usando el theorem prover. Lo que se intenta en este caso es determinar la satisfactibilidad del assert en el programa. Si es posible violarlo, el algoritmo concluye que hay un *bug*.

En caso contrario se procede a la segunda etapa. En esta se recurre a la sobre-aproximación, mediante el uso de los summaries. Cada call-site es reemplazado con el summary del método al que hace referencia. Si el programa resultante es correcto, debido a que todas las llamadas fueron sobre-aproximadas se puede concluir que el programa original P es *correcto*.

Si no se pudiese concluir en la etapa anterior que el programa es correcto, entonces se obtiene una prueba que involucra a algunos call-sites. Como consecuencia, dichas llamadas son *inlined* en P y se vuelve a comenzar.

Este proceso se repite hasta que se pueda concluir algo acerca del programa, o bien, que se alcance la cota de recursión.

En la siguiente sección explicaremos cómo se resuelven las queries derivadas del Algoritmo 1. Es importante notar que a diferencia de Blast y Code Contracts que utilizan únicamente sobre-aproximaciones, el hecho de que Corral utilice un enfoque híbrido exige rediseñar cómo deben ser creadas dichas queries. Para conocer en más detalle cómo fueron creadas las queries para Blast y Code Contracts, referirse a [8] y [25] respectivamente.

Capítulo 4

Implementación

En esta sección se describe como es utilizado Corral para proveer la funcionalidad requerida por el algoritmo de exploración del espacio de estados de la EPA descripto en la sección 3.2.1. En este sentido, se presentará una definición más concreta de las *queries* utilizadas y cómo es interpretado el resultado del *model checker* para que tenga el significado necesario.

4.1. Corral como motor de decisión

El primer paso en la integración de todos los componentes desarrollados en capítulos anteriores, es analizar en qué partes del algoritmo de creación de EPAs descripto en el Algoritmo 1, es necesaria la intervención de Corral.

El algoritmo tiene una naturaleza de exploración, esto quiere decir que la idea detrás del mismo es ir descubriendo los estados y transiciones que componen a la EPA. Para esto primero se necesita tener un mecanismo que permita, a partir de un estado cualquiera 's', determinar cuáles son los estados potencialmente alcanzables y luego en una etapa posterior verificar cuáles de las potenciales transiciones son efectivamente válidas.

Para la primera de las preguntas, el método es separar las acciones de la clase bajo análisis en dos conjuntos que son utilizados para encontrar estados potencialmente alcanzables. Para poder determinar qué elementos pertenecen a cada conjunto la propuesta en este caso es utilizar Corral. Dependiendo de cuál sea el conjunto a determinar vamos a recurrir a diferentes *queries* de *reachability* para caracterizarlo.

Determinar estados potencialmente alcanzables

De los dos conjuntos requeridos el más importante es el conjunto de las acciones habilitadas. Este caracteriza la respuesta a la pregunta: *¿qué acciones quedan* ***necesariamente*** *habilitadas si estando en un estado 's' ejecuto la acción 'a'?*.

Análogamente, el algoritmo sugiere encontrar el conjunto de acciones deshabilitadas ya que estas permiten recortar el espacio de búsqueda.

Dicha clasificación se logra evaluando cada acción de la clase mediante lo que llamaremos una query positiva y otra negativa. Las queries positivas intentan probar que desde 's' ejecutando 'a' siempre se puede probar que es válida la precondición de 'b', o en otras palabras, que la acción 'b' pertenece al conjunto de acciones habilitadas.

Formalmente, lo que se intenta encontrar es el siguiente conjunto de acciones:

$$B^+ = \{\ b \in A \mid \forall c, p.\ inv_s(c) \wedge R_a(c,p) \Rightarrow \exists p'.R_b(f_a(c,p), p')\}$$

Debido a que en .NET se cuenta con la biblioteca Code Contracts, en [25] se optó por reescribir las *queries* descriptas en [7] con el objetivo de hacer uso de la biblioteca y de Clousot para poder determinar si la acción cumplía con la condición deseada. Valiéndonos de esta modificación nuestro primer enfoque fue el de utilizar las mismas *queries* como input de Corral. A modo de ejemplo, se puede observar en el Algoritmo 2 el esquema de una query positiva.

Algoritmo 2 Query para determinar si una acción debe estar habilitada usando Code Contracts

```
procedure MustBeEnabledAction(s : State, a, b : Action)
    requires inv(s)
    ensures pre(b)
    call a()
```

Sin embargo, tal como están armadas las queries no es posible usarlas directamente con Corral. El primer obstáculo surge debido a las tecnologías empleadas, tal como se menciona en la sección 3.3 la entrada que recibe Corral es un programa escrito en Boogie. Por lo tanto, el primer paso necesario para poder utilizarlo consiste en traducir las queries escritas en .NET junto con sus respectivos contratos a código Boogie.

Para realizar la traducción del código recurrimos al traductor de bytecode de .NET a Boogie llamado BytecodeTranslator[1]. Lamentablemente, al momento de escribir esta tesis el traductor no es capaz de traducir los contratos escritos para .NET en contratos que decoren el código escrito en Boogie, aunque este último permita el uso de los mismos. Como consecuencia, es necesario modificar las queries generadas para poder traducir los contratos de forma satisfactoria. Para que las precondiciones y postcondiciones estén presentes en el código escrito en Boogie, se procedió al reemplazo de los **requires** y **ensures** por **assume** y **assert** respectivamente, como se puede apreciar en el Algoritmo 3. Notar, que también el orden de las instrucciones debe cambiar debido a que las postcondiciones, en forma de asserts, deben ser evaluadas al finalizar la ejecución de la query.

Algoritmo 3 Query para determinar si una acción debe estar habilitada

```
procedure MustBeEnabledAction(s : State, a, b : Action)
    assume inv(s)
    call a()
    assert pre(b)
```

Una vez que la traducción de la clase bajo análisis junto con las queries está terminada, se realiza una llamada a Corral por cada una de las queries. Esto se realiza de esta manera ya que Corral necesita que se le especifique un único *entrypoint* a partir del cual empezar a analizar cada vez que es ejecutado.

Es interesante notar que acá hay una diferencia con el uso de Code Contracts como *solver* ya que este último requiere de una única llamada para analizar toda una clase. En este sentido, Code Contracts actúa como caja negra para nuestro algoritmo y resulta más complejo tener un control más fino de la forma de resolución. Aunque éste cuenta con la posibilidad de verificar los métodos en forma paralela, es necesario especificarle a Code Contracts qué es lo que tiene que verificar ya que por defecto analiza todo el *assembly* recibido como entrada. Esto conlleva una penalización de performance que puede ser considerable teniendo en cuenta lo potencialmente grande que puede ser el espacio de búsqueda.

[1] Disponible en: `https://github.com/boogie-org/bytecodetranslator`

Como Corral es ejecutado como un proceso externo y dado que el resultado de la ejecución es mostrado por consola, se realiza un proceso de *parseo* del output del análisis de forma de obtener la conclusión a la que se arribó. Al contener un algoritmo de semi-decisión, los resultados a los que se puede llegar al analizar una query son:

- "True bug": se encontraron valores para las variables con los cuales halló una traza que hace fallar un *assert*
- "No bugs found": Corral pudo probar que no hay ejecución posible que haga fallar los *asserts*.
- "Reached Recursion Bound": no se encontraron bugs, sin embargo, tampoco se puede concluir que no existan. Esto se debe a que se alcanzó la cota de recursión que se está dispuesto a intentar.

Para poder preservar la semántica definida en [7] al utilizar Corral para ejecutar las queries de reachability, vamos a reinterpretar los resultados de la siguiente manera:

- "True bug" $\rightarrow$ Target reached
- "No bugs found" $\rightarrow$ Target unreachable
- "Reached Recursion Bound" $\rightarrow$ Target may be reachable

De esta forma, de acuerdo al algoritmo de construcción de EPAs de [7] se utilizan las siguientes reglas para determinar si una acción pertenece al conjunto de 'habilitadas':

- Target Reached: no se agrega la acción al conjunto de habilitadas.
- Target may be reachable: no se la agrega al conjunto de habilitadas y se deja asentado la imposibilidad de Corral de concluir un resultado.
- Target Unreachable: se agrega al conjunto de habilitadas.

Considerando las conclusiones de Corral de esta manera nos permite redefinir las queries, que inicialmente estaban pensadas para Blast, para incorporar el enfoque híbrido de Corral.

Análogamente, como fue mencionado anteriormente también se utilizan queries negativas. Estas son las que intentan ver que la negación de la precondición de 't' es cierta, sin importar los valores de las variables involucradas. En otras palabras, las acciones que seguro no van a poder ser ejecutadas. Formalmente,

$$B^{-} = \{\ b \in A \mid \forall c, p.\ inv_s(c) \wedge R_a(c,p) \Rightarrow \neg\exists p'.R_b(f_a(c,p), p')\}$$

Para las queries negativas utilizamos un esquema muy similar al de las positivas, tal como se puede ver en el Algoritmo 4. La única diferencia se puede observar en la condición que vamos a intentar analizar si es alcanzable o no, en este caso, la negación de la precondición de la acción 'b'.

Algoritmo 4 Query para determinar si una acción debe estar deshabilitada

```
procedure MustBeDisabledAction(s : State, a, b : Action)
    assume inv(s)
    call a()
    assert not(pre(b))
```

De esta manera, nos quedan definidos tres conjuntos disjuntos de acciones, las que siempre están *habilitadas*, las que necesariamente están *deshabilitadas* y el resto de las acciones, que no pertenecen a ninguno de los conjuntos anteriores. Esto nos permite resolver las instrucciones 5, 6, 14 y 15 del Algoritmo 1 de creación de EPAs. De esta forma, nos es posible definir todos los posibles estados alcanzables desde 's'.

Testear los estados candidatos

La segunda etapa que requiere de la intervención de Corral es cuando se quiere responder la pregunta: de todos los estados candidatos ¿cuáles son efectivamente alcanzables si estoy en el estado 's' y ejecuto la acción 'a'?

Al igual que en la etapa anterior vamos a recurrir al uso de queries de reachability para encontrar la respuesta. En este caso, se crea una sola query negativa por cada tupla de estado origen, acción a ejecutar y estado destino. El esquema utilizado para estas consultas es el descripto en el Algoritmo 5.

Algoritmo 5 Query para determinar si una transición debe ser agregada a la EPA

```
procedure FeasibleTransition(s : State, a : Action, t : State)
    assume inv(s)
    call a()
    assert ¬inv(t)
```

Utilizando el mapeo semántico descripto anteriormente, el resultado de la ejecución de Corral es interpretado siguiendo las reglas enumeradas a continuación:

- Target Reached: se considera a la transición como válida y por lo tanto se la agrega al EPA.
- Target May be reachable: la transición se la supone posible, ya que no hay un resultado concluyente, y se la agrega al EPA. Debido a la falta de certeza se la denota con un '?'.
- Target Unreachable: nunca vale el invariante del estado destino utilizando la acción a, por lo tanto, no se agrega la transición al EPA.

Una vez que todas las queries fueron ejecutadas y sus resultados obtenidos de la salida de la consola, se cuenta con un subconjunto de los estados candidatos obtenidos previamente. De esta manera, se cumple con las instrucciones 8 y 18 del Algoritmo 1, agregando todas las transiciones necesarias a la EPA junto con los nuevos estados descubiertos.

Extendiendo las queries para incorporar el soporte de parámetros

Hasta este punto se trabajó con la suposición de que las acciones no hacían uso de parámetros. Esta suposición propone una simplificación muy importante en las queries. Permitir la inclusión de parámetros requiere de un tratamiento particular de las condiciones ya que algunas de estas no pueden ser representadas como queries de reachability. Esto se debe a que su validez requiere poder construir una función que devuelva el valor de los parámetros para cada posible instancia. En [7] se propone una solución que requiere de la inclusión de predicados por parte del programador. Básicamente, la inclusión de estos predicados evitan el uso de parámetros mediante nuevas queries que usan sobre-aproximaciones y sub-aproximaciones de precondiciones. Nosotros optamos por un enfoque intermedio que permite no perder tanta precisión al utilizar parámetros bajo determinadas condiciones y que a la vez no utiliza los predicados propuestos en [7], ya que se delega la posibilidad de encontrar dichas funciones en el *model checker*.

Ejemplo 3

Veamos primero cuál es el problema que surge al utilizar parámetros en las queries.

Supongamos que estamos intentando determinar si la acción **IngresarDinero** de la clase **MaquinaExpendedora** está siempre habilitada luego de ejecutar **DarVuelto**, partiendo de un

Algoritmo 6 Ejemplo de query positiva con parámetros

procedure MustBeEnabledAction(s : State, $DarVuelto, IngresarDinero$: Action)
  assume inv(s)
  call DarVuelto()
  assert montoReconocido >0 ∧ !EstaVendiendo()

estado s. Tal como se describió anteriormente, la correspondiente query positiva sería como la que se puede visualizar en el Algoritmo 6.

Debido a que *montoReconocido* es un parámetro, el predicado *montoReconocido* > 0 del assert de la query no puede ser afectado por ninguna de las instrucciones anteriores. Por lo tanto, *montoReconocido* es una variable libre en el contexto del *assert* y es el camino más fácil que tiene Corral para poder encontrar un bug en el código. Simplemente, hallando algún valor para los parámetros que hagan falso el assert, es suficiente para que la query tenga un error. Luego, siempre que exista un parámetro las queries darán como resultado "Target reached" y de esta manera anulan cualquier tipo de información que se pudiera obtener de ellas. Es importante notar que la pérdida de información no afecta el hecho de que la EPA sea una sobre-aproximación, aunque si puede afectar la precisión de la misma. Sin embargo, estas queries tienen un impacto mucho más significativo en la performance. Perder información obliga a tener que generar más consultas para poder testear todas las posibles transiciones de la EPA. Dado que las llamadas al model checker son costosas, tener una mayor cantidad de consultas que realizar impacta seriamente en el tiempo necesario para poder resolverlas.

Para poder solucionar este problema vamos dividir a las precondiciones en dos categorías, las que predican únicamente sobre los parámetros y las que incluyen referencias a variables de la instancia. Para una clasificación más detallada, más precisa, pero dependiente de anotaciones referirse a [7].

Para las precondiciones que afectan exclusivamente a los parámetros vamos a asumir que existe una valuación de los mismos que las hacen verdaderas. Esta suposición se basa en que los contratos de todos los métodos deben ser satisfactibles. De todas formas, aunque estos no lo fueran, es fácilmente verificable a través de otra query de reachability.

Teniendo esto en cuenta, reescribimos las consultas de modo tal que los asserts que hagan únicamente referencia a precondiciones de parámetros pasen a ser assumes. De esta manera, Corral no podrá encontrar un error en parámetros que no cumplan la precondición y solo podrá encontrarlos en condiciones que afecten a las variables internas de la instancia. Luego, suponiendo que existen valores para los parámetros la query queda reescrita de la siguiente manera:

Algoritmo 7 Query extendida al uso de parámetros para determinar si una acción debe estar habilitada

procedure MustBeEnabledAction(s : State, a, b : Action)
  assume inv(s)
  call a()
  assume pre_de_param(b) ▷ los requieres de parámetros que son independientes del estado de la clase.
  assert pre_de_variables_internas(b) ▷ los requieres que hacen referencia a la comparación del estado con los parámetros y controles sobre el estado mismo.

Capítulo 5

Resultados

En este capítulo comentaremos sobre los aspectos involucrados en la validación de nuestra herramienta. Para ello primero presentamos algunos ejemplos básicos que nos servirán para analizar las fortalezas y debilidades de cada una de las implementaciones existentes. Luego, pondremos a prueba la eficacia de la nueva implementación de Contractor.NET comparándola con los resultados de su versión previa para casos de estudio más reales.

Esencialmente nos interesa responder las siguientes preguntas:

- ¿Cuáles son las fortalezas y debilidades de utilizar Corral versus las otras implementaciones?
- ¿Cómo se comparan en términos de precisión?
- Basado en el análisis de código .NET ¿cómo se comportan las versiones de Code Contracts y Corral para ejemplos reales y complejos?
- ¿Es este prototipo lo suficientemente eficiente para analizar los casos de estudio en un tiempo razonable?

5.1. Comportamiento de Corral en casos básicos

A continuación presentamos los experimentos realizados para determinar cómo varía la precisión de Corral a medida que el código a analizar se vuelve más complejo. Otro de los objetivos de estos casos de estudio es poder comparar los resultados obtenidos con todas las implementaciones existentes de generación de EPAs. De esta manera, esperamos obtener un primer indicio de la utilidad de utilizar Corral para generar EPAs.

Para llevar a cabo estas pruebas creamos una clase utilizando código C# (Figura 5.1) para la cual generamos diversas EPAs. Utilizamos las EPAs como representación del resultado ya que nos permite visualizar de una forma amigable las conclusiones a las que arriba Corral. Por otro lado, otra de las ventajas que ofrece utilizar las abstracciones es que nos brinda un lenguaje en común para poder comparar los resultados que se pueden obtener con las distintas implementaciones.

Es interesante notar que, aunque nuestros casos de estudio son lo suficientemente simples como para poder validarlos manualmente, esta metodología permite comparar resultados a mayor escala. Esto se debe a que es posible contrastar las EPAs obtenidas y obtener aquellos estados o transiciones que no pertenecen a la intersección de las mismas.

Con respecto a la clase utilizada para los experimentos, esta cuenta con un campo 'Estado' inicializado con el valor 0 en el constructor de la clase y dos métodos testigos. El objetivo de incluir

```
class TestCorral
{
    public int Estado;

    public TestCorral()
    {
        Estado = 0;
    }

    public void TestigoEstado1()
    {
        Contract.Requires(Estado == 1);
    }

    public void TestigoEstado10()
    {
        Contract.Requires(Estado == 10);
    }


}
```

Figura 5.1: Clase base utilizada para los experimentos

dos métodos testigos es poder visualizar en la EPA los posibles estados internos a los que pueden llegar las instancias.

A continuación veremos como, a partir de esta clase base, analizamos la incorporación de diversos métodos que presentan las variadas situaciones con las que nos podemos encontrar al analizar código. En particular, sesgamos los casos de estudios con el fin de buscar los límites de la precisión de CORRAL.

Con respecto a la configuración de hardware utilizada para realizar estos experimentos vale la pena mencionar que se utilizó una máquina con 1 core físico Intel Core i7 3.1 GHz junto con 4GB de RAM. En lo referente a la configuración de software se utilizaron todas las implementaciones con sus configuraciones por defecto. La única excepción es el *RecursionBound* de CORRAL que fue definido con un valor de 3.

Caso 1: Ciclo más chico que la cota de recursión

En el primer método que consideramos (Figura 5.2), evaluamos el caso más simple de todos. En este caso CORRAL debería ser capaz de recorrer el ciclo en toda su profundidad ya que la cota de recursión supera la cantidad de iteraciones totales del ciclo y por lo tanto poder recorrer todo el código del método.

Al tener una precondición que exige que el 'Estado' sea 0, el único resultado posible es que las instancias pasen a tener el 'Estado' en 1.

```
public void CicloMasChicoQueRecursionBound()
{
    Contract.Requires(Estado == 0);
    for (int i = 0; i < 1; i++)
    {
        Estado++;
    }
}
```

Figura 5.2: Código del experimento CicloMasChicoQueRecursionBound

Las tres implementaciones existentes lograron obtener la EPA esperada como se puede apreciar en las Figuras 5.3, 5.4 y 5.5. Sin embargo, en este caso empezamos a detectar la dificultad que demuestra Code Contracts para poder arribar a conclusiones concretas.

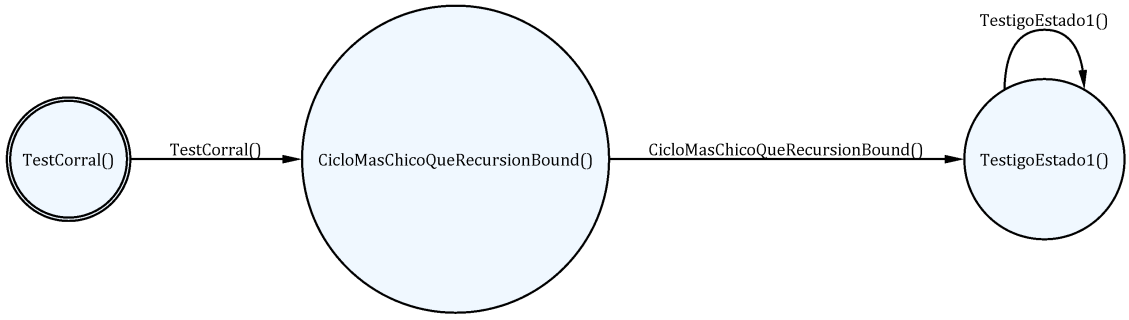

Figura 5.3: Resultado generado con Corral para CicloMasChicoQueRecursionBound

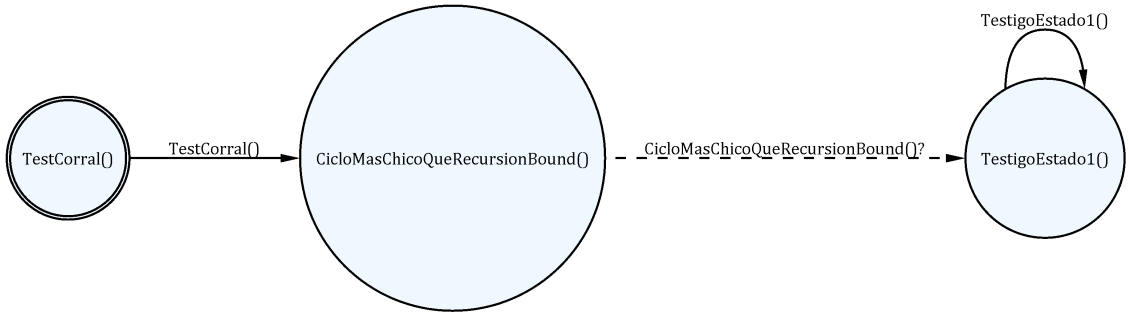

Figura 5.4: Resultado generado con Code Contracts para CicloMasChicoQueRecursionBound

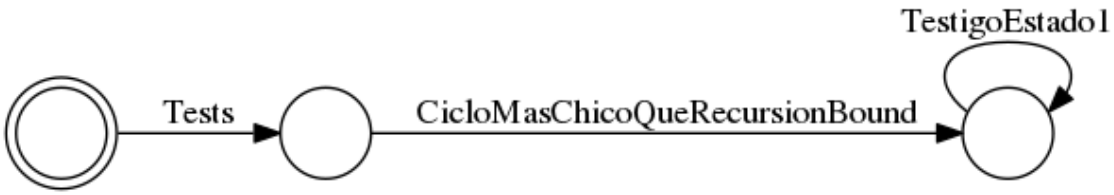

Figura 5.5: Resultado generado con Blast para CicloMasChicoQueRecursionBound

Caso 2: Ciclo más largo que la cota de recursión

Para agregar un poco más de dificultad, intentamos ver que sucede si la cota de recursión no es suficiente para analizar todo el código. Intuitivamente, este caso es similar al anterior. El único estado al que se puede ir, utilizando este método, es al que tiene habilitado únicamente el 'TestigoEstado10'.

```
public void CicloMasLargoQueRecursionBound()
{
    Contract.Requires(Estado == 0);

    for (int i = 0; i < 10; i++)
    {
        Estado++;
    }
}
```

Figura 5.6: Código del experimento CicloMasLargoQueRecursionBound

La EPA obtenida al utilizar CORRAL contiene una gran cantidad de transiciones que no pudieron ser confirmadas (Figura 5.7). Esto significa que CORRAL no fue capaz de determinar qué puede pasar con las instancias una vez ejecutado el método, si este cuenta con un ciclo cuya cota supera el límite de recursión que se está dispuesto a analizar. Sin embargo, lo que podemos aseverar en este caso es que el resultado que ofrece es sound, ya que todas las posibilidades están cubiertas por la abstracción obtenida.

Al igual que en el caso anterior CODE CONTRACTS no fue capaz de confirmar la transición que afirma la posibilidad de pasar al estado en donde la única acción habilitada es 'TestigoEstado10' (Figura 5.8). Es interesante notar que CODE CONTRACTS es capaz de obtener más información acerca del ciclo que la que pudo conseguir CORRAL, aunque esta no haya sido lo suficiente como para demostrar la transición.

Por su parte BLAST pudo determinar el resultado correcto del método, quedando como resultado la EPA esperada que se puede ver en la Figura 5.9.

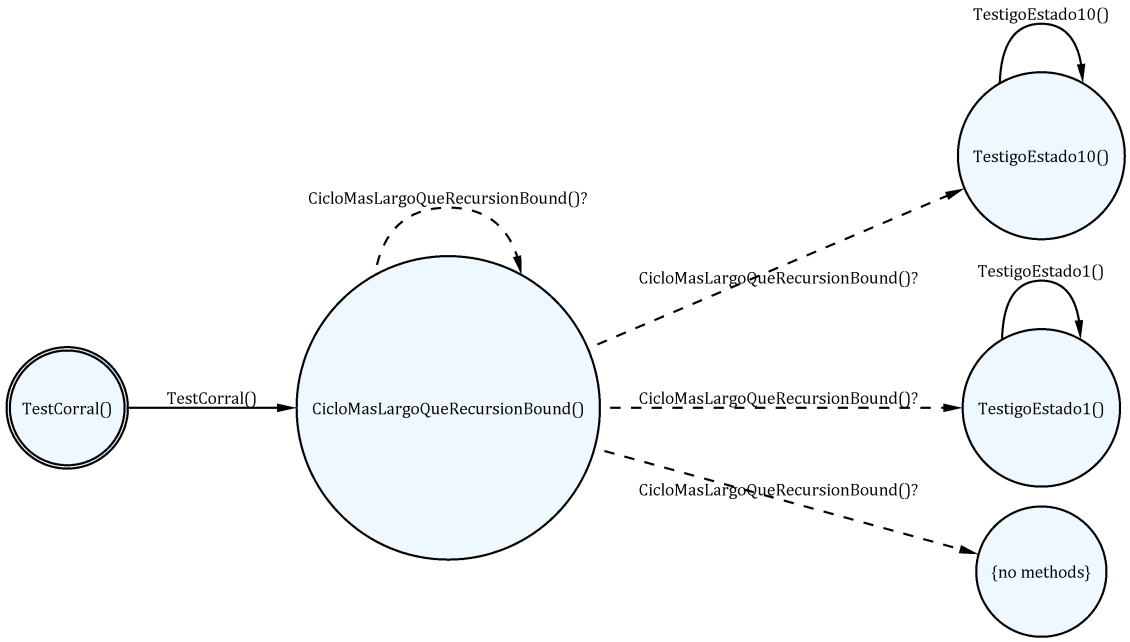

Figura 5.7: Resultado generando con CORRAL para CicloMasLargoQueRecursionBound

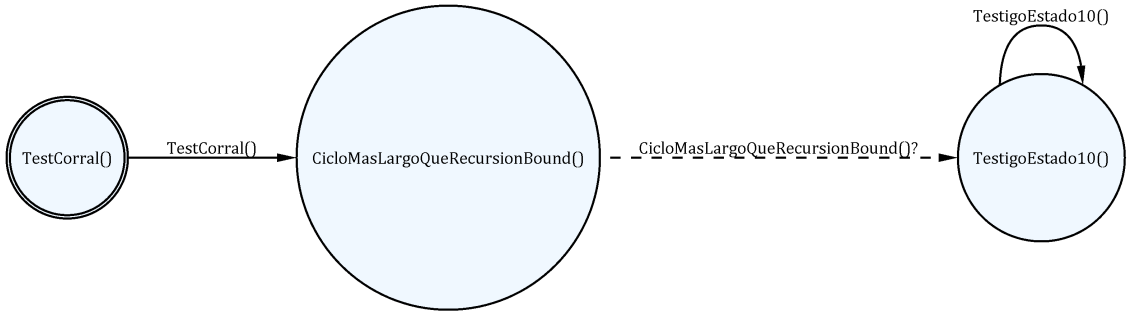

Figura 5.8: Resultado generando con CODE CONTRACTS para CicloMasLargoQueRecursionBound

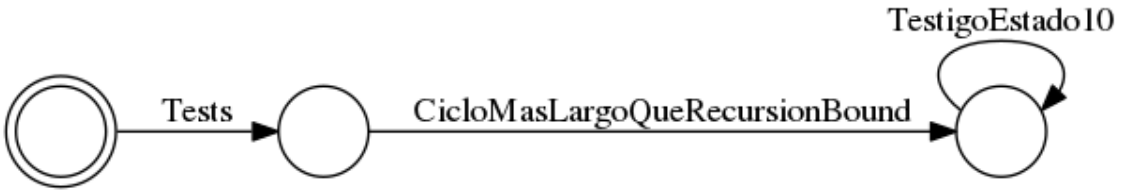

Figura 5.9: Resultado generando con BLAST para CicloMasLargoQueRecursionBound

Caso 3: Ciclo acotado por un parámetro (no determinismo)

¿Y si se introduce no determinismo en los métodos? ¿Cómo se comporta CORRAL con respecto al no determinismo? Con el código de la Figura 5.10 se prende analizar el caso en el cual la cota del ciclo depende de un parámetro y por consiguiente el resultado del ciclo es no determinístico.

```
public void CicloBasadoEnParam(int cota)
{
    Contract.Requires(Estado == 0);

    for (int i = 0; i < cota; i++)
    {
        Estado++;
    }
}
```

Figura 5.10: Código del experimento CicloBasadoEnParam

Como se puede ver en la Figura 5.11 CORRAL es capaz de encontrar valores de parámetros para aquellas transiciones que no exceden el límite de recursión definido. Al igual que en los casos anteriores, al no poder explorar ni inferir el resultado del ciclo cuando la cota es mayor al límite disponible de recursión, es conservador en su resultado.

En el caso de CODE CONTRACTS (Figura 5.12) la EPA obtenida es la esperada, es decir que hay valores para el parámetro que hacen posible llegar a cualquier valor de 'Estado'. Como lo logrado hasta el momento, se puede observar que es capaz de dejar únicamente las transiciones factibles aunque no tiene el poder suficiente como para confirmarlas.

Por su parte, BLAST detecta las mismas transiciones posibles que el resto de las tecnologías (Figura 5.13). Sin embargo, en este caso no es capaz de demostrar la veracidad de ninguna de ellas de forma fehaciente, al igual que CODE CONTRACTS. Este es el primer caso que encontramos en el cual el resultado de CORRAL es más preciso que el de BLAST dando evidencia de la factibilidad de mejora en la precisión de las EPAs generadas.

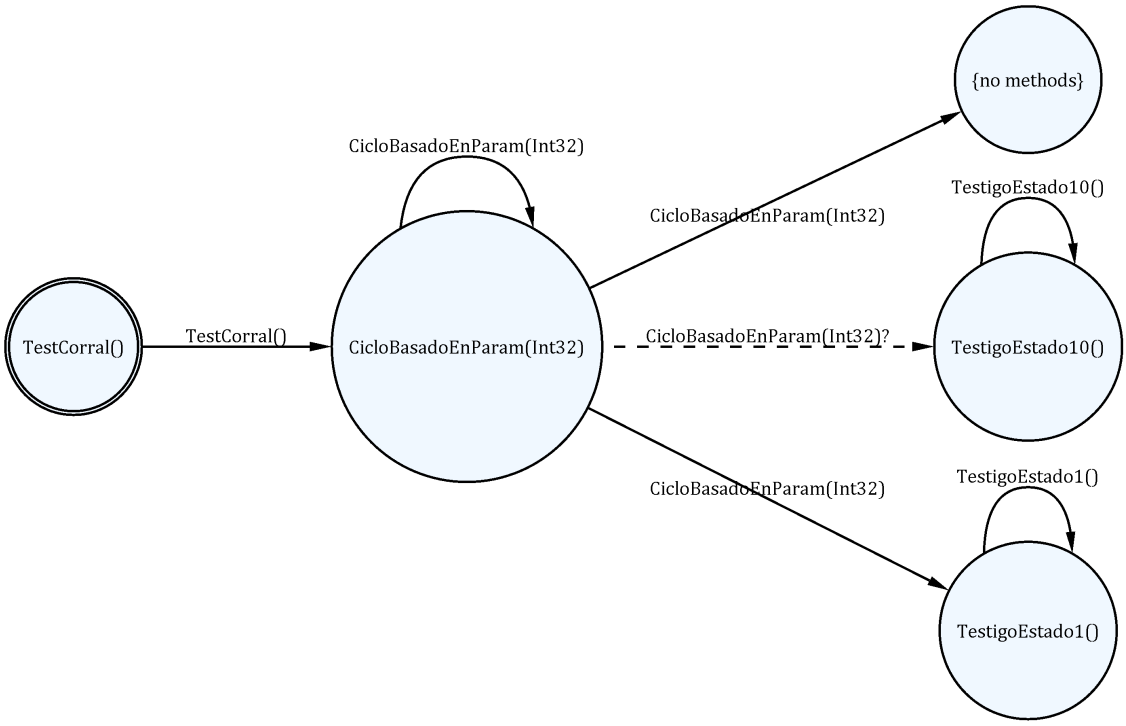

Figura 5.11: Resultado generando con CORRAL para CicloBasadoEnParam

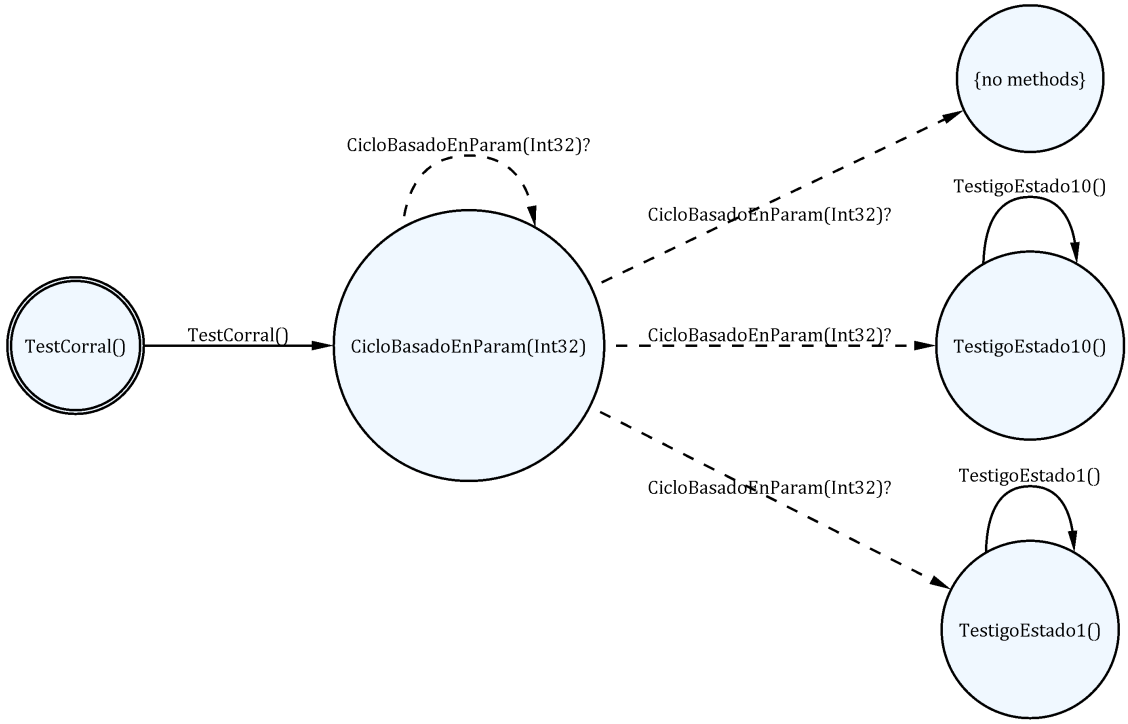

Figura 5.12: Resultado generando con CODE CONTRACTS para CicloBasadoEnParam

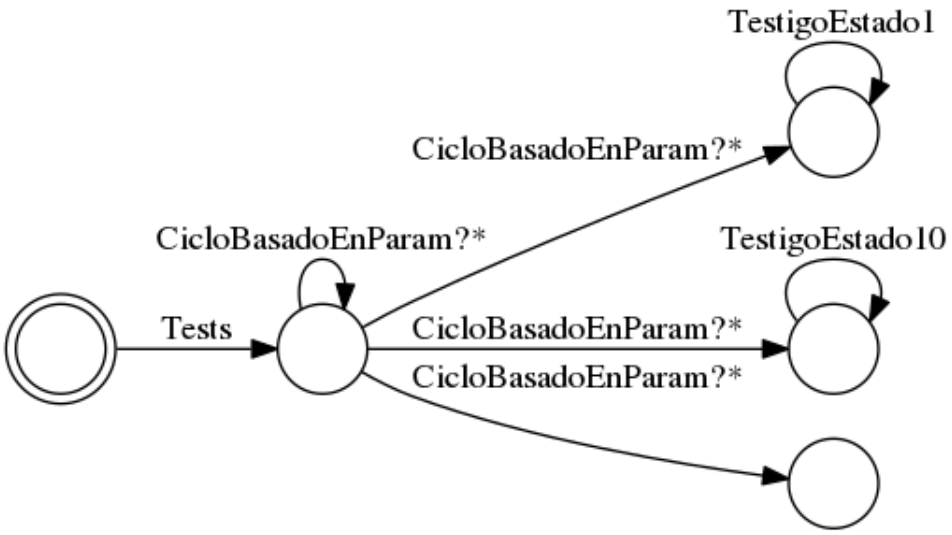

Figura 5.13: Resultado generando con BLAST para CicloBasadoEnParam

Caso 4: Ciclo con modificaciones espurias

Como pudimos ver previamente CORRAL tiene dificultades para analizar código que está más allá de la cota de recursión. Por este motivo, en este caso nos interesa ver qué pasa si las modificaciones que hace el ciclo son espurias. ¿Sigue siendo sound el resultado?

```
public void CicloDeberiaIrA10()
{
    Contract.Requires(Estado == 0);

    for (int i = 0; i < 30; i++)
    {
        Estado++;
    }
    Estado = 10;
}
```

Figura 5.14: Código del experimento CicloDeberiaIrA10

Al igual que en los casos anteriores el resultado sigue siendo sigue siendo una sobre-aproximación de la EPA esperada (Figura 5.15). En particular, es la EPA correcta aunque no se tenga certeza de una de sus transiciones. Es interesante notar que a pesar de no ser capaz de analizar en profundidad todo el ciclo, CORRAL tiene la posibilidad de inferir que las modificaciones que realiza el ciclo, de finalizar (al estar basado en BOOGIE no prueba terminación del ciclo), van a ser reescritas por la instrucción inmediatamente posterior.

A su vez, tanto CODE CONTRACTS (Figura 5.16) como BLAST (Figura 5.17) pudieron entender el código a la perfección dando una EPA 100 % precisa.

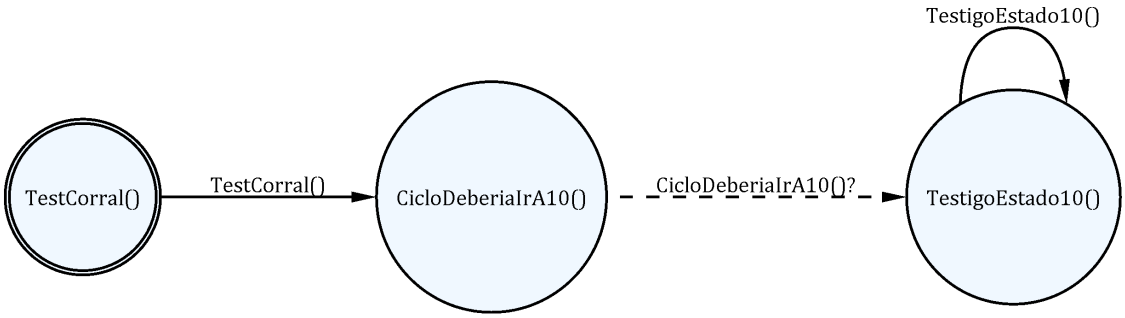

Figura 5.15: Resultado generando con CORRAL para CicloDeberiaIrA10

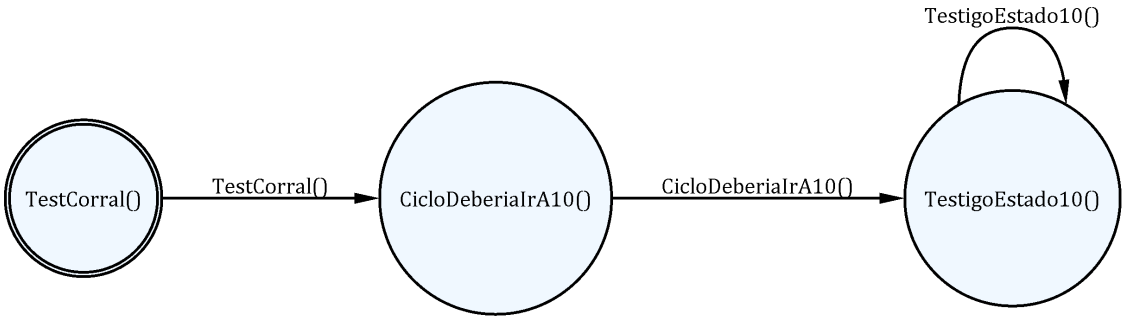

Figura 5.16: Resultado generando con CODE CONTRACTS para CicloDeberiaIrA10

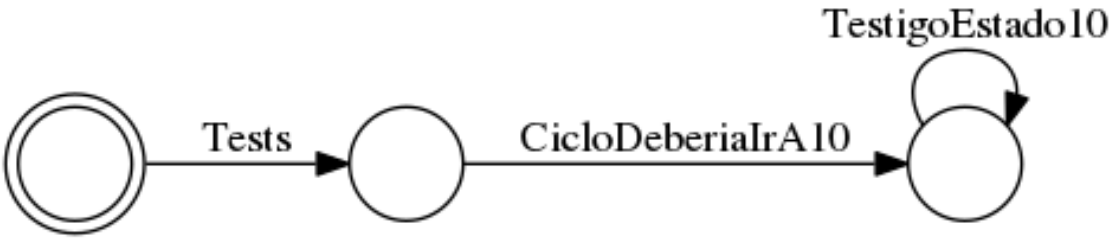

Figura 5.17: Resultado generando con BLAST para CicloDeberiaIrA10

Caso 5: Ciclo más largo que la cota pero de una sola iteración

Para hacer el análisis un poco más complejo nos preguntamos qué pasaría si el ciclo terminara en distintas iteraciones. ¿CORRAL llega a un resultado incorrecto? En los próximos casos lo ponemos a prueba con diferentes métodos.

En particular, a través del código de la Figura 5.18 utilizamos un ciclo que siempre termina en la primera iteración.

```
public void CicloNoDeberiaIrA10()
{
    Contract.Requires(Estado == 0);

    for (int i = 0; i < 30; i++)
    {
        return;
    }
    Estado = 10;
}
```

Figura 5.18: Código del experimento CicloNoDeberiaIrA10

En este caso, debido a que el ciclo termina siempre antes de sobrepasar la cota de recursión tanto CORRAL (Figura 5.19) como BLAST (Figura 5.21) pudieron llegar a la EPA esperada. Sin embargo, CODE CONTRACTS obtuvo el peor resultado conseguido hasta el momento (Figura 5.20). No sólo no fue capaz de pasar el obstáculo del ciclo que termina prematuramente, sino que incluyó en el resultado estados abstractos que no contienen ninguna instancia. Esto se debe a que no pudo demostrar que los invariantes de dichos estados son insatisfactibles.

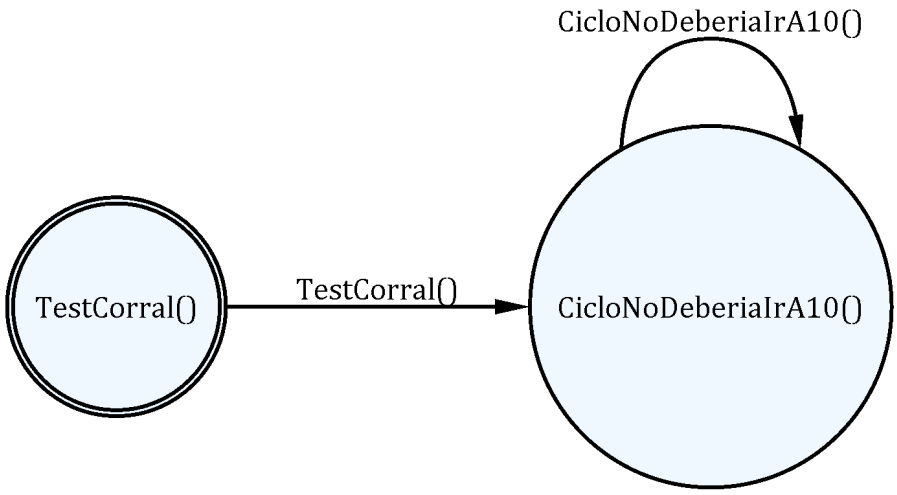

Figura 5.19: Resultado generando con CORRAL para CicloNoDeberiaIrA10

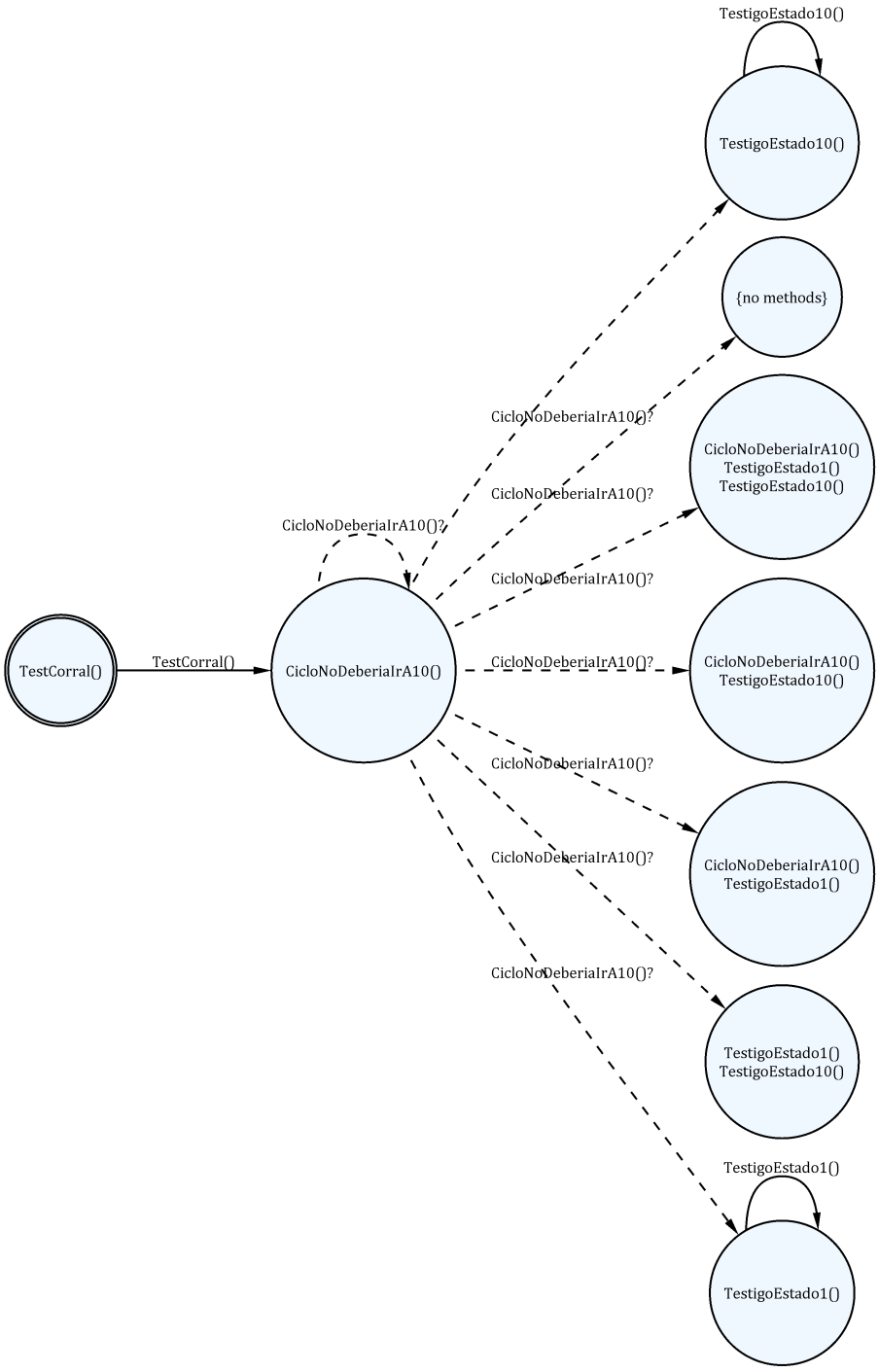

Figura 5.20: Resultado generando con Code Contracts para CicloNoDeberiaIrA10

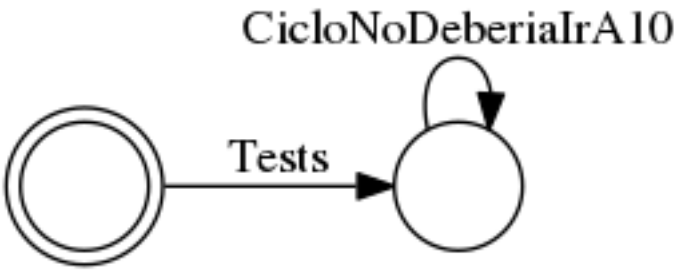

Figura 5.21: Resultado generando con BLAST para CicloNoDeberiaIrA10

Caso 6: Ciclo más largo que la cota que hace espuria la instrucción posterior

En este experimento repetimos la misma idea que en el anterior, terminando el ciclo antes de llegar a su condición de corte, pero haciendo que esto suceda más allá de la cota de recursión de CORRAL.

```
public void CicloNoDeberiaIrA10ConIf()
{
    Contract.Requires(Estado == 0);

    for (int i = 0; i < 30; i++)
    {
        if (i == 10)
            return;
    }
    Estado = 10;
}
```

Figura 5.22: Código del experimento CicloNoDeberiaIrA10ConIf

Claramente, tener la salida del ciclo en una iteración que no es posible analizar por CORRAL altera el resultado que puede concluir del análisis. Como se puede ver en la Figura 5.23 la EPA obtenida es la misma que en todas las demás oportunidades que no logra analizar en profundidad el código. Una sobre-aproximación de la EPA que se pudo obtener con máxima precisión a través del uso de BLAST (Figura 5.25).

Nuevamente, el hecho de que el ciclo no se complete parece afectar seriamente las conclusiones a las que puede arribar CODE CONTRACTS. La EPA obtenida (Figura 5.24) es una sobre-aproximación incluso más débil que la que se obtuvo con CORRAL.

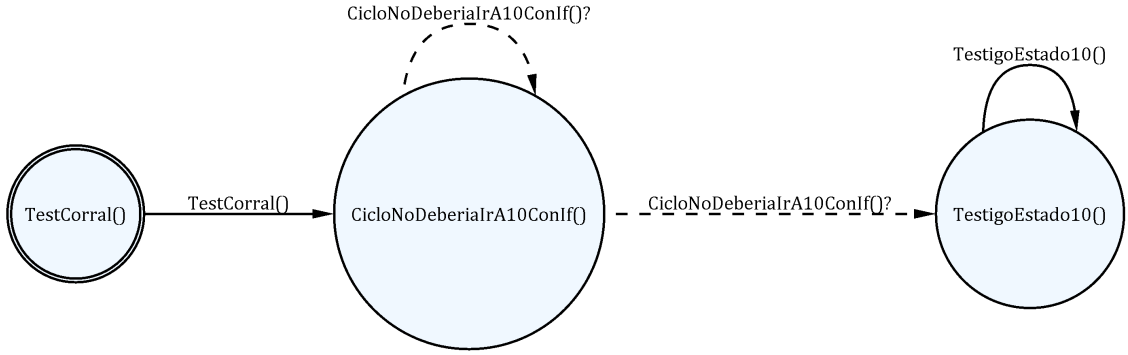

Figura 5.23: Resultado generando con CORRAL para CicloNoDeberiaIrA10ConIf

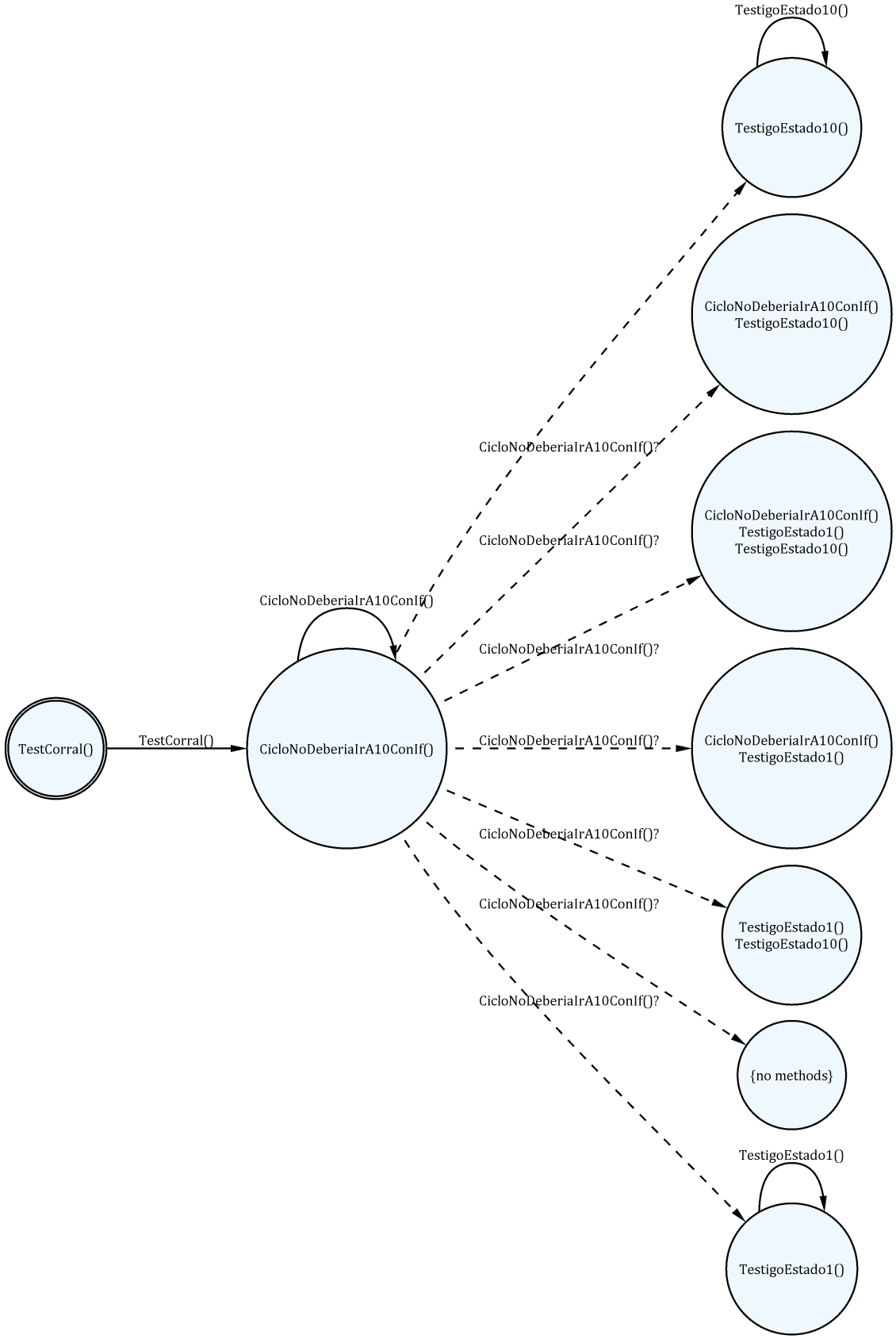

Figura 5.24: Resultado generando con Code Contracts para CicloNoDeberiaIrA10ConIf

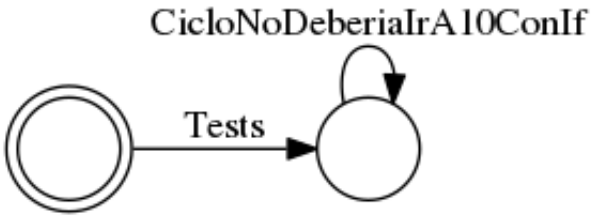

Figura 5.25: Resultado generando con BLAST para CicloNoDeberiaIrA10ConIf

Caso 7: Interacción compleja de estructuras de control de flujo

Por último decidimos probar una combinación más compleja entre estructuras de control. Intuitivamente, este caso pasa todas las instancias con 'Estado' == 0 a 'Estado' == 10. Sin embargo, aunque es muy simple para una persona reconocer este comportamiento, nuestras herramientas tienen que poder ser capaces de comprender la dependencia que existe entre el ciclo y el *if*.

```
public void CicloDeberiaIrA10ConIf()
{
    Contract.Requires(Estado == 0);

    for (int i = 0; i < 30; i++)
    {
        Estado++;
    }
    if (Estado == 30)
        Estado = 10;
}
```

Figura 5.26: Código del experimento CicloDeberiaIrA10ConIf

Nuevamente CORRAL no fue capaz de encontrar esta relación, debido a la longitud del ciclo, aunque de todas formas la EPA obtenida sigue siendo una sobre-aproximación razonable.

CODE CONTRACTS logró un mejor resultado que CORRAL, ya que pudo descartar uno de los estados abstractos inalcanzables. Notar que en este caso el ciclo no tiene una salida prematura.

Por último, BLAST puedo concluir que el código se comporta como nosotros esperábamos.

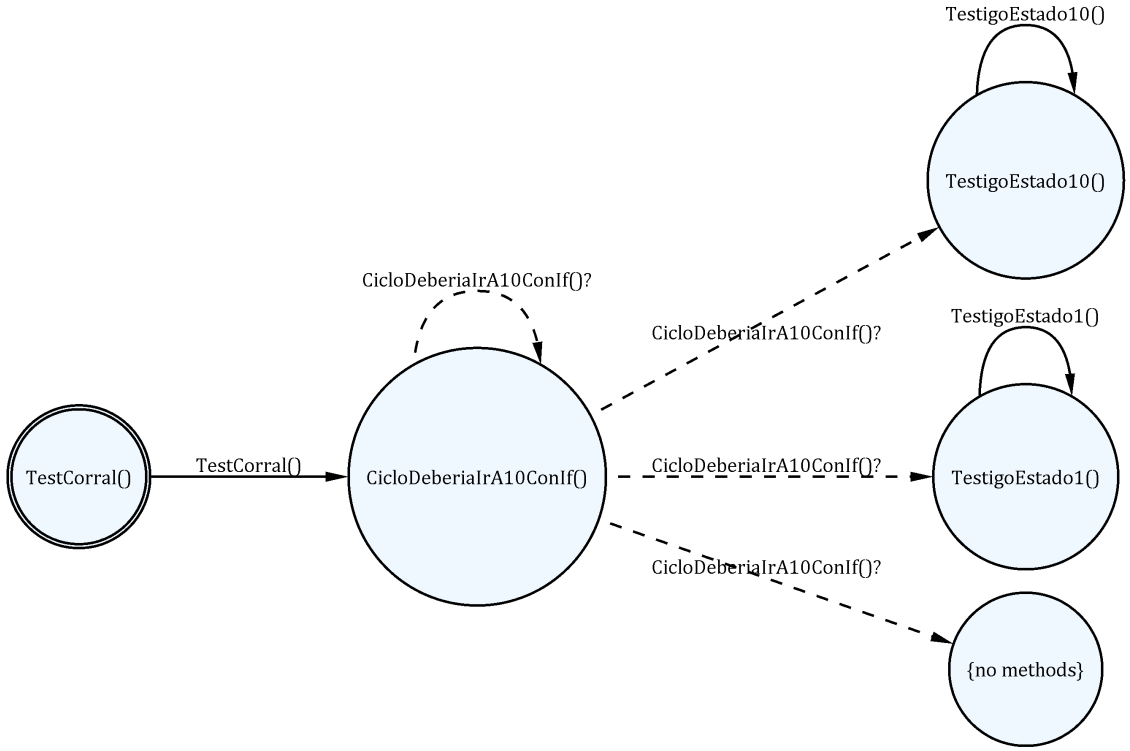

Figura 5.27: Resultado generando con CORRAL para CicloDeberiaIrA10ConIf

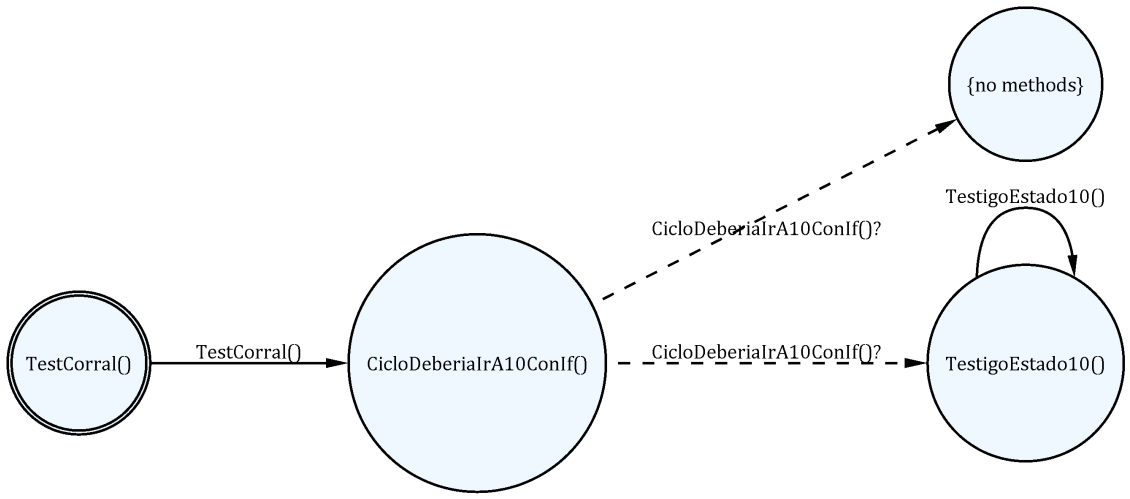

Figura 5.28: Resultado generando con CODE CONTRACTS para CicloDeberiaIrA10ConIf

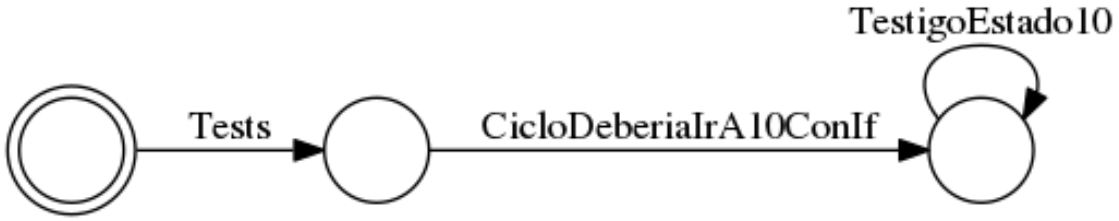

Figura 5.29: Resultado generando con BLAST para CicloDeberiaIrA10ConIf

Como resumen de los comportamientos observados en los diversos casos de estudio podemos aseverar que en todas las oportunidades los resultados de CORRAL fueron sound tal como era esperado. De la misma manera, las EPAs generadas tanto con CODE CONTRACTS como BLAST representan sobre-aproximaciones de las que uno puede construir manualmente.

Claramente los ciclos son un problema para todas las implementaciones. En el caso de CORRAL no contar con una cota adecuada para el análisis puede perjudicar considerablemente el resultado del análisis. La sobre-aproximación que utiliza por defecto para caracterizar aquello que no alcanza a analizar no llega a ser suficientemente robusta como para superar determinados obstáculos.

En el caso de CODE CONTRACTS el uso de ciclos causa una variabilidad importante en los resultados. Como pudimos observar, si el ciclo realiza todas las iteraciones hasta alcanzar la condición de la guarda, es capaz de inferir más información que la obtenida por CORRAL. Por el contrario, si el ciclo puede terminar en alguna iteración arbitraria, la precisión decrece significativamente consiguiendo resultados peores que en el peor caso de CORRAL.

En la mayoría de los casos BLAST es el que obtuvo las mejores abstracciones. A pesar de haber logrado superar la mayoría de los obstáculos los ciclos significan una penalidad en el tiempo necesario para poder analizar el código. Básicamente, el hecho de no contar con una cota que limite el tiempo de exploración facilita la posibilidad de hallar la respuesta correcta a cada query.

Vale la pena destacar que CORRAL pudo lograr los mejores resultados en presencia de no determinismo. Debido a que los métodos de las clases suelen tener parámetros, esta característica es prometedora para el uso de CORRAL en casos de estudio más complejos y más cercanos a la realidad.

5.2. CORRAL vs. CODE CONTRACTS

Habiendo realizado pruebas en una escala pequeña para comprender las fortalezas y debilidades del uso de CORRAL para el análisis de código seguimos con el estudio de ejemplos concretos. Recurrimos a los casos utilizados en [25] y en [7] junto con algunos ejemplos nuevos que no habían sido evaluados previamente[1]. El hecho de usar casos de estudio ya validados previamente nos permite tener un punto de comparación para poder responder los interrogantes que nos planteamos. Sin embargo, agregar algunos casos nuevos nos pareció interesante para poder enriquecer las situaciones evaluadas.

RQ1: ¿La precisión de las abstracciones generadas es adecuada para validar el modelo?

Quizá una de las preguntas fundamentales que toda implementación del algoritmo de generación de EPAs debe responder es si es capaz de generar EPAs precisas. Para poder contestar esto es necesario primero introducir una noción de precisión de las EPAs. Para esto vamos a recurrir a

[1] Los ejemplos y la implementación de la herramienta CONTRACTOR.NET están disponibles para ser descargados en: `http://lafhis.dc.uba.ar/contractor/contractor.net-web/`

la medida de precisión utilizada en [25]. Esto quiere decir que para determinar cuán precisa es una EPA vamos a considerar el porcentaje de queries generadas que tuvo un resultado concluyente.

Luego, procedimos a analizar nuestros casos de uso tanto con CORRAL como con CODE CONTRACTS obteniendo los resultados que se pueden observar en la Figura 5.30. Lo primero que sale a la luz es que utilizar CORRAL permite alcanzar la máxima precisión en la mayoría de las oportunidades. No es casual que en el caso del GenericStackSet<T> CORRAL pierda precisión. Dicho ejemplo, agregado en este trabajo, caracteriza un stack que no permite agregar elementos duplicados. Para poder llevar esto a cabo la estructura de datos utiliza un ciclo que controla que el elemento a agregar al stack no haya sido agregado previamente. Como era esperado, por los resultados obtenidos en los casos básicos, el uso de ciclos trae complicaciones a CORRAL.

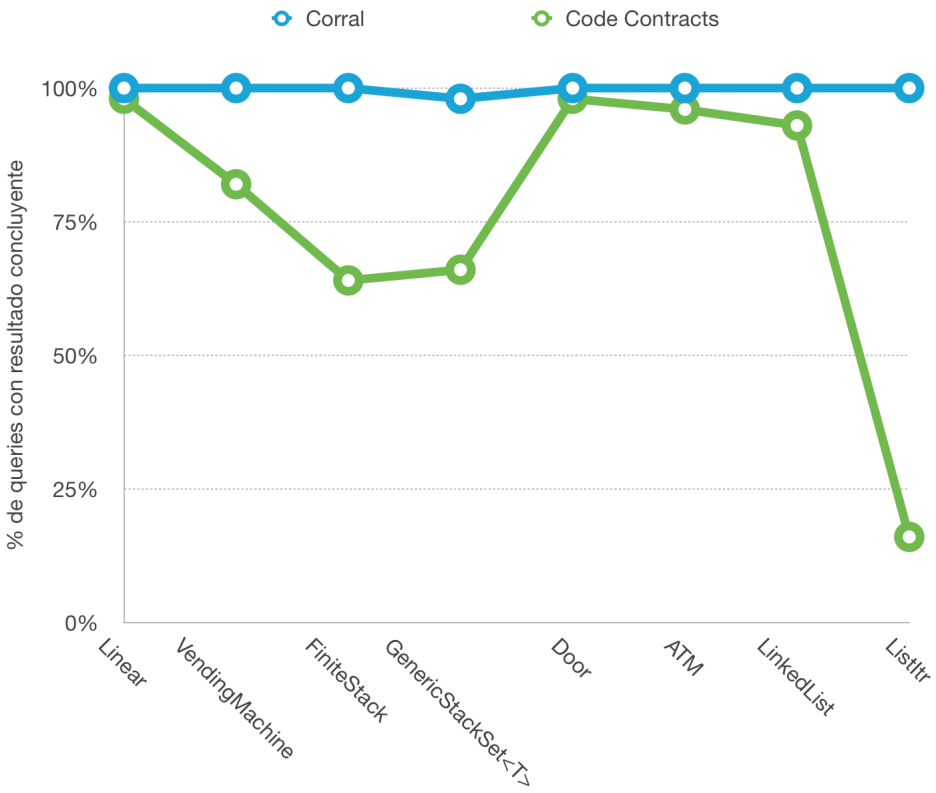

Figura 5.30: Comparación entre la precisión de CORRAL y la de CODE CONTRACTS

Por su parte, CODE CONTRACTS logró una exactitud de los resultados razonable en la mayoría de los casos. Naturalmente, los casos que no habían sido contemplados lograron perjudicar la precisión del resultado. Particularmente resalta el caso del ListItr. Este caso es la reescritura de la clase ListItr del JDK de Java, utilizado en [7]. Acá se hace notorio que el no utilizar un model checker impide que se puedan analizar casos de escala real y por ende más complejos.

RQ2: ¿Se generan las EPAs en un tiempo aceptable?

Es notable la exactitud con la que CORRAL puede resolver la gran mayoría de las consultas. Sin embargo, si el tiempo necesario para obtener tanta precisión crece de forma desproporcionada las ventajas de esta implementación no serían tan significativas. Por esta razón realizamos una comparación entre el tiempo de análisis necesario para conseguir las EPAs para nuestros casos de estudio.

Como se puede apreciar en la Figura 5.31 los tiempos de ejecución de ambas implementaciones son muy similares en un gran porcentaje de los casos evaluados. Aunque esta performance sin lugar a dudas es un gran acierto para CORRAL contra CODE CONTRACTS, nos lleva a preguntarnos por qué es que esto sucede. Para encontrar la respuesta es necesario analizar con más detalle cada ejemplo.

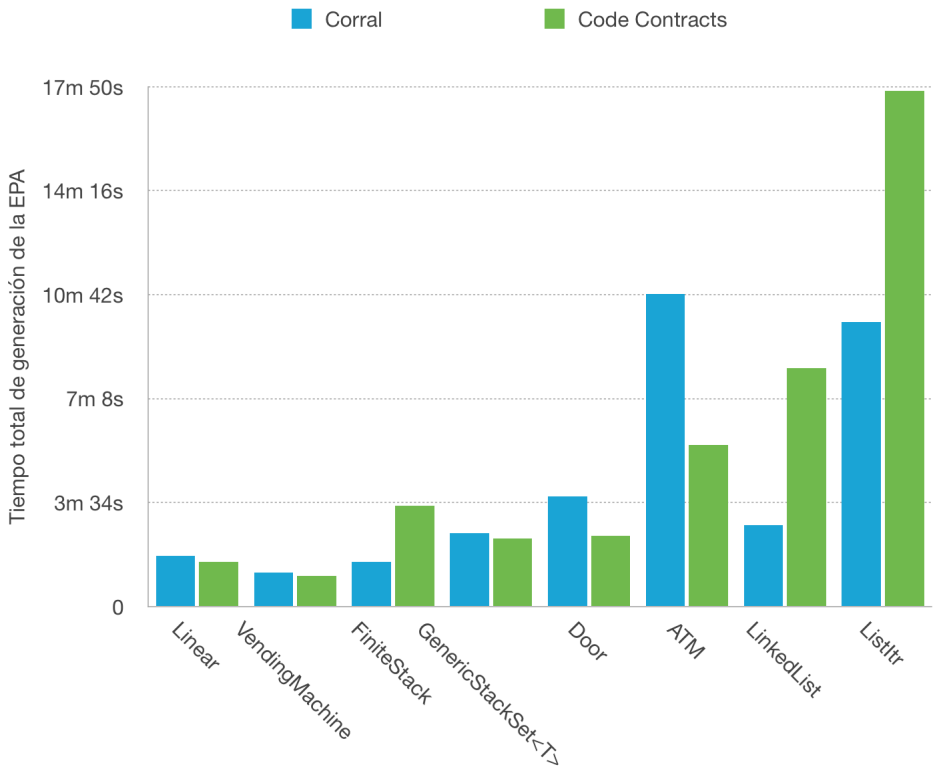

Figura 5.31: Comparación entre el tiempo de análisis de CORRAL y el de CODE CONTRACTS

Los primeros dos casos 'Linear' y 'VendingMachine' son demasiado sencillos y pequeños como para que se acentúen diferencias en los analizadores. Más aún, en ambos casos los dos analizadores obtuvieron porcentajes de precisión muy altos.

Más interesantes para analizar son las dos representaciones de stack 'FiniteStack' y 'GenericStackSet<T>'. En estos casos se visualiza una baja considerable en la precisión de la EPA generada por CODE CONTRACTS.

Si bien estos ejemplos representan un incremento de dificultad, siempre manteniendo una escala chica, el aumento de interacción interprocedural empieza a dificultar la tarea de CODE CONTRACTS. Esto se debe a que inicialmente, el objetivo de CODE CONTRACTS como tecnología fue la demostración formal intraprocedural. Aunque posteriormente fue incorporada la posibilidad de realizar demostraciones utilizando un análisis interprocedural, la interacción entre distintos métodos es una debilidad de la tecnología. Por esta razón, las precisión de CODE CONTRACTS cae considerablemente. Sin embargo, se puede visualizar en el caso de 'GenericStackSet<T>' que el hecho de contar con un análisis más liviano, y la inferencia de invariantes, permite equiparar la penalidad por la precisión que paga CORRAL.

Prestemos atención en este momento a los casos 'Door' y 'ATM'. Estas clases son interesantes ya que muestran por primera vez ejemplos en donde el tiempo de ejecución de CORRAL supera significativamente al de CODE CONTRACTS. En primer lugar, reafirmando lo mencionado anteriormente, al acceder directamente a los campos de la clase, en vez de a través de una propiedad

ayuda considerablemente a CODE CONTRACTS. Notar que la precisión del análisis vuelve a índices muy altos. En segundo lugar, las EPAs resultantes de ambos ejemplos tienen un alto nivel de conectividad. La interacción entre los métodos es elevada y por lo tanto la cantidad de queries analizadas empieza a ser un factor determinante en el tiempo de ejecución. Naturalmente, las llamadas al SMT solver tienen como contrapartida un tiempo de ejecución más largo.

Por último se encuentran 'LinkedList' y 'ListItr', dos implementaciones cercanas a lo que uno se puede encontrar al analizar código. Estas son las clases en las cuales CORRAL logra la mayor diferencia con respecto a CODE CONTRACTS, dando pruebas concretas de los beneficios de utilizar CORRAL para la construcción de las EPAs. El hecho de lograr una precisión tan baja en el caso de 'ListItr' provoca que se sobre-aproxime demasiado la EPA aumentando considerablemente la cantidad de queries a resolver. Aún siendo mucho más ligero para contestarlas, esto es un factor determinante en la construcción de las EPAs.

RQ2.1: ¿Es beneficioso el uso de paralelismo con los analizadores existentes?

Los resultados visualizados anteriormente son los mejores obtenidos al utilizar ambas implementaciones. Dado que con el objetivo de reducir el tiempo necesario para construir EPAs, realizamos una implementación parcial de una estrategia paralela, es necesaria una evaluación de los resultados logrados para determinar la eficiencia de la misma.

Para esto, realizamos una comparación entre los tiempos necesarios para obtener las EPAs usando la versión secuencial y la nueva implementación paralela. Debido a que la inclusión de procesamiento paralelo impacta tanto sobre el algoritmo general de creación de EPAs como también a la ejecución propia del model checker es necesario analizar ambos ejes.

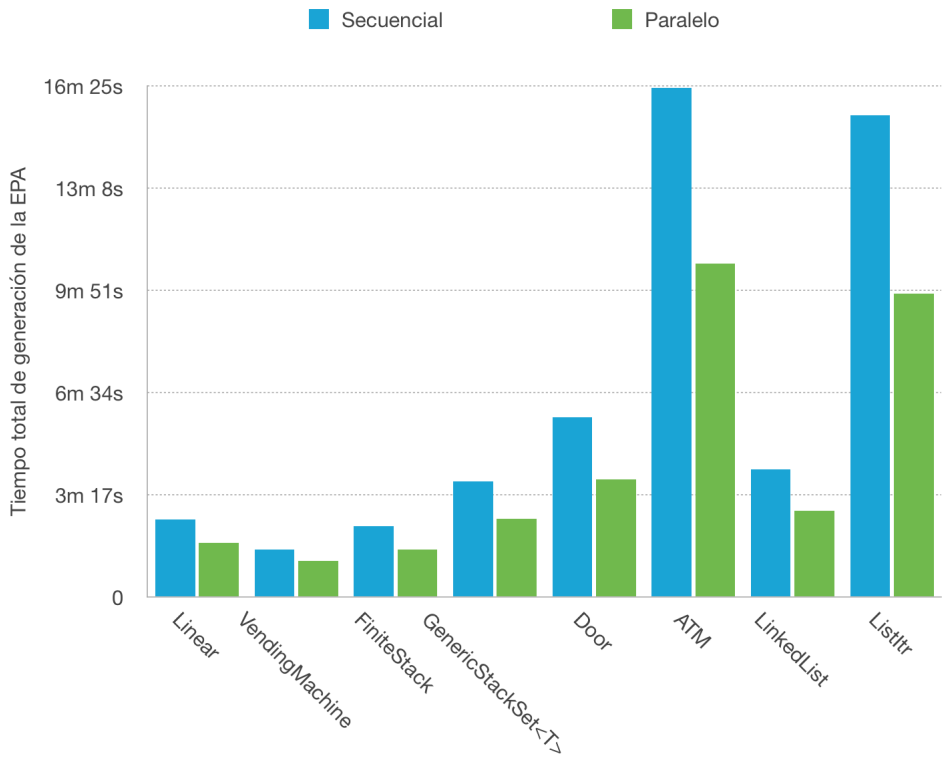

Figura 5.32: Comparación entre la versión secuencial y la paralela usando CORRAL

Al observar los tiempos necesarios para obtener las EPAs al usar CORRAL tanto en su versión

secuencial como paralela (Figura 5.32) notamos una reducción significativa de los valores. Sin dudas, esto marca un claro beneficio en la incorporación de paralelismo al utilizar CORRAL. Sin embargo, dado que CORRAL hace uso del paralelismo en los dos ejes mencionados previamente, es necesario evaluar los efectos de los cambios en el algoritmo general. En este sentido, repetimos el experimento utilizando CODE CONTRACTS ya que únicamente hace uso de las modificaciones en el algoritmo general.

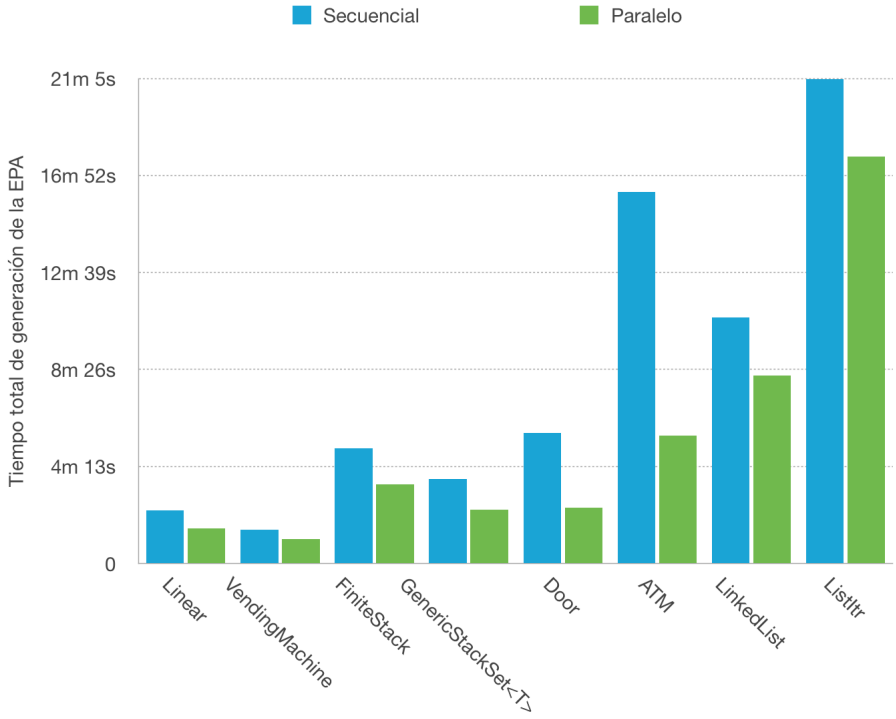

Figura 5.33: Comparación entre la versión secuencial y la paralela usando CODE CONTRACTS

Afortunadamente, las mismas conclusiones aplican para el algoritmo general. En la Figura 5.33 se vuelve a repetir los observado al utilizar CORRAL. En todos los casos, se redujo considerablemente el tiempo necesario para obtener la EPA. Esto nos permite afirmar que más allá de los costos extras inherentes a la programación paralela, esta nos abre posibilidades de continuar mejorando la eficiencia de este prototipo.

RQ3: ¿Esta nueva implementación es lo suficientemente robusta como para manejar programas complejos?

Por último, es interesante discutir acerca de la capacidad de este prototipo de procesar la gran variedad de clases que existe. Como referencia, en la Figura 5.34 podemos ver una versión compacta y agregada de los resultados previos.

Es indudable el poder de CORRAL como model checker. Tanto en los casos básicos descriptos anteriormente como con los ejemplos concretos considerados obtuvo resultados satisfactorios. En particular, para estos últimos pudo conseguir una precisión virtualmente máxima. Aún así, en algunas ocasiones el costo del model checker es mayor al necesario para el análisis. En estos casos, gracias a las características de CODE CONTRACTS, podemos lograr resultados igualmente precisos sin tanto overhead de análisis. Por lo tanto, si bien no podemos decir que CORRAL sea una alternativa completamente superadora, consideramos que se complementa muy bien con la

implementación previa.

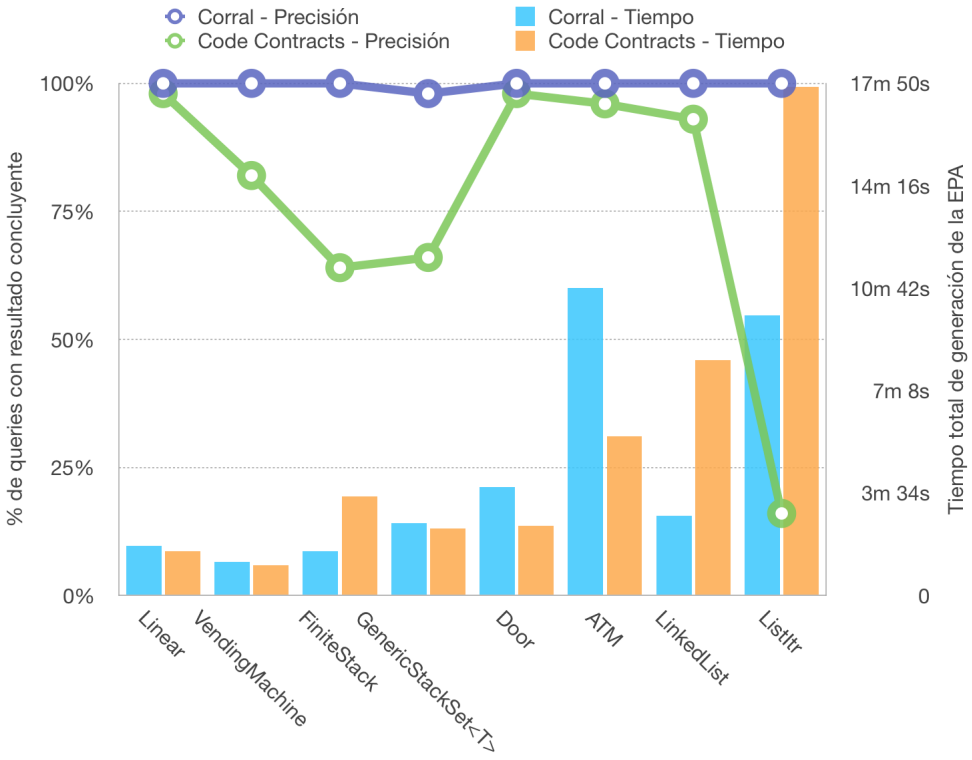

Figura 5.34: Relación entre la precisión y tiempo de generación de la EPA

Con todos los avances desarrollados para está nueva versión del prototipo Contractor.NET logramos alcanzar el estudio de casos reales en tiempos razonables para la complejidad inherente a las consultas involucradas. Desde luego, todavía hay muchas posibilidades de lograr mejores resultados, de todas formas, la combinación de herramientas de resolución define un hito en el manejo de programas complejos.

5.3. Limitaciones conocidas

Si bien en varias oportunidades logramos obtener una EPA de máxima precisión esto no garantiza que las abstracciones resultantes acepten únicamente las trazas del código. A continuación discutimos algunos de los factores que pueden contribuir a una sobre-aproximación en la EPA.

Recordemos que para obtener una EPA necesitamos tanto del código de la implementación de la API como también de la especificación con los contratos que definen formalmente el comportamiento esperado. Más aún, también podría ser que el model checker no sea lo suficientemente poderoso como para poder concluir al analizar las queries. Por lo tanto, ante la aparición, o ausencia, de una transición sospechosa la tarea de la revisión puede deberse a diversos factores. ¿Es un problema en el código? ¿El contrato no refleja correctamente la semántica de la implementación? ¿El model checker no es lo suficientemente poderoso? ¿El algoritmo de construcción tiene una incidencia significativa? Intentemos analizar cada uno de estos interrogantes.

Relación código / contratos

Idealmente, los contratos definen de forma precisa, correcta y completa la semántica del código que representan. De ser así, uno podría determinar la existencia de una transición revisando indistintamente el código o los contratos. Sin embargo, en la práctica esto suele ser muy poco frecuente.

Supongamos que la precondiciones que estamos utilizando son más débiles de lo que en realidad podrían ser. En este caso, dado que el resultado de las queries que testean la validez de una transición (Algoritmo 5) depende del invariante del estado al que llega la transición. Y considerando que el invariante de estado depende de las precondiciones de los métodos habilitados y deshabilitados, tener precondiciones muy débiles contribuyen a que una mayor cantidad de queries resulten como afirmativas y pasen a formar parte del resultado. Esto sucede ya que se analizan únicamente los contratos del objetivo.

En el caso de ser muy fuertes van a imponer mayores restricciones a las potenciales instancias que podrían hacer uso de los métodos y por lo tanto la ausencia de una transición podría deberse a un contrato demasiado restrictivo. En este caso, habría que preguntarse si es que el código contempla casos innecesarios o si la semántica que se pensaba para el método es demasiado reducida.

Por último, tenemos que considerar el invariante de la clase. El invariante de un estado abstracto también se compone del invariante de la clase. Este define una condición global sobre las instancias que representan al tipo y también afecta al testeo de las transiciones, si varias transiciones resultan sospechosas el típico candidato suele ser el invariante.

Las limitaciones del model checker

El model checker tiene un rol preponderante en la posibilidad de lograr una buena precisión ya que es quien determina la veracidad de las queries.

En nuestro caso particular, al utilizar Corral para analizar código .NET estamos sujetos a una limitación que puede afectar nuestro análisis, el mismo framework .NET. Debido a que Corral entiende código Boogie es necesario traducir la implementación de la API, como ya fue mencionado previamente. Sin embargo, para tener un conocimiento completo de la semántica del programa habría que traducir todas aquellas APIs que son utilizadas por el mismo pero que son ajenas a nuestro control. El más usual de estos casos se debe a la interacción con el framework. El no contar con información sobre estas llamadas a métodos "desconocidos" hace que para seguir conservando la propiedad de soundness es necesario asumir que cualquier cosa puede pasar en dicha llamada, llevando en un gran porcentaje de estos casos a una explosión de estados perjudicando notablemente el resultado. Un workaround que utilizamos en estos casos es hacer explicito el contrato del método mediante un *assume* y *assert*. De esta manera, forzamos un summary preciso del resultado de la ejecución. Típicamente la explosión de estados es causado por esta limitación.

Sobre la precisión de la abstracción

La definición de las EPAs, en particular el método de decisión para agregar transiciones, determina una forma de mirar las transiciones que lleva potencialmente a sobre-aproximaciones debido al agregado de transiciones espurias. Básicamente, la definición asume que solo alcanza con conocer los dos estados involucrados por la transición para determinar si pertenece o no a la abstracción. Esta hipótesis no siempre es cierta como mencionamos previamente. La falta de registro de la historia conlleva a que un estado abstracto pueda representar muchos más estados concretos que a los que realmente se llegaría si se aplicara la secuencia de métodos necesaria para arribar a dicho estado abstracto. Veamos un ejemplo.

Intuitivamente podemos observar que el código de la Figura 5.35 describe un protocolo lineal.

```
public class Test2
{
    public int x;
    public int y;

    public Test2()
    {
        x = 0;
        y = 0;
    }

    public void m1()
    {
        Contract.Requires(x == 0 && y == 0);
        x = 5;
    }

    public void m2()
    {
        Contract.Requires(x > 0 /* x == 5 */ && y == 0);
        y = 7;
    }

    public void m3()
    {
        Contract.Requires(x == 5 && y == 7);
        x = 3;
        y = 3;
    }
}
```

Figura 5.35: Código con precondiciones débiles

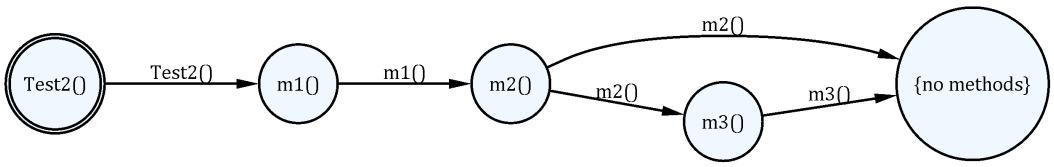

Figura 5.36: EPA con estados con invariantes débiles

Está pensado para que los métodos se ejecuten de manera secuencial. Sin embargo, como se puede observar en la EPA generada el resultado no es el esperado. Esto se debe a que la precondición del método **m2** es más débil de lo que podría ser.

Una forma de mitigar este problema de precisión sería mediante el refuerzo del invariante de la clase para reducir la cantidad de estados posibles considerando la historia. Esto requiere de la revisión manual de la EPA y el protocolo para determinar cuáles son las condiciones necesarias para incorporar.

Otra alternativa para poder recuperar la información perdida es utilizar la técnica de Stratified Inlining implementada en CORRAL para ratificar transiciones que pueden ser ejecutadas por alguna instancia o removerlas en caso contrario.

Para este caso ideamos una segunda etapa para el algoritmo BFS que considera a la EPA como si fuese un Control Flow Graph e intenta hacer fallar un assert por cada transición en la EPA. A continuación mostramos el pseudocódigo relacionado con la segunda etapa de análisis.

Algoritmo 8 Transformar los estados de la EPA en métodos

```
1: procedure ConvertStatesToMethods(states:[State])
2:     for each state s ∈ states do
3:         CreateStateMethod(s)
```

Algoritmo 9 Template del método de cada estado

```
1: procedure S##( )
2:     a ← Random(0, #enabledActions(s))
3:     if a == 0 then S##_A( )
4:     if a == 1 then S##_B( )
5:     ...
6:     if a == #enabledActions(s) then S##_K( )
```

Algoritmo 10 Template del método de estado S01 usando la acción A

```
1: procedure S01_A( )
2:     Create random instances for each parameter (a_0, a_1, ..., a_n) of the method A
3:     A(a_0, a_1, ..., a_n)
4:     for each state t ∈ targets(S01, A) do
5:         if inv(target) then
6:             assert < UniqueNumber > != id
7:             S#target()
```

Si analizamos la EPA de la Figura 5.36, observaremos que al testear la transición, que sabemos que es espuria, CORRAL es capaz de probar que no hay errores en el programa. Para todas las demás transiciones es capaz de hacer fallar el assert que identifica a cada transición.

Realizamos pruebas codificando manualmente las EPAs obtenidas de algunos de los ejemplos utilizados previamente logrando resultados satisfactorios.

Quizá lo más interesante es que este algoritmo abre la puerta a la utilización de CORRAL para generar una test suite cuyo criterio de cobertura sea la cantidad de transiciones de la EPA. Esto se debe a que si es posible obtener los valores y el orden de ejecución de los métodos de la traza eso nos define un caso de test que garantiza cubrir una transición en particular.

Capítulo 6

Conclusiones

En este trabajo presentamos un nuevo prototipo de Contractor.NET, una herramienta que permite construir de manera estática y automática modelos abstractos de comportamiento. Las abstracciones son generadas, bajo la forma de typestates, utilizando un nivel de abstracción denominado *enabledness-preserving*. Dicho criterio ha demostrado expresar de forma concisa y representativa el comportamiento que las clases tendrán en tiempo de ejecución.

Para desarrollar el prototipo hemos reinterpretado el algoritmo de construcción de las abstracciones para utilizar Corral como motor de decisión para resolver las queries necesarias.

Adicionalmente, mostramos cómo se pueden utilizar estas abstracciones para validar y facilitar el entendimiento del código fuente subyacente por parte del programador, posibilitando la identificación de problemas existentes en el mismo.

Con respecto a la precisión y performance, realizamos una comparación de todas las implementaciones existentes reconociendo fortalezas y debilidades para las mismas. En este sentido, también pudimos evidenciar mejoras considerables con respecto a la capacidad de obtener resultados más precisos. No menos importante, demostramos la posibilidad de realizar mejoras significativas respecto al tiempo requerido para generar las abstracciones para ambas implementaciones de .NET.

Por último, discutimos sobre algunas limitaciones respecto a la definición de las abstracciones y a las tecnologías utilizadas para mejorar aún más las precisión de los modelos generados.

Consideramos que este tipo de abstracciones pueden ser útiles como documentación adicional de las APIs, ya que permiten expresar información adicional, generalmente no documentada, como es el caso de las interacciones entre los distintos métodos de las clases. Además, cuentan con la ventaja de poder calcularse de forma automática, directamente a partir del código fuente, por lo que no quedan desactualizadas con el transcurso del tiempo como suele pasar comúnmente con otros tipos de documentación. Por otro lado, aseguran una gran precisión y correspondencia con la implementación real, ya que no están basadas en los requerimientos, que la gran mayoría de las veces son inexistentes o poco precisos.

Capítulo 7

Trabajo futuro

Son muchas las mejoras y extensiones que se pueden realizar al prototipo de este trabajo. Desde un punto de vista más ingenieril, todavía queda lugar para mejoras en cuanto a la implementación paralela. Aunque se lograron avances en este sentido en la versión actual, los estados descubiertos por el algoritmo siguen siendo analizados secuencialmente. Esto definitivamente permitiría seguir bajando los tiempos de ejecución del algoritmo al hacer un uso más intensivo de los recursos de la computadora.

Otra de las etapas que todavía tiene muchas oportunidades de mejora es la de traducción. Claramente, este es una de las dependencias más sensibles del proceso. Para poder seguir ampliando el espectro de código que es posible analizar es necesario trabajar en las capacidades que tiene el traductor. Una de las alternativas es hacer un uso más fino de las funcionalidades que provee Boogie. Otra posibilidad es limitar el uso de la traducción y realizar la manipulación de las queries desde Boogie.

Quizá más interesante es la posibilidad de enriquecer las EPAs construidas para proveer información de mayor calidad.

Actualmente se cuenta con una gran cantidad de código disponible en Internet, desde preguntas en sitios de Q&A hasta repositorios de código abierto. Teniendo la posibilidad de contar con diversos usos de una API, es interesante analizar la posibilidad de analizar patrones de uso de la misma y poder reflejar esta información en la abstracción. Esto siginificaría dar más valor a la EPA como documentación y a la vez permitiría discriminar flujos sospechosos por su baja frecuencia de uso (potencialmente no deseado).

Por último, sería muy interesante extender la detección de transiciones espurias a generación de casos de test de la API. Poder confirmar la factibilidad de una transición implica contar con al menos una traza que pruebe su uso. De ser posible obtener esta información se podrían generar test suites que utilicen la EPA como criterio de cubrimiento.

Capítulo 8

Bibliografía

[1] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 98–109, 2005.

[2] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 1–3, 2002.

[3] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pages 364–387, 2005.

[4] E. M. Clarke, R. P. Kurshan, and H. Veith. The localization reduction and counterexample-guided abstraction refinement. In *Time for Verification, Essays in Memory of Amir Pnueli*, pages 61–71, 2010.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977.

[6] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 85–96, 2010.

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

[8] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Enabledness-based program abstractions for behaviour validation. *ACM Transactions on Software Engineering and Methodology*, 22(3):art. 25, feb 2013.

[9] G. de Caso, V. A. Braberman, D. Garbervetsky, and S. Uchitel. Automated abstractions for contract validation. *IEEE Trans. Software Eng.*, 38(1):141–162, 2012.

[10] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 59–69, 2001.

[11] R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, pages 465–490, 2004.

[12] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 2103–2110, 2010.

[13] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 51–60, 2008.

[14] D. Giannakopoulou and C. S. Pasareanu. Interface generation and compositional verification in javapathfinder. In *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 94–108, 2009.

[15] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pages 72–83, 1997.

[16] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA*, pages 112–122, 2002.

[17] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 31–40, 2005.

[18] A. Lal and S. Qadeer. Reachability modulo theories. In *Reachability Problems - 7th International Workshop, RP 2013, Uppsala, Sweden, September 24-26, 2013 Proceedings*, pages 23–44, 2013.

[19] D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 264–274, 1992.

[20] L. L. Liu, B. Meyer, and B. Schoeller. Using contracts and boolean queries to improve the quality of automatic test generation. In *Tests and Proofs, First International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers*, pages 114–130, 2007.

[21] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 501–510, 2008.

[22] B. Meyer. *Object-Oriented Software Construction, 1st edition.* Prentice-Hall, 1988.

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

[24] T. E. Uribe. Abstraction-based deductive-algorithmic verification of reactive systems. 1999.

[25] E. Zoppi, V. A. Braberman, G. de Caso, D. Garbervetsky, and S. Uchitel. Contractor.net: inferring typestate properties to enrich code contracts. In *Proceedings of the 1st Workshop on Developing Tools as Plug-ins, TOPI 2011, Waikiki, Honolulu, HI, USA, May 28, 2011*, pages 44–47, 2011.