DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Enriqueciendo Code Contracts con Typestates

Tesis de Licenciatura en
Ciencias de la Computación

Diciembre de 2012

Alumno

Edgardo Julio Zoppi
ezoppi@dc.uba.ar
LU 55/05

Directores

Dr. Diego Garbervetsky
Lic. Guido de Caso

Resumen

Por lo general un componente de software tiene requerimientos que no son para nada triviales cuando se trata del orden en que sus métodos o procedimientos pueden ser ejecutados. Esta situación es común, por ejemplo, en el caso de APIs que implementan protocolos. Este trabajo trata de ayudar en el problema de validar si una implementación particular cumple con el comportamiento esperado en los casos en que su descripción es informal, parcial o directamente inexistente. Con esto en mente, presentamos CONTRACTOR.NET, una herramienta que permite construir especificaciones por contratos con información de *typestate* que puede ser utilizada para verificar código cliente. CONTRACTOR.NET usa y extiende CODE CONTRACTS para proveer especificaciones por contratos más robustas. El proceso se realiza en dos etapas. Primero, el código fuente de una clase es analizado de forma estática para extraer un modelo abstracto finito de comportamiento (en la forma de *typestate*) adecuado para su validación y refinamiento por parte del programador. La segunda etapa consiste en aumentar la especificación por contratos original de la clase en cuestión con la información de *typestate* inferida en la etapa anterior, con el objetivo de proveer más datos para la verificación del código cliente. Los *typestates* inferidos cumplen con la propiedad *enabledness preserving*, un nivel de abstracción que 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.

Agradecimientos

Agradezco principalmente a mi familia, por apoyarme y acompañarme en todo momento a lo largo de mi carrera.

A mis directores Diego Garbervetsky y Guido de Caso, por darme la oportunidad de hacer mi tesis con ellos y ayudarme siempre que lo necesité. Sobre todo quiero resaltar la paciencia y dedicación que tuvieron conmigo hasta el último día.

A los jurados Hernán Melgratti y Mariano Moscato, por tomarse el trabajo de leer y corregir esta tesis rápidamente.

A mis amigos Javier y Bruno, por haber sido mis compañeros de trabajos prácticos en casi todas las materias de la carrera, facilitando enormemente que haya podido llegar a finalizarla con éxito.

Por último, a mis amigos y compañeros de la facu Chapa, Facu, Fer, Giga, Lean, Leo R., Leo S., Luisito, Marta, Martín, Mati, Maxi, Nati, Nelson, Pablito, Palla, Serch, Tara, Vivi y Z, con los que cursé, estudié y me divertí mucho durante todos estos años.

Índice general

Capítulo 1

Introducción

El diseño por contratos [Mey88] 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, extendiendo la definición estándar de los tipos abstractos de datos con precondiciones, postcondiciones e invariantes. Code Contracts [FBL10] es un proyecto de Microsoft Research que acerca las ventajas de la programación de diseño por contratos a todos los lenguajes basados en .NET, posibilitando el uso de contratos sin la necesidad de utilizar un compilador específico. Además estos contratos sirven como documentación adicional y pueden ser usados para mejorar la calidad del software por medio de chequeos en tiempo de ejecución y verificación estática de los mismos en código cliente.

Sin embargo, el proyecto Code Contracts parece estar más enfocado actualmente en contratos que especifican precondiciones sobre los parámetros de los métodos en lugar de contratos que se refieran al estado del objeto. Es decir, son pocas las clases que cuentan con especificaciones que describen el estado en el cual una instancia debe encontrarse después de la ejecución de uno de sus métodos. Este hecho no es sorprendente debido a que no es fácil escribir contratos que se refieran a estructuras de datos compartidas y modificadas por varios métodos. Aún en los casos donde pudieron ser escritos, su validación es compleja debido a que los contratos se combinan de formas inesperadas cuando los métodos son invocados en secuencia, provocando documentación incorrecta y peor aún, comportamientos no deseados, que eventualmente desembocan en problemas a la hora de tratar de asegurar la calidad del código cliente.

Las especificaciones de *typestates* [DF04] 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. 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* [AČMN05] si no existe una secuencia de invocaciones que viole los invariantes internos de la librería y es *permisivo* [HJM05] si contiene a todas y cada una de dichas secuencias.

También pueden ser usados para verificar el cumplimiento de los contratos por parte del código cliente, es decir, obligar a los clientes a que siempre realicen secuencias de invocaciones válidas a los métodos de una API. Dicho cumplimiento es típicamente asegurado usando técnicas de verificación de tipos (como por ejemplo el verificador de protocolos Fugue [DF04]) o por medio de la codificación de los *typestates* como máquinas de estado a ser verificadas con la ayuda de *model checkers* tales como Slam

SDV [BR02] o Blast [BHJM07].

Otro enfoque para garantizar la correcta utilización de la librería es el *typestate monitoring*. Esta técnica propone verificar dinámicamente el código cliente en tiempo de ejecución, lanzando excepciones cuando se viole el *typestate*.

Existen también enfoques híbridos como el adoptado por Clara [Bod09] que realizan tanto verificación estática como dinámica, complementándose.

Sin embargo, el uso de especificaciones de *typestates* aplicadas a ciertas áreas particulares como es la verificación de clientes que utilizan APIs de bajo nivel no está muy difundido y por este motivo creemos que este tipo de especificación puede ayudar a la documentación del comportamiento de una clase y a la verificación de código cliente en un contexto más grande. Consideramos que la construcción automática de una abstracción como ésta puede ser de utilidad para validar la implementación de una API contra requerimientos pobremente documentados o el modelo mental del programador. Con esto en mente, proponemos entonces enriquecer Code Contracts con los beneficios antes mencionados que los *typestates* proporcionan.

Nos enfocaremos en generar abstracciones de tipo *enabledness preserving* [dCBGU09], que consisten en modelos finitos de comportamiento que agrupan instancias de una clase según los métodos que se encuentran habilitados en ellas. Como ventaja permiten trazar formalmente su relación con el código fuente. Este nivel de abstracción mostró resultados positivos al momento de validar modelos, ayudando a la detección de errores tanto en los contratos como en el código fuente, en los requerimientos y en el entendimiento de los mismos por parte del diseñador de la API. Permite también la generación de documentación más completa y detallada.

1.1. Objetivo

El objetivo de este trabajo es proveer técnicas automatizadas para dar soporte a la construcción de abstracciones de tipo *enabledness preserving* a partir de código fuente anotado con precondiciones para posibilitar su validación; y posteriormente enriquecer la especificación por contratos original con la información del *typestate* generado, para que pueda ser utilizada para la verificación automática de código cliente.

Concretamente el objetivo es implementar una herramienta en .NET utilizando Code Contracts, basada en la ya existente Contractor [dCBGU11].

Dicha herramienta debe ser capaz de analizar el código fuente de una clase anotada con precondiciones y generar el *typestate* correspondiente. Presentarlo de forma sencilla permitiéndole al usuario validar la abstracción generada contra su modelo mental para poder detectar inconsistencias de forma fácil y rápida en una primer etapa. Por último, reforzar la especificación de la clase utilizando la información obtenida de la abstracción con el objetivo de poder verificar (dinámica o estáticamente) de forma automática el correcto uso por parte de algún cliente externo en una etapa posterior.

Principalmente proveer al desarrollador de una herramienta que le ayude a detectar (de forma previa a la ejecución) errores relacionados con el comportamiento de un objeto a partir de los distintos estados por los que va transitando al ejecutar cada uno de sus métodos. En esta primer etapa los errores que puedan ser descubiertos surgen como resultado de la validación contra el modelo mental del desarrollador.

Asimismo la abstracción generada sirve como documentación para los usuarios de la API y permite verificar estática y dinámicamente su correcto uso. En este sentido la herramienta también ayuda a detectar errores en la utilización de la librería. En esta segunda etapa los errores surgen como resultado de la verificación de la especificación enriquecida de la API.

A diferencia del trabajo realizado previamente por [dCBGU11], donde las abstracciones son generadas a partir de código C utilizando *reachability queries* para ser evaluadas por algún *model checker* como BLAST, nuestra implementación utiliza en cambio, el verificador estático de CODE CONTRACTS para la plataforma .NET. Además, extendemos el alcance del trabajo antes mencionado incorporando la instrumentación opcional de la clase con el objetivo de enriquecer la especificación original de la misma.

1.2. Estructura de la tesis

En el capítulo 2 presentamos un ejemplo ilustrativo pero completo sobre el modo de uso de la herramienta CONTRACTOR.NET así como también los beneficios obtenidos.

En el capítulo 3 presentamos dos versiones del algoritmo de construcción de EPAs: el primero es la versión ingenua, fácil de entender pero muy poco escalable en la práctica; y el segundo basado en exploración mucho más eficiente. En éste capítulo nos enfocaremos en los algoritmos en sí mismos, abstrayendo la implementación particular desarrollada así como también las tecnologías utilizadas.

En el capítulo 4 presentamos y discutimos diferentes alternativas acerca de cómo utilizar la información obtenida mediante la generación de la abstracción en el capítulo anterior con el objetivo de enriquecer la especificación por contratos original. Nuevamente nos abstraeremos de los detalles específicos de la implementación realizada.

En el capítulo 5 presentamos la herramienta CONTRACTOR.NET que implementa lo explicado en los capítulos anteriores. Aquí nos centraremos en los detalles de implementación particulares, las librerías utilizadas y el modo de uso.

En el capítulo 6 discutimos algunos ejemplos representativos y analizamos los resultados obtenidos en ambas etapas del proceso (validación y verificación).

En el capítulo 7 mencionamos brevemente otros trabajos relacionados al nuestro.

Finalmente, en el capítulo 8 presentamos las conclusiones de esta tesis y discutimos algunas alternativas de trabajo a futuro.

Capítulo 2

Motivación

A continuación presentamos un ejemplo muy básico y sencillo para mostrar los beneficios que CONTRACTOR.NET puede proporcionarle a los programadores en su tarea.

Consideremos una hipotética (y muy simplificada) implementación de un controlador para la puerta de un vagón de tren para pasajeros. Cabe mencionar que debido a que nuestra intención es enfocarnos en el tema de estudio de este trabajo, nos abstraeremos de todos los detalles específicos y complejos que conlleva dicha implementación en la realidad.

Uno de los requerimientos de seguridad del controlador estipula que la puerta debe permanecer cerrada mientras el tren se encuentre en marcha para evitar accidentes. Sin embargo, bajo ciertas circunstancias, como es el caso de una emergencia, la puerta debe abrirse de todas formas aún en movimiento[1] para posibilitar la evacuación de los pasajeros del tren.

El programa 2.1 muestra parte de la implementación del controlador. El mismo se encuentra encapsulado en la clase `Door`, la cual expone los siguientes métodos anotados con precondiciones:

- `Open`, que se encarga de abrir la puerta del vagón. Requiere que la puerta esté cerrada y el tren detenido.
- `Close`, por su parte, se encarga de cerrar la puerta. Pide como precondición que esté abierta y que el tren no se encuentre en estado de alarma.
- `Start`, que establece que el tren se puso en marcha y pasa a estar en movimiento. Por lo tanto es necesario que se encuentre detenido. Además, se asegura de que la puerta esté cerrada en caso de que no haya una emergencia, cerrándola de ser necesario.
- `Stop`, que por el contrario, indica que el tren dejó de moverse y pasa a estar detenido. Para esto es necesario que el tren esté en marcha.
- `Alarm`, que establece que el tren se encuentra en estado de emergencia debido a que un pasajero activó la alarma. Tiene como precondición que la alarma no esté activada previamente. Además, se asegura de que la puerta esté abierta, abriéndola de ser necesario.

[1] En una implementación real la puerta debería permanecer cerrada a pesar de encontrarse el tren en una situación de emergencia siempre que la velocidad sea mayor a un determinado límite considerado seguro.

```
1  public class Door {
2      public bool danger, closed, moving;
3
4      private void Invariant() {
5          Contract.Invariant(!danger || !closed);
6      }
7      public Door() {
8          closed = true; moving = false; danger = false;
9      }
10     // Controlled operations
11     public void Open() {
12         Contract.Requires(closed && !moving);
13         closed = false;
14     }
15     public void Close() {
16         Contract.Requires(!closed && !danger);
17         closed = true;
18     }
19     // Monitored events
20     public void Start() {
21         Contract.Requires(!moving);
22         moving = true;
23         if (!danger) closed = true;
24     }
25     public void Stop() {
26         Contract.Requires(moving);
27         moving = false;
28     }
29     public void Alarm() {
30         Contract.Requires(!danger);
31         danger = true; closed = false;
32     }
33     public void Safe() {
34         Contract.Requires(danger);
35         danger = false;
36     }
37 }
```

Programa 2.1: Controlador de la puerta de un vagón de tren.

- `Safe`, por último, indica que el estado de emergencia terminó y la alarma fue desactivada. Para esto es necesario que el tren se haya encontrado en esa situación previamente.

Es importante mencionar que tanto el método `Open` como el `Close` son operaciones propias del controlador de la puerta mientras que el resto de los métodos son eventos monitoreados por este componente, sobre los cuales no tiene ningún control.

El requerimiento de seguridad comentado anteriormente se ve reflejado en el invariante de clase que asegura que la puerta permanezca abierta siempre que el tren se encuentre en estado de emergencia.

Como se puede ver, cada uno de los métodos está anotado con precondiciones en la forma usual que Code Contracts provee para hacerlo. De ésta manera se previene su ejecución en contextos incorrectos, que de no ser así, podrían causar resultados inesperados. Sin embargo, ninguno de ellos cuenta con postcondiciones asociadas. Esto puede parecer extraño en una primera impresión, pero la experiencia indica que

típicamente las anotaciones que se escriben y suelen encontrarse en la práctica son únicamente precondiciones que se refieren a validaciones de los parámetros de los métodos, como es el caso de las clases anotadas del *framework* .NET. Más aún, existe una construcción especial de Code Contracts para facilitar la generación de este tipo de precondiciones[2].

Por otro lado, entender si un método dado en particular provee la funcionalidad esperada es relativamente fácil si lo comparamos con comprender su interacción con el resto de los métodos de la clase. La complejidad radica en que para lograrlo es necesario analizar cada posible secuencia de invocaciones a dichos métodos. Por lo tanto, teniendo como objetivo principal asistir al programador en este desafío, proponemos como primer paso de nuestro enfoque la construcción automática de una abstracción de estados finita con información de *typestates*. La figura 2.1 muestra dicha abstracción para el controlador antes presentado.

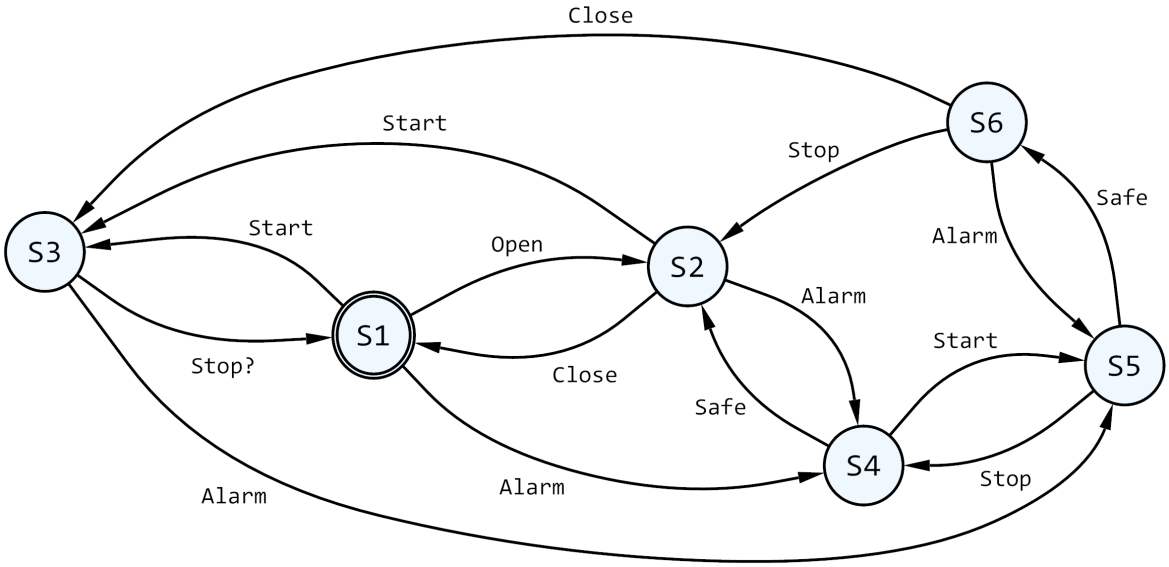

Figura 2.1: Abstracción correspondiente al controlador de la puerta de un vagón de tren. El estado inicial está marcado con un doble círculo exterior.

Cada uno de los estados abstractos agrupa todas las instancias de la clase que habilitan el mismo conjunto de métodos, destacando el estado inicial con un doble círculo. Por ejemplo, `S2` es el único estado de la abstracción que habilita los métodos `Close`, `Start` y `Alarm` a la vez que deshabilita `Open`, `Stop` y `Safe`. Esta estrategia de agrupamiento de las instancias de la clase, denominada *enabledness preserving*, provee un buen compromiso entre el tamaño de la abstracción y el nivel de detalle requerido, siendo de utilidad para descubrir inconsistencias entre la implementación particular de la clase y su funcionalidad esperada [dCBGU11]. Este tipo de análisis entra en el marco de la validación del componente contra los requerimientos y es el principal resultado de la primer etapa de nuestro enfoque.

Asimismo, con una inspección rápida de la abstracción podemos deducir que el tren inicialmente se encuentra detenido, con las puertas cerradas y en un contexto donde no hay una situación de emergencia (observar el estado inicial `S1`). De ésta forma podemos considerarla además un complemento a la documentación propia de la clase.

[2]Para más información al respecto leer sobre `Contract.EndContractBlock`.

Posteriormente, una vez finalizada la implementación del controlador y distribuida, otros componentes de software pueden empezar a utilizarla. En el programa 2.2 se muestra uno de éstos posibles clientes. En este hipotético escenario la idea es modelar cuando el tren sufre una falsa alarma de emergencia mientras se encuentra viajando de una estación a otra.

```
1  public void AlarmScenario() {
2      Door door = new Door();
3
4      door.Start();    // Departing station
5      door.Alarm();    // Emergency button pressed...
6      door.Safe();     // ...but it was a false alarm
7      door.Stop();     // Arriving at next station
8      door.Open();     // Opening doors
9      door.Close();    // Getting ready to depart again
10 }
```

Programa 2.2: Posible cliente del controlador de la puerta de un vagón de tren.

Es probable que el programador de dicho cliente quiera verificar de forma estática su código utilizando Code Contracts como haría habitualmente con cualquier otro componente de software. Desafortunadamente, el controlador de la puerta no incluye ninguna postcondición imposibilitando a Code Contracts su correcta verificación. Cabe mencionar que el análisis efectuado por esta herramienta es modular, inspeccionando cada método de forma aislada, para luego propagar sus postcondiciones al código cliente. Por este motivo la ausencia de contratos hace imposible que el verificador estático pueda conocer lo que hacen los métodos del controlador cuando son invocados por terceros.

La figura 2.2 muestra los mensajes de advertencia generados por Code Contracts para cada una de las invocaciones a los métodos públicos del controlador efectuadas por el cliente. Como se puede ver, el verificador estático no puede probar las precondiciones de dichos métodos debido a la falta de postcondiciones en los mismos (incluyendo el constructor de la clase `Door`).

Para facilitar esta tarea, presentamos como segundo y último paso de nuestra propuesta, un mecanismo para reforzar la especificación original de la clase `Door` enriqueciéndola con postcondiciones y otras restricciones inferidas y obtenidas de la abstracción generada en el paso anterior. Como resultado intentamos facilitarle el trabajo al verificador estático de Code Contracts proveyéndole mayor información para descubrir errores en el código cliente. Este tipo de análisis entra en el marco de la verificación automática del componente y conforma el principal objetivo de la segunda etapa de nuestro enfoque.

En particular, si observamos la línea 8 en el programa 2.2 podemos ver que el cliente está intentando abrir la puerta cuando en realidad ésta ya ha sido abierta al momento de producirse la emergencia, provocando una violación de la precondición del método `Open`. Éste es un ejemplo de cómo puede resultar la interacción de los distintos métodos de una clase, que por lo general, termina desembocando en errores inesperados y difíciles de encontrar.

El problema se origina por la incorrecta suposición de que la puerta se cierra automáticamente una vez que la situación de emergencia llega a su fin (evento capturado por

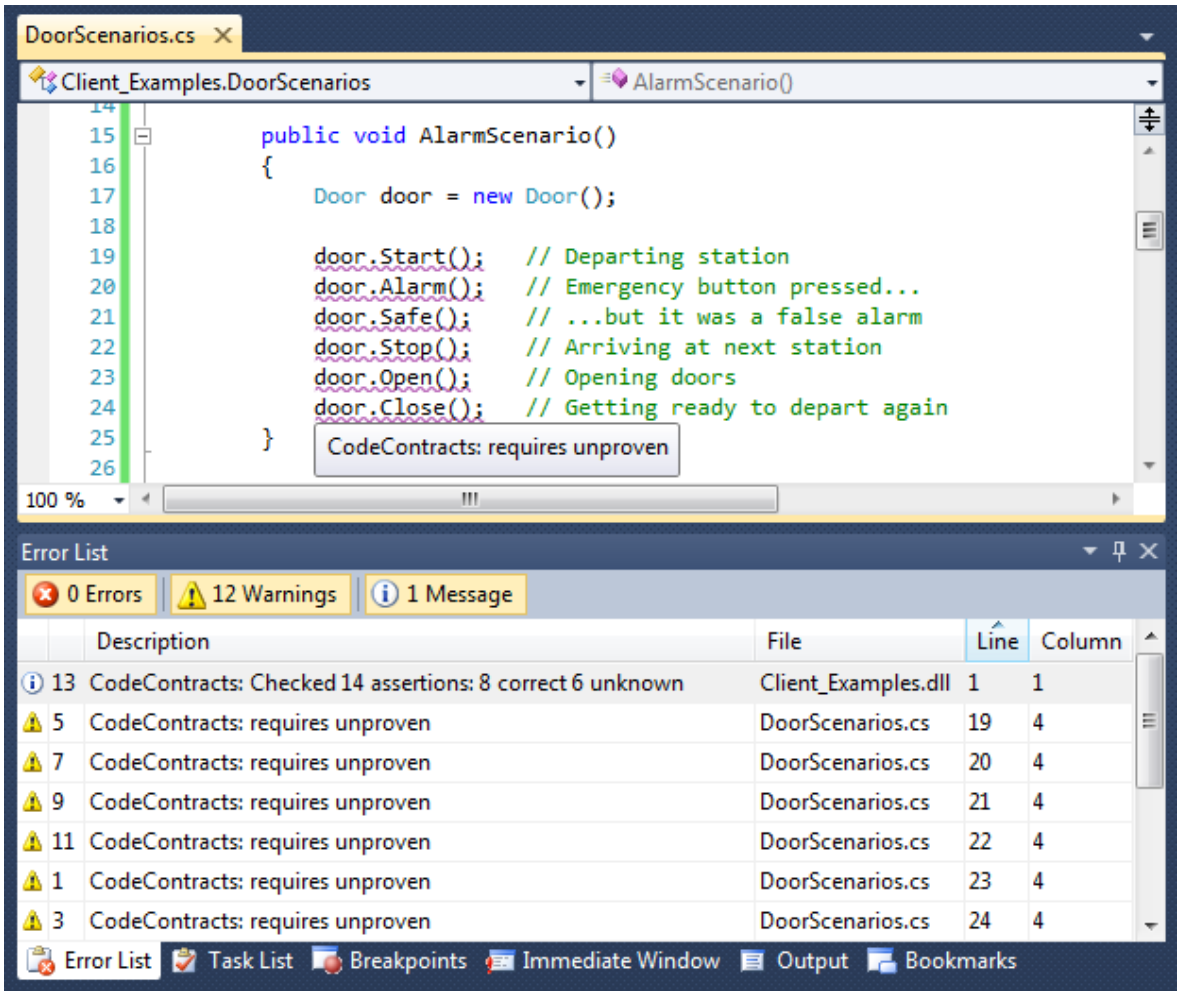

Figura 2.2: Mensajes de advertencia generados por CODE CONTRACTS.

el método `Safe`). Este malentendido de la funcionalidad de la clase `Door` podría haberse evitado inspeccionando cuidadosamente la abstracción de la figura 2.1. Al realizar un seguimiento paso a paso de la secuencia de llamadas a los métodos del controlador que realiza el código cliente presentado en el programa 2.2 obtenemos la siguiente traza:

$$\texttt{S1} \xrightarrow{\texttt{Start}} \texttt{S3} \xrightarrow{\texttt{Alarm}} \texttt{S5} \xrightarrow{\texttt{Safe}} \texttt{S6} \xrightarrow{\texttt{Stop}} \texttt{S2}$$

En este último estado abstracto `S2`, el método `Open` no se encuentra habilitado, por lo que la versión enriquecida de la especificación, que cuenta con información proveniente de los *typestates*, hace posible que CODE CONTRACTS detecte el error y se lo notifique al programador en tiempo de compilación.

La figura 2.3 muestra ahora un único mensaje de advertencia generado por CODE CONTRACTS, correspondiente a la invocación del método `Open` efectuada por el cliente. Como se puede ver, el analizador estático cuenta ahora con información suficiente para refutar la precondición de dicho método (y verificar las precondiciones del resto de los métodos). Esto se debe a la utilización de la versión enriquecida del controlador, la cual cuenta con postcondiciones inferidas al generar la abstracción.

A modo de resumen, mediante este ejemplo sencillo mostramos cómo el enfoque adoptado puede asistir a los programadores en el reconocimiento de problemas al mo-

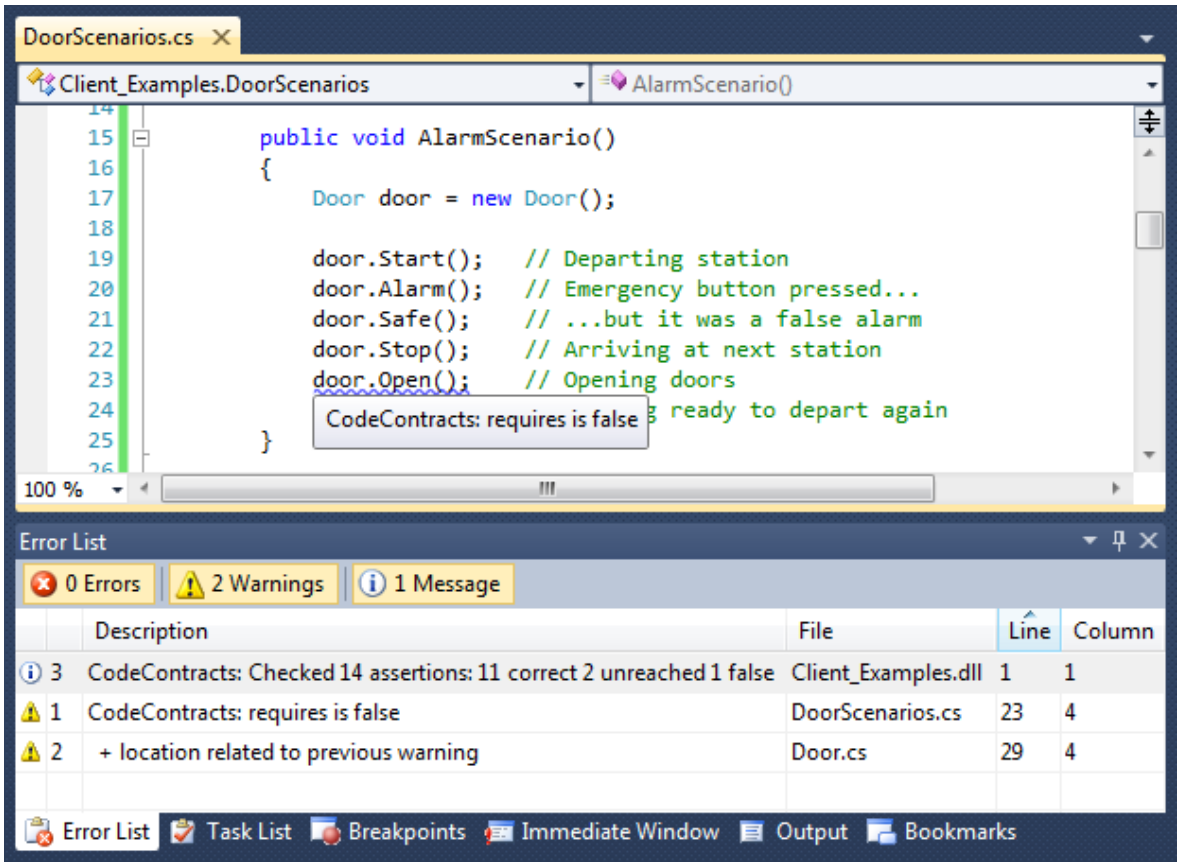

Figura 2.3: Único mensaje de advertencia generado por Code Contracts cuando el cliente utiliza la versión enriquecida de la clase `Door`.

mento de utilizar una API en su código cliente. También da una primera impresión del potencial que brindan éste tipo de abstracciones, especialmente las basadas en modelos que cumplen con la propiedad denominada *enabledness preserving*, para validar especificaciones por contratos (ya sean explícitos mediante anotaciones y/o implícitos inferidos de la implementación de cada uno de los métodos).

En el próximo capítulo nos enfocamos en los detalles teóricos que hacen posible la construcción automática de este tipo de *typestates*.

Capítulo 3

Construcción

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 [dCBGU11], trabajo en el cual nos basamos.

3.1. Modelo formal

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

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

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

Veamos un ejemplo concreto. Consideremos la implementación de una pila genérica de capacidad limitada como la que muestra el programa 3.1. Consta de los métodos `Push` para apilar un nuevo elemento y `Pop` para desapilar el elemento que se encuentra en el tope de la pila. Es fácil ver que sólo puede almacenar cinco elementos. Su correspondiente interpretación semántica es $Stack = \langle M, F, R, inv, init \rangle$, donde:

- $M = \{push, pop\}$. Este conjunto de métodos constituye la interfaz pública de la implementación de la pila.
- $F = \{f_{push}, f_{pop}\}$. Donde estas funciones corresponden a la interpretación semántica del código de los métodos `Push` (líneas 16-19) y `Pop` (líneas 20-23) respectivamente.
- R_{push} es el predicado que es verdadero sólo para las instancias de la pila que no estén llenas.

- R_{pop} es el predicado que es verdadero sólo para las instancias de la pila que no estén vacías.
- *inv* es el predicado que es verdadero sólo para las instancias de la pila que cumplan $0 \leq count \leq capacity$.
- *init* es el predicado que es verdadero sólo para las instancias de la pila que estén vacías, condición que se cumple en las nuevas instancias.

```
1  public class Stack<T> {
2      public const int capacity = 5;
3      public int count;
4      private T[] data;
5
6      [ContractInvariantMethod]
7      private void Invariant() {
8          Contract.Invariant(count >= 0);
9          Contract.Invariant(count <= capacity);
10     }
11
12     public Stack() {
13         count = 0;
14         data = new T[capacity];
15     }
16     public void Push(T item) {
17         Contract.Requires(count < capacity);
18         data[count++] = item;
19     }
20     public T Pop() {
21         Contract.Requires(count > 0);
22         return data[--count];
23     }
24 }
```

Programa 3.1: Pila de capacidad limitada.

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

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.

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 $c_1 \equiv_e c_2$), sii para cada método $m \in M$ vale $R_m(c_1) \Leftrightarrow R_m(c_2)$.

Dado el LTS del espacio de estados posibles correspondiente a la interpretación semántica de una clase 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 [dCBGU11] 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 todas las instancias que comparten el mismo conjunto de métodos habilitados y pueden ser caracterizados con un *invariante de estado*.

Dicho invariante de estado hace posible la construcción de este tipo de abstracciones, debido a que si bien desde un punto de vista teórico podemos obtener el EPA a partir del LTS utilizando el concepto de equivalencia de instancias descripto en el párrafo anterior, en la práctica esto no es posible dado que el LTS es 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 $inv_{ms} : C \to \{true, false\}$,

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

De esta definición se desprende que un estado abstracto ms es válido sii $\exists c \in C.\ inv_{ms}(c)$. Por lo tanto, un estado abstracto es simplemente un conjunto de métodos; que en caso de ser válido, existe una instancia $c \in C$ en la que se encuentran habilitados (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).

Volviendo al ejemplo de la pila, podemos calcular los invariantes de sus estados abstractos de la siguiente forma:

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

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

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

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

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

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

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

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

Notar que el punto 2 define a S como el conjunto de partes del conjunto finito M, con lo cual el EPA caracterizado resulta ser un LTS finito. Además, la función de transición se define como $\delta : S \times \Sigma \to 2^S$, cuyo codominio es el conjunto de partes de S, por lo tanto, el LTS resultante es no determinístico.

3.2. Algoritmo

De la caracterización anterior de un EPA se puede deducir fácilmente una primer versión del algoritmo de construcción. Se basa en el hecho de que una clase con k métodos públicos puede tener potencialmente como máximo 2^k estados abstractos alcanzables (ya que un método puede pertenecer o no a un determinado estado). Teniendo en cuenta esto, una implementación ingenua generaría exhaustivamente todos los estados y sus transiciones sólo para luego restringir el resultado al fragmento realmente alcanzable desde los estados iniciales. Es inmediato notar que dicho algoritmo es exponencial en la cantidad de métodos, por lo tanto no escala, resultando inadecuado para implementar en la práctica.

Una opción mucho más eficiente es la presentada en [dCBGU11], cuyo pseudo código exhibimos en el algoritmo 3.1. En este caso se realiza una exploración de búsqueda a lo ancho (*breadth first search* o simplemente BFS) del espacio de estados abstracto a partir de los estados iniciales, reduciendo drásticamente el tiempo de ejecución.

La función de transición se inicializa vacía. El conjunto M_0^- contiene todos los métodos que nunca están habilitados en ningún estado inicial (porque sus precondiciones nunca se cumplen luego de crear una instancia de la clase). De forma análoga, el conjunto M_0^+ contiene todos los métodos que siempre están habilitados en todos los estados iniciales (porque sus precondiciones siempre se cumplen luego de crear una instancia de la clase).

Por su parte, S_0^C es el conjunto de estados candidatos para ser iniciales, los cuales se caracterizan por contener todos los métodos de M_0^+ y ninguno de M_0^-. Siguiendo con el ejemplo del programa 3.1 de la sección anterior, tenemos que $M_0^+ = \{push\}$, $M_0^- = \{pop\}$ y por lo tanto $S_0^C = \{\{push\}\}$.

1: **procedure** `ConstructEPA`(C: $\langle M, F, R, inv, init\rangle$) : $\langle \Sigma, S, S_0, \delta\rangle$
2: $\quad \Sigma \leftarrow M$
3: $\quad S \leftarrow \varnothing$
4: $\quad \delta(ms, m) \leftarrow \varnothing \qquad \forall ms, m$
5: $\quad M_0^- \leftarrow \{m \in M \mid \forall c.\ init(c) \Rightarrow \neg R_m(c)\}$
6: $\quad M_0^+ \leftarrow \{m \in M \mid \forall c.\ init(c) \Rightarrow R_m(c)\}$
7: $\quad S_0^C \leftarrow \{ms \subseteq M \mid M_0^+ \subseteq ms, M_0^- \cap ms = \varnothing\}$
8: $\quad S_0 \leftarrow \{ms \in S_0^C \mid \exists c.\ inv_{ms}(c) \wedge init(c)\}$
9: $\quad W \leftarrow$ queue initialized with elements in S_0
10: $\quad$ **while** there is a state ms at the top of W **do**
11: $\quad\quad W \leftarrow W - [ms]$
12: $\quad\quad S \leftarrow S \cup \{ms\}$
13: $\quad\quad$ **for each** method $m \in ms$ **do**
14: $\quad\quad\quad M^- \leftarrow \{n \in M \mid \forall c.\ inv_{ms}(c) \Rightarrow \neg R_n(f_m(c))\}$
15: $\quad\quad\quad M^+ \leftarrow \{n \in M \mid \forall c.\ inv_{ms}(c) \Rightarrow R_n(f_m(c))\}$
16: $\quad\quad\quad S^C \leftarrow \{ns \subseteq M \mid M^+ \subseteq ns, M^- \cap ns = \varnothing\}$
17: $\quad\quad\quad$ **for each** state $ns \in S^C$ **do**
18: $\quad\quad\quad\quad$ **if** $\exists c.\ inv_{ms}(c) \wedge inv_{ns}(f_m(c))$ **then**
19: $\quad\quad\quad\quad\quad \delta(ms, m) \leftarrow \delta(ms, m) \cup \{ns\}$
20: $\quad\quad\quad\quad\quad$ **if** $ns \notin S \wedge ns \notin W$ **then**
21: $\quad\quad\quad\quad\quad\quad W \leftarrow W \cup [ns]$

Algoritmo 3.1: Construcción de un EPA

Luego, los estados iniciales serán aquellos candidatos que verifiquen la condición inicial así como también su invariante de estado (exactamente lo que estipula el punto 3 de la caracterización). Es importante notar que mientras más métodos sean clasificados como necesariamente habilitados o deshabilitados, obtenemos menos estados iniciales candidatos que deban ser verificados posteriormente.

Una vez determinado S_0, se inicializa una cola W con los estados pendientes de ser analizados. Cada vez que un estado ms es analizado se consideran todos sus métodos habilitados $m \in ms$.

De forma similar a la etapa de inicialización del algoritmo, el conjunto M^- contiene todos los métodos que nunca están habilitados luego de ejecutar m a partir del estado ms (porque sus precondiciones nunca se cumplen en este contexto). De forma análoga, el conjunto M^+ contiene todos los métodos que siempre están habilitados luego de ejecutar m a partir del estado ms (porque sus precondiciones siempre se cumplen en este contexto).

Por su parte, S^C es el conjunto de estados candidatos destino, los cuales se caracterizan por contener todos los métodos de M^+ y ninguno de M^-, similar al conjunto S_0^C. Por ejemplo, si $M^+ = \{pop\}$ y $M^- = \varnothing$ entonces $S^C = \{\{pop\}, \{push, pop\}\}$. Notar que si ambos conjuntos M^+ y M^- están vacíos, tenemos que $S^C = 2^M$, es decir, el

conjunto de estados candidatos destino es el conjunto de partes de M.

Luego, cada uno de éstos estados candidato es considerado para verificar si puede ser alcanzado realmente al evolucionar ms tras la ejecución del método m. Por último, cada vez que un nuevo estado abstracto ns es alcanzado, se lo encola en W para ser analizado posteriormente.

La demostración de correctitud de éste algoritmo fue realizada en el trabajo previo antes citado de [dCBGU11]. Cabe mencionar que su complejidad temporal y espacial en el peor caso es exponencial respecto de la cantidad de métodos, al igual que la versión ingenua discutida anteriormente. Sin embargo, la diferencia radica en que mientras podamos clasificar más métodos como necesariamente habilitados o deshabilitados en un estado particular, menos estados candidatos deben ser considerados por el algoritmo. En la práctica, esta optimización reduce de forma significativa los tiempos de ejecución. Para más información al respecto ver [dCBGU10].

Por último, éste algoritmo es un *template* que estipula *cuáles* son las validaciones que se deben realizar, pero no indica *cómo* resolverlas. Éste es por lo general un problema indecidible, lo que nos obliga a analizar el impacto que pueden tener respuestas poco precisas e inciertas en la abstracción resultante y qué cambios deberán realizarse en el algoritmo para contemplarlas. Dicho análisis concluye que la incertidumbre en las respuestas sólo afecta en las validaciones de las líneas 8 y 18 del algoritmo 3.1, mientras que las validaciones de las líneas 5, 6, 14 y 15 no tienen consecuencias en el resultado.

Analicemos primero éstas últimas. Como se puede ver, dichas validaciones son muy similares. En caso de obtener una respuesta incierta al momento de validar la implicación, lo más seguro es excluir el método m del conjunto correspondiente ya que no hay ninguna garantía que indique que siempre (o nunca, según el caso) va a estar habilitado (o deshabilitado). De hecho, se trata sólo de una optimización del algoritmo, bien podrían éstos conjuntos M_0^-, M_0^+, M^- y M^+ establecerse como $\varnothing$ sin alterar el resultado. La única consecuencia sería hacer más lenta su ejecución ya que éstos conjuntos se utilizan para reducir los posibles estados abstractos que posteriormente serán analizados.

Distinto es el caso de las otras validaciones (líneas 8 y 18), que son cruciales. La primera afecta el conjunto de estados iniciales, mientras que la segunda afecta las transiciones, y por lo tanto se ven también afectados los estados alcanzables que deben ser analizados. Para solucionar dichos inconvenientes, decidimos que lo mejor en estos casos ante la eventual incertidumbre es no restringir el resultado, agregando los estados y transiciones de todas formas (adornadas con el signo ? para distinguirlas). Lo que se obtiene entonces en el peor caso es un superconjunto 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 ésta decisión se deduce que la abstracción resultante del algoritmo, en un contexto de incertidumbre, es una simulación del EPA. 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 su contrapartida exacta. A su vez, tiene la desventaja de aceptar trazas que puedan llegar a ser inválidas. 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 vá-

lidas posiblemente no deseadas, pero que pasarían desapercibidas justamente por no mostrarle todo el comportamiento.

Para terminar, recordamos que la abstracción generada tiene múltiples usos. Principalmente se la puede utilizar para la validación de clases, que constituye la primer fase de nuestro enfoque. Es decir, comparar el *typestate* con el entendimiento que el programador tiene acerca de la funcionalidad esperada, para encontrar discordancias y descubrir potenciales errores en la implementación. O bien, para contrastarla contra los requerimientos con el objetivo de hallar incongruencias en los mismos.

Por otro lado, también se la puede utilizar para asegurar el correcto uso de la librería por parte del código cliente. Es decir, verificar que las secuencias de llamadas a los métodos sean válidas, o lo que es lo mismo, estén incluidas en las trazas del *typestate*. Dicha verificación constituye la segunda fase de nuestro enfoque, la cual consta de utilizar la información descubierta al generar la abstracción para reforzar los contratos de la clase, como se verá en el próximo capítulo.

Por último, la abstracción en sí misma puede ser utilizada como complemento de la documentación, para ayudar a los programadores de código cliente a entender la especificación de la librería y el comportamiento de la misma.

Capítulo 4

Instrumentación

La segunda etapa de nuestra propuesta consiste en utilizar la información de *typestate* inferida al momento de generar la abstracción, con el objetivo de aportar una mayor cantidad de datos que permitan garantizar la correcta utilización de la librería por parte del código cliente. Para lograr dicho objetivo, proponemos extender la especificación por contratos original, agregando restricciones que obliguen a la implementación a respetar la abstracción subyacente. El enfoque que adoptamos entra en el contexto de lo que se denomina monitoreo de *typestates* (*typestate monitoring* [BLH08]), en el cual se instrumenta el código de una clase para lanzar una excepción (u otro tipo de error) cuando un cliente que la usa lo viola.

La instrumentación realizada por nuestra herramienta consiste, por un lado, en enriquecer las precondiciones de los métodos mediante cláusulas que restringen su uso incorrecto, y por otro lado, en la adición de postcondiciones inferidas del *typestate* para asegurar que el estado alcanzado sea el esperado, favoreciendo el análisis modular.

4.1. Modelo formal

A diferencia del capítulo anterior, no es necesario introducir conceptos nuevos. Sin embargo haremos una pequeña modificación, o mejor dicho, una extensión a la estructura de una clase antes presentada.

Dada una clase C definimos su correspondiente clase enriquecida como una extensión de C, cuya estructura tiene la forma $\langle M, F, R, E, inv, init \rangle$, donde todos sus componentes denotan exactamente lo mismo que antes y se añade uno nuevo, tal que:

- E es el conjunto de postcondiciones (o *ensures clauses*) indexado por $m \in M$.

4.2. Algoritmo

Con la definición anterior, estamos en condiciones de presentar el algoritmo 4.1, encargado construir, dadas una clase y su abstracción, su correspondiente versión enriquecida.

Como primer paso se agrega una nueva variable privada $state \in S$, que contiene en todo momento el estado abstracto actual en que se encuentra una determinada instancia.

1: **procedure** `InstrumentClass`(C: $\langle M, F, R, inv, init\rangle$, EPA: $\langle \Sigma, S, S_0, \delta\rangle$) : $\langle M, F, R', E, inv, init\rangle$
2: $E_{ctor}(c) \leftarrow state(c) \in S_0 \quad \forall c$
3: $S_m^P \leftarrow \{s \in S \mid \delta(s, m) \neq \varnothing\} \quad \forall m$
4: **for each** method $m \in M$ **do**
5: $R'_m(c) \leftarrow R_m(c) \wedge state(c) \in S_m^P \quad \forall c$
6: $E_m(c) \leftarrow \bigwedge_{s \in S_m^P} state(c) = s \Rightarrow state(f_m(c)) \in \delta(s, m) \quad \forall c$

Algoritmo 4.1: Instrumentación de una clase.

El algoritmo establece la postcondición del constructor de la clase como el predicado que asegura que toda instancia nueva se encuentra en algún estado inicial. La expresión $state(c)$ se refiere al valor de la variable privada *state* de la instancia $c \in C$.

Retomemos el ejemplo de la pila presentado en el capítulo anterior (ver programa 3.1). Antes de seguir, analicemos primero su correspondiente EPA de la figura 4.1. Como se puede ver, cuenta con tres estados `S1` $= \{push\}$, `S2` $= \{push, pop\}$ y `S3` $= \{pop\}$ los cuales representan la pila vacía, con algunos elementos y llena respectivamente. Dado que la abstracción tiene un único estado inicial `S1` $= \{push\}$, tenemos que E_{ctor} es verdadero sólo para las instancias de la pila que se encuentren en el estado `S1`. De esta forma, se asegura que todas las instancias recién creadas se encuentran en el estado inicial.

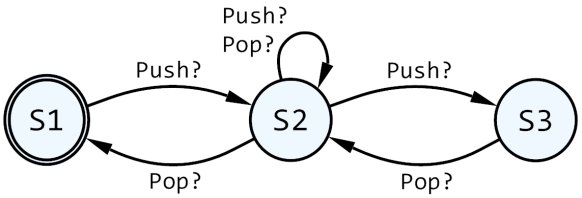

Figura 4.1: Abstracción correspondiente a una pila genérica de capacidad limitada. El estado inicial está marcado con un doble círculo exterior.

A continuación, el algoritmo precalcula el conjunto de partida S_m^P para todo método $m \in M$, el cual contiene los estados de partida de todas las transiciones etiquetadas con m. Luego, para cada estado de partida s correspondiente a m:

1. se agrega una nueva cláusula a la precondición del método, la cual es verdadera sólo cuando la instancia se encuentra en el estado de origen s.
2. se agrega una nueva cláusula a la postcondición del método, la cual es verdadera sólo cuando la instancia evoluciona del estado de origen s a algún estado destino $r \in \delta(s, m)$ tras la ejecución del método m.

Siguiendo con el ejemplo de la pila, tenemos que el conjunto de partida para cada método es:

- $S_{push}^P = \{\texttt{S1}, \texttt{S2}\} = \{\{push\}, \{push, pop\}\}$
- $S_{pop}^P = \{\texttt{S2}, \texttt{S3}\} = \{\{push, pop\}, \{pop\}\}$

Por lo tanto las precondiciónes y postcondiciones de cada método son instrumentadas tal que:

- R_{push} es verdadero sólo para las instancias de la pila que no estén llenas y se encuentren en los estados `S1` o `S2`.
- E_{push} es verdadero sólo para las instancias de la pila que, luego de la ejecución del método `Push`,
  - evolucionaron del estado de partida `S1` al estado destino `S2`.
  - evolucionaron del estado de partida `S2` al estado destino `S3`.
  - permanecieron en el estado `S2`.
- R_{pop} es verdadero sólo para las instancias de la pila que no estén vacías y se encuentren en los estados `S2` o `S3`.
- E_{pop} es verdadero sólo para las instancias de la pila que, luego de la ejecución del método `Pop`,
  - evolucionaron del estado de partida `S3` al estado destino `S2`.
  - evolucionaron del estado de partida `S2` al estado destino `S1`.
  - permanecieron en el estado `S2`.

La siguiente tabla resume los contratos luego de aplicar la instrumentación. Mediante la letra $d \in C$ denotamos la instancia resultante tras la ejecución de un método m a partir de una instancia dada c, es decir, $d = f_m(c)$.

Método	Requiere	Asegura
Constructor	$true$	$state(c) = 1$
Push	$count(c) < capacity(c)$ $state(c) \in \{1, 2\}$	$state(c) = 1 \Rightarrow state(d) = 2$ $state(c) = 2 \Rightarrow state(d) \in \{2, 3\}$
Pop	$count(c) > 0$ $state(c) \in \{3, 2\}$	$state(c) = 3 \Rightarrow state(d) = 2$ $state(c) = 2 \Rightarrow state(d) \in \{2, 1\}$

Al igual que el algoritmo de construcción, el pseudo código presentado en esta sección es un *template*, el cual establece *qué* cláusulas extra se deben agregar a la especificación de la clase sin indicar *cómo* resolverlas. Por lo tanto, es importante tener en cuenta que este algoritmo propone enriquecer la interfaz de la clase mediante la adición de cláusulas en las precondiciones y postcondiciones de los métodos. Sin embargo, es responsabilidad del programador que lo implemente analizar si los nuevos contratos se pueden verificar con la implementación original o si es necesario extenderla con este fin. Dependerá de los recursos propios del verificador estático con el que se esté trabajando (por ejemplo, si es capaz de inferir la información necesaria para verificar las postcondiciones a partir de las precondiciones).

En el próximo capítulo hablaremos sobre la implementación particular de nuestra propuesta y detallaremos *cómo* resolvimos cada uno de los puntos antes mencionados en los cuales los algoritmos no hacen hincapié.

Capítulo 5

Implementación

Siguiendo el trabajo realizado en [dCBGU11], donde las abstracciones son generadas utilizando *reachability queries*, nuestra implementación utiliza en cambio, el verificador estático de Code Contracts [ABF+09]. Por este motivo, antes de discutirla es necesario primero presentarlo brevemente.

Se trata de un proyecto de Microsoft Research que extiende todos los lenguajes de la plataforma .NET con las ventajas de la programación de diseño por contratos [Mey88], posibilitando su uso sin la necesidad de modificar los compiladores ya existentes. Su principal novedad radica en que los contratos son especificados por medio de construcciones propias de cada uno de los lenguajes de programación, adoptando la forma de llamadas a métodos de una API particular o expresiones booleanas que los codifican. Es justamente este motivo el que permite su integración sin la necesidad de desarrollar un compilador específico.

Code Contracts cuenta además con un conjunto de herramientas que procesan las especificaciones por contratos para generar automáticamente su documentación (`ccdoc`), realizar chequeos en tiempo de ejecución (`ccrewrite`) y en tiempo de compilación (`cccheck`). Debido a estos chequeos nuestra implementación se ve beneficiada tanto con verificaciones estáticas como dinámicas, resultando un enfoque híbrido que se complementa.

La verificación estática se realiza de forma modular, analizando cada método por separado. Para cada uno, se asume su precondición y se trata de verificar su postcondición. Así mismo, para cada invocación de método, se trata de verificar su precondición y se asume su postcondición. De esta forma se van propagando las postcondiciones de los métodos invocados hacia sus invocadores.

Para poder verificar los contratos, el analizador estático de Code Contracts trata de inferir del código del método bajo análisis la información necesaria, utilizando un motor de interpretación abstracta [FL09] en lugar de un demostrador de teoremas, común en este tipo de herramientas. Como ventajas podemos mencionar que el verificador estático es capaz de inferir automáticamente los invariantes de ciclo y su resultado es determinístico.

5.1. Construcción

El algoritmo 3.1 que construye la abstracción, presentado en el capítulo 3, toma decisiones según el resultado de ciertas expresiones lógicas. Por ejemplo, para saber si un método m pertenece a M_0^-, es decir, si m nunca está habilitado en ningún estado inicial, es necesario conocer el resultado de la expresión de la línea 5. De la misma forma, para saber si se debe agregar una transición de un estado ms a otro ns etiquetada con un método m ($ms \xrightarrow{m} ns$), es decir, si una instancia $c \in C$ que cumple con el invariante de estado de ms puede evolucionar a otra instancia $d \in C$ que cumple con el invariante de estado de ns luego de la ejecución del método m, es necesario conocer el resultado de la expresión de la línea 18.

Para poder evaluar dichas expresiones generamos y extendemos la clase C con métodos extra (*queries*), cada uno de los cuales tiene un propósito específico, para ser analizados posteriormente por el verificador estático de Code Contracts. De esta forma, transferimos entonces el problema de decibilidad de las expresiones lógicas a la verificación de las postcondiciones de cada uno de estos métodos, intencionalmente construidos para preservar su validez y equivalencia. Sin embargo, cabe mencionar que por lo general este es un problema indecidible por lo que no siempre el verificador estático puede llegar a un resultado concreto, en cuyo caso tenemos que lidiar con la incertidumbre como se vio al final del capítulo 3.

Las expresiones que necesitan ser evaluadas se encuentran en las líneas 5, 6, 8, 14, 15 y 18 del algoritmo 3.1. Las tres primeras líneas se encargan de descubrir los estados iniciales, que son aquellos estados en los que toda instancia de la clase se encuentra luego de ser creada por medio de un constructor. Debido a que los constructores no son más que métodos especiales, nuestra implementación no hace diferencia ni distinciones al momento de considerarlos. Por lo tanto, no es necesario detenerse en las líneas 5, 6 y 8 ya que son idénticas a las líneas 14, 15 y 18 respectivamente. Por su parte, cada una de ellas requiere la generación de métodos distintos, con lo cual es conveniente analizarlas por separado.

La expresión de la línea 14 intenta descubrir, para cada método $n \in M$, si es cierto que *nunca* se encuentra habilitado luego de la ejecución de un método dado $m \in M$ a partir de un estado dado $ms \in S$. De forma análoga, la expresión de la línea 15 intenta descubrir, para cada método $n \in M$, si es cierto que *siempre* se encuentra habilitado luego de la ejecución de un método dado $m \in M$ a partir de un estado dado $ms \in S$. Como se puede apreciar, ambas expresiones son muy similares, por lo tanto, como es de esperar, los métodos generados para resolverlas también lo son. En el algoritmo 5.1 se presentan los pseudo códigos de dichos métodos. La única diferencia entre ambos es la negación en la postcondición de la línea 3. Es importante aclarar que a pesar de su ubicación, la postcondición se evalúa al finalizar el método.

```
1: procedure QueryForLine14()
2:     requires inv_ms(c)
3:     ensures ¬R_n(d)
4:     d ← f_m(c)
```

```
1: procedure QueryForLine15()
2:     requires inv_ms(c)
3:     ensures R_n(d)
4:     d ← f_m(c)
```

Algoritmo 5.1: Pseudo código de los métodos generados para evaluar las expresiones de las líneas 14 y 15 del algoritmo 3.1 para una instancia $c \in C$ y métodos $m, n \in M$.

En el primer caso, si la postcondición es verificada por el analizador estático de Code Contracts, entonces toda instancia que satisface el invariante del estado ms *no satisface* la precondición del método n luego de la invocación de m. Formalmente, $\forall c.\ inv_{ms}(c) \Rightarrow \neg R_n(f_m(c))$ que es exactamente lo que queremos verificar.

En el segundo caso, si la postcondición es verificada por el analizador estático, entonces toda instancia que satisface el invariante del estado ms *satisface* la precondición del método n luego de la invocación de m. Formalmente, $\forall c.\ inv_{ms}(c) \Rightarrow R_n(f_m(c))$ que es exactamente lo que queremos verificar.

Retomando el ejemplo de la pila del programa 3.1, para saber si `Pop` pertenece a M^- luego de la invocación de `Push`, dado un estado $ms = \{push, pop\}$, extendemos la clase `Stack` con el método presentado en el programa 5.1.

Notar que las cláusulas que conforman la precondición corresponden al invariante del estado ms, como se lo calculó en el capítulo 3. Mientras que la postcondición corresponde a la negación de la precondición de `Pop`. En este ejemplo, el analizador estático devuelve como resultado que la postcondición es falsa, por lo que `Pop` no pertenece a M^-. Esto es debido a que inicialmente como mínimo $count = 1$, que tras la ejecución de `Push` quedaría $count = 2$, claramente mayor que cero.

Por otro lado, es importante tener en cuenta que el método `Push` recibe un parámetro, para lo cual decidimos que la mejor opción es que el método generado también lo reciba. De esta forma, se propagan los parámetros de los métodos invocados a sus invocadores, en lugar de fijar valores predeterminados como argumentos.

```
1 public void PushPop_Push_NotPop(T item) {
2     Contract.Requires(count > 0);
3     Contract.Requires(count < capacity);
4     Contract.Ensures(count <= 0);
5     Push(item);
6 }
```

Programa 5.1: Método para evaluar si `Pop` pertenece a M^- luego de la invocación de `Push`, dado un estado $ms = \{push, pop\}$.

Analicemos ahora la expresión de la línea 18. Su objetivo es decidir si se debe agregar una transición de un estado ms a otro ns etiquetada con un método m, es decir, $ms \xrightarrow{m} ns$. En el algoritmo 5.2 se presenta el pseudo código del método generado para resolver esta expresión.

```
1: procedure QueryForLine18()
2:         requires inv_ms(c)
3:         ensures ¬inv_ns(d)
4:         d ← f_m(c)
```

Algoritmo 5.2: Pseudo código del método generado para evaluar si $ms \xrightarrow{m} ns$ es una transición válida.

En esta ocasión, el razonamiento es un poco más complicado. Si la postcondición es verificada por el analizador estático de Code Contracts, entonces toda instancia que satisface el invariante del estado ms *satisface la negación* del invariante del estado

ns luego de la invocación de m. Formalmente, $\forall c.\ inv_{ms}(c) \Rightarrow \neg inv_{ns}(f_m(c))$ por lo que la transición no es válida.

Por el contrario, siguiendo con la discusión del capítulo anterior, si la postcondición *no* es verificada por el analizador estático, o si éste es incapaz de devolver una respuesta definitiva, entonces interpretamos que existe por lo menos una instancia que satisface el invariante del estado ms, así como también satisface el invariante del estado ns luego de la invocación de m. Formalmente, $\exists c.\ inv_{ms}(c) \wedge inv_{ns}(f_m(c))$ que es exactamente lo que queremos verificar para considerar válida la transición y alcanzable el estado ns.

Por ejemplo, dados los estados $ms = \{push, pop\}$ y $ns = \{push\}$, para saber si la transición $ms \xrightarrow{pop} ns$ es válida, extendemos la clase `Stack` con el método presentado en el programa 5.2.

Al igual que antes, las cláusulas que conforman la precondición corresponden al invariante del estado ms. Mientras que la postcondición corresponde a la negación del invariante del estado ns, calculado en el capítulo 3. Como se puede ver, la postcondición no es cierta para toda instancia que inicialmente se encuentre en el estado ms (considerar nuevamente el caso $count = 1$), con lo cual el analizador estático no puede verificarla, ocasionando que dicha transición se considere válida.

```
1 public void PushPop_Pop_Push() {
2     Contract.Requires(count > 0);
3     Contract.Requires(count < capacity);
4     Contract.Ensures(count != 0);
5     Pop();
6 }
```

Programa 5.2: Método para evaluar si $\{push, pop\} \xrightarrow{pop} \{push\}$ es una transición válida.

5.2. Instrumentación

El segundo paso de nuestra propuesta consiste en utilizar la información inferida al generar el *typestate* para verificar que los clientes de la API la usen correctamente. Esto se logra extendiendo la especificación original de la clase con restricciones que la obliguen a comportarse como lo indica su *typestate* subyacente. Luego, el código cliente puede ser verificado en tiempo de compilación utilizando el analizador estático de Code Contracts. Sin embargo, aquellos usuarios que desactiven esta opción, o en los casos donde la verificación no provea una respuesta definitiva (debido a potenciales falsos positivos), todavía pueden beneficiarse de los chequeos realizados en tiempo de ejecución.

Volviendo al ejemplo de la pila del programa 3.1, consideremos el método `Pop`, cuya versión enriquecida se presenta en el programa 5.3. Como se puede observar en la línea 2, la clase cuenta con una nueva variable privada `$state`, la cual se distingue de las otras variables ya existentes por medio de la anteposición del símbolo `$`, el cual no es permitido como prefijo en el lenguaje (pero sí a bajo nivel, donde la instrumentación es aplicada), para evitar potenciales conflictos de nombres. Su función, como se explicó en el capítulo 4, es contener en todo momento el estado actual.

```
1  public class Stack<T> {
2    private uint $state = 0;
3    // ...
4    public T Pop() {
5       Contract.Requires(count > 0);
6       Contract.Requires($state == 3 || $state == 2);
7       Contract.Ensures(old($state) != 3 || $state == 2);
8       Contract.Ensures(old($state) != 2 || $state == 2 || $state == 1);
9       try {
10         return data[--count];
11      } finally {
12         switch ($state) {
13            case 3: $state = 2; break;
14            case 2: $state = (count > 0 ? 2 : 1); break;
15         }
16      }
17   }
18 }
```

Programa 5.3: Método `Pop` luego de ser instrumentado.

Dicha variable es verificada en la precondición de cada método mediante la adición de nuevas cláusulas para asegurar que sean invocados correctamente. En el caso de `Pop`, esta verificación se realiza en la línea 6, restringiendo los estados en los cuales puede ser invocado según establece la abstracción de la figura 4.1. De esta forma, el método `Pop` sólo puede ser ejecutado en las instancias que se encuentren en los estados `S3` o `S2`.

Por otro lado, las líneas 7 y 8 especifican cómo es actualizada la variable `$state` en función de su valor anterior. La expresión `old($state)` se refiere al valor original de la misma antes de la ejecución del método (`Contract.OldValue<uint>($state)` en la sintaxis de Code Contracts). Es decir, establecen cuál es el nuevo estado alcanzado tras la invocación de `Pop`. Por ejemplo, si el estado original es `S3`, entonces el nuevo estado será `S2`.

Luego, las líneas 12-15 se encargan de actualizar efectivamente la variable `$state` con el objetivo de que el analizador estático de Code Contracts pueda verificar la nueva postcondición. En los casos donde la abstracción no es determinística, como sucede con el ejemplo de la pila, el nuevo estado alcanzado depende además de los valores de algunas otras variables privadas, como se puede apreciar en la línea 14. Es justamente por este motivo que la actualización de la variable `$state` debe realizarse después del cuerpo original del método. Una forma fácil de garantizarlo es utilizando la estructura de control `try-finally`, que asegura la ejecución del bloque `finally` luego del `try` en cualquier situación. Aún en el caso de producirse una excepción o retorno, como se ve en el ejemplo.

Finalmente, las líneas 5 y 10 provienen del código original del método `Pop`.

Como comentario al margen, es importante notar que las clausulas que constituyen la postcondición no son necesarias si la única intención es verificar el cumplimiento del *typestate* en tiempo de ejecución. Code Contracts utiliza estas clausulas para realizar la verificación estática modular en el código cliente que hace uso de la clase `Stack`.

5.3. Algunas consideraciones generales

En esta sección queremos aprovechar para mencionar algunos aspectos relevantes relacionados con la implementación en general.

El primero consiste en el soporte que tiene CONTRACTOR.NET para manipular tipos de datos genéricos, en cualquiera de los lenguajes de la plataforma .NET. Con esto nos referimos tanto a la etapa de construcción de la abstracción como a su posterior instrumentación, siendo un buen ejemplo la clase `Stack` anteriormente presentada.

Por otro lado, debido a la metodología modular en que el verificador estático de CODE CONTRACTS analiza y propaga los contratos de cada método, no siempre es suficiente invocarlos desde los métodos generados de forma automática en la etapa de construcción de la abstracción. Esto se debe a que el verificador estático infiere para cada método, información a partir del código de su cuerpo, con el objetivo de verificar la cláusulas que constituyen su postcondición. Sin embargo, dicha información inferida se pierde al no ser propagada junto con la postcondición en los lugares de invocación del método.

Para solventar este inconveniente, desarrollamos la posibilidad de insertar el cuerpo del método invocado *inline*. Es decir, en lugar de realizar una invocación, simplemente embebemos el código original del método junto con sus contratos. De esta forma, las cláusulas `requires` se transforman en `asserts`, mientras que las cláusulas `ensures` se transforman en `assumes`. Incluso, para no perder funcionalidad decidimos dejar ambas opciones y que sea el usuario quien elija en cada caso cual aplicar.

Otro aspecto interesante radica en la ausencia de una API que permita acceder desde código a las funcionalidades proporcionadas por CODE CONTRACTS. Por este motivo, nos vimos obligados a tener que ejecutar el verificador estático como un nuevo proceso y capturar el *output* de la consola generado por el mismo, para posteriormente *parsearlo* utilizando expresiones regulares y obtener así el resultado del análisis. Esta metodología obviamente no es la mejor, ya que este proceso tiene un costo adicional (*overhead*), reflejado en el tiempo total de ejecución de CONTRACTOR.NET, como se verá con mayor detalle en el capítulo 6. Peor aún, versiones posteriores de CODE CONTRACTS podrían cambiar el formato de su *output*, perdiendo compatibilidad con nuestra herramienta y ocasionando su mal funcionamiento.

De hecho, el mismo Francesco Logozzo[1], creador del verificador estático de CODE CONTRACTS, nos comentó que el mismo no esta diseñado para ser utilizado como motor de *back-end* en otras herramientas, sin embargo, tampoco descartó su potencial en este sentido.

Para finalizar, durante el desarrollo de nuestra propuesta, nos hemos encontrado con algunos errores presentes en la biblioteca CCI[2] de Microsoft, la cual utilizamos internamente para acceder y modificar los ejecutables (*assemblies*). Como consecuencia, nos vimos obligados a tener que corregirlos nosotros mismos para poder continuar con nuestro trabajo. Por supuesto, todas las correcciones fueron enviadas a través de la página web y validadas por los responsables del proyecto.

[1] Francesco Logozzo es un investigador de Microsoft Research, encargado de desarrollar el analizador estático de CODE CONTRACTS. En el año 2011 tuvimos la oportunidad de conocerlo personalmente cuando visitó nuestro departamento de computación para dictar un curso al respecto bajo el contexto de la ECI.

[2] Más información en: `http://cciast.codeplex.com`

5.4. Contractor.NET

Nuestra implementación consta de tres herramientas de igual funcionalidad, cada una concebida con un propósito específico. La primera y principal consiste en una extensión para el Visual Studio, lista para ser instalada y utilizada por el usuario final. La segunda herramienta se trata de un programa de consola para ser ejecutado en *batch* desde la línea de comandos, dando la posibilidad al usuario avanzado de integrarla a un proceso ya existente, como podría ser el de integración continua. Por último, la tercera consiste directamente en una API para que el usuario programador pueda utilizarla en sus propios proyectos de software.

Es importante mencionar que nuestra versión actual de Contractor.Net[3] implementa todas las funcionalidades presentadas en este trabajo, las cuales incluyen la generación automática de la abstracción, así como también la instrumentación; como fue explicado en los capítulos anteriores. Además, trabaja al nivel del *Common Intermediate Language* (CIL, a través de la biblioteca CCI de Microsoft) por lo que es capaz de manejar clases escritas en cualquiera de los lenguajes de la plataforma .NET.

5.4.1. Extensión para el Visual Studio

Luego de instalar Contractor.Net, los usuarios podrán hacer uso de las funcionalidades que la nueva ventana *Contractor Explorer*, presentada en la figura 5.1, ofrece dentro del Visual Studio.

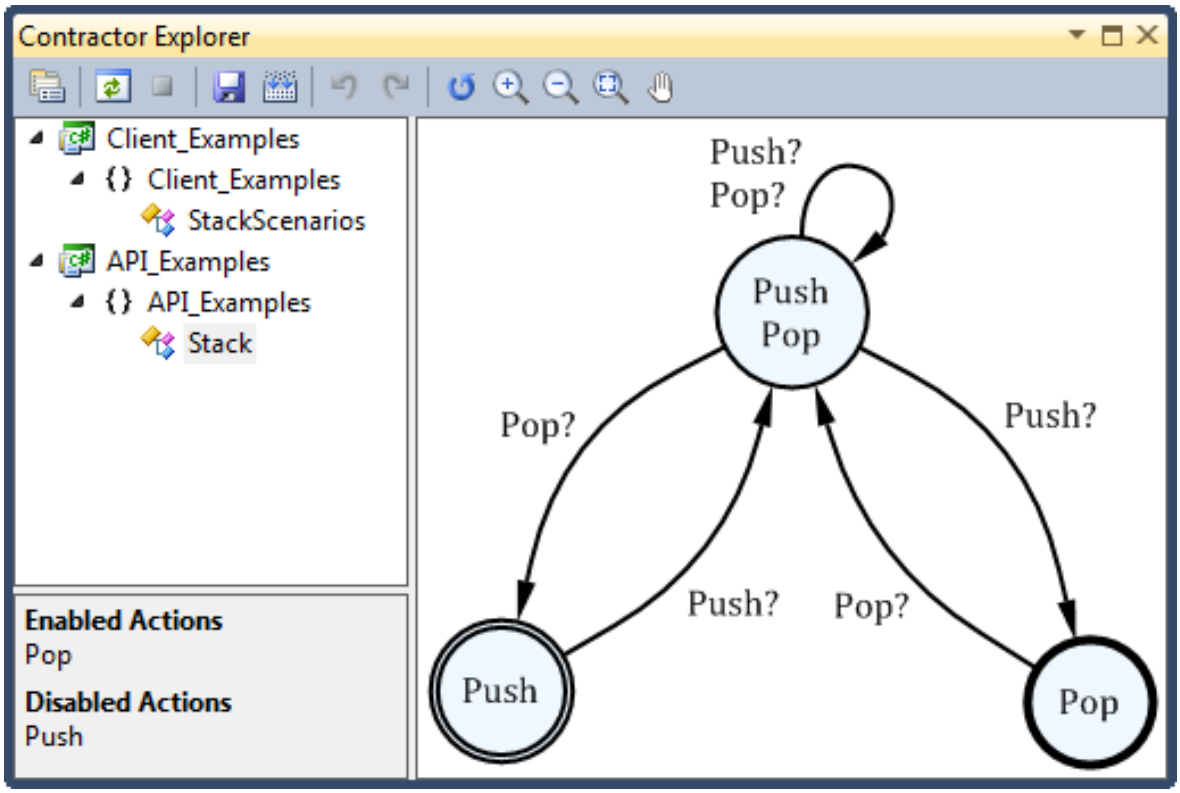

Figura 5.1: Ventana principal de Contractor.Net.

Desde allí, el usuario puede elegir del panel izquierdo qué clase analizar. Una vez

[3]Disponible en: `http://lafhis.dc.uba.ar/contractor.net`

seleccionada, el análisis comienza en *background*, permitiendo en todo momento su cancelación. A medida que se van encontrando nuevos estados y transiciones, el panel de la derecha se actualiza para presentar una versión preliminar de la abstracción generada (utilizando la biblioteca MSAGL[4] de Microsoft), de manera que el usuario pueda ver el resultado sin tener que esperar hasta que el análisis termine.

De forma adicional, se puede manipular el *typestate* resultante para reorganizar la disposición de los estados y transiciones a gusto, contando con funcionalidades básicas como deshacer, rehacer, *zooming*, *panning* y restaurar el *layout* original. Además, se pueden seleccionar los estados para obtener información relevante de cada uno, como por ejemplo, los métodos que se encuentran habilitados y deshabilitados.

Finalmente, el usuario tiene la posibilidad de exportar la abstracción generada, para lo cual se pueden seleccionar diversos formatos entre los que se incluyen gráficos vectoriales (EMF y WMF), mapas de bits (PNG, JPG, GIF y BMP), XML y Graphviz.

Una vez concluido el trabajo, se puede optar por generar la versión instrumentada de la clase seleccionada, enriquecida con la información inferida de la abstracción. En tal caso, el programa generará un nuevo ejecutable sin sobrescribir la versión original.

Por último, podemos mencionar que en todo momento se puede acceder a las opciones propias de la herramienta para modificarlas. Las mismas se encuentran integradas con el resto de las opciones del Visual Studio, como se muestra en la figura 5.2.

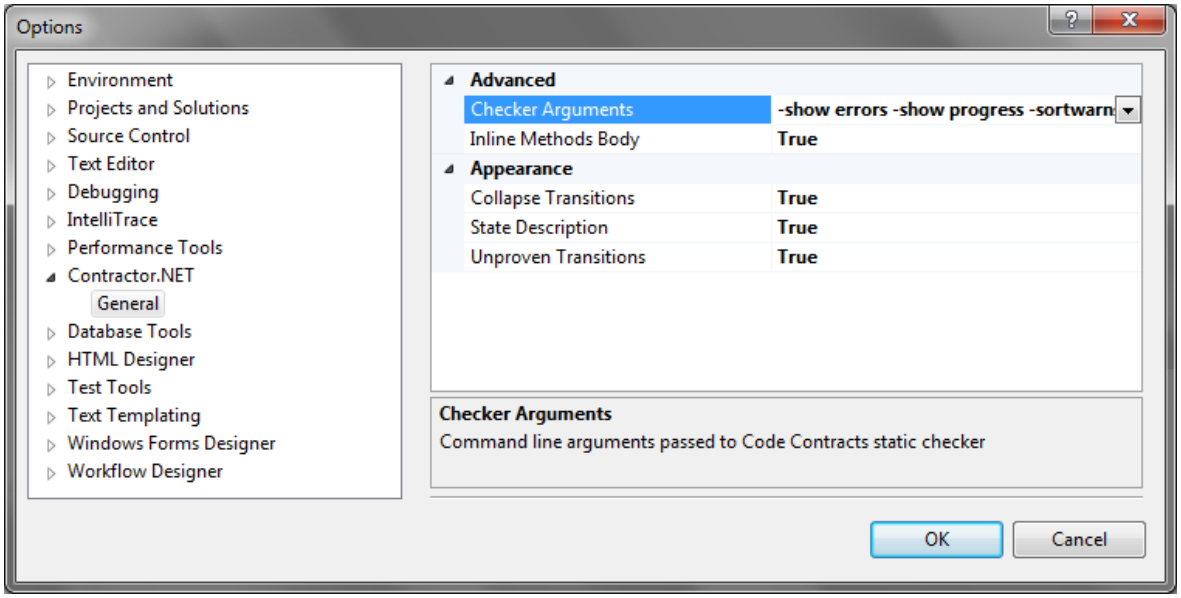

Figura 5.2: Opciones disponibles en CONTRACTOR.NET.

5.4.2. Consola para la línea de comandos

Como mencionamos anteriormente, esta versión de CONTRACTOR.NET está pensada para integrar la herramienta a procesos automáticos ya existentes, como podría ser MSBUILD[5]. El siguiente listado muestra la salida generada por la ayuda del programa al ser ejecutado desde la línea de comandos. Como se puede observar, se indican el

[4]Más información en: `http://research.microsoft.com/en-us/projects/msagl`
[5]Más información en: `http://msdn.microsoft.com/en-us/library/vstudio/dd393574.aspx`

modo de uso y las opciones disponibles con sus correspondientes valores por defecto y abreviaturas.

```
Contractor.NET Version 1.4.0.0
Copyright (C) LaFHIS - UBA. All rights reserved.

usage: <general-option>*

where <general-option> is one of
    -input <string-arg>          : Name of the input assembly to analyze [-i]
    -type <string-arg>           : Full name of the type to analyze [-t]
    -generateAssembly            : Generate the strengthened output assembly [-ga]
    -output <string-arg>         : Name of the strengthened output assembly [-o]
    -graph <string-arg>          : Directory used to store the output graphs [-g]
    -temp <string-arg>           : Directory used to store temporary files [-tmp]
    -cccheck <string-arg>        : Full path and file name were find cccheck.exe [-c]
    -cccheckArgs <string-arg>    : Command line arguments passed to cccheck.exe [-ca]
    -collapseTransitions (def=true) : Collapse transitions between states [-ct]
    -unprovenTransitions (def=true) : Distinguish unproven transitions with '?' [-ut]
    -inline (def=true)           : Inline methods body instead of method calls [-il]
    -stateDescription (def=true) : Show states descriptions [-sd]

To clear a list, use -<option>=!!

To remove an item from a list, use -<option> !<item>
```

En caso de no especificar la opción `type`, todas las clases públicas del *assembly* indicado mediante la opción `input` serán analizadas. Para generar la versión enriquecida del mismo es necesario especificar además la opción `generateAssembly`. El apéndice A muestra la salida producida por esta herramienta para los ejemplos presentados en el capítulo 6.

5.4.3. Biblioteca para programadores

Se trata de una API que expone todas las funcionalidades de Contractor.Net, para ser referenciada desde otros proyectos de software con el objetivo de facilitar su reutilización y extensión. Claros ejemplos de uso de esta biblioteca y su potencial son la extensión para el Visual Studio y la versión de consola de Contractor.Net, ambas presentadas en las secciones anteriores.

La figura 5.3 muestra el diagrama de clases exportadas por esta biblioteca. La clase `EpaGenerator` es la principal encargada de proveer toda la funcionalidad de Contractor.Net. La misma expone métodos para cargar y descargar *assemblies*, analizar todas las clases públicas para generar las abstracciones correspondientes a cada una de ellas, o simplemente especificar una en particular. Esta clase también es la encargada de instrumentar el *assembly* original para enriquecerlo con la información obtenida en la etapa de construcción de las abstracciones. A su vez, permite hacer un seguimiento del progreso del análisis mediante la suscripción a los eventos que expone, los cuales cuentan con información detallada del estado o transición que se acaba de descubrir. Una vez finalizado el análisis, provee además algunas métricas interesantes, mediante la clase `TypeAnalysisResult`, como es el tiempo total transcurrido o la cantidad total de transiciones que el verificador estático no pudo demostrar ni refutar. Es posible también modificar la configuración predeterminada utilizando la clase estática `Configuration`, la cual es consultada internamente por el generador de abstracciones.

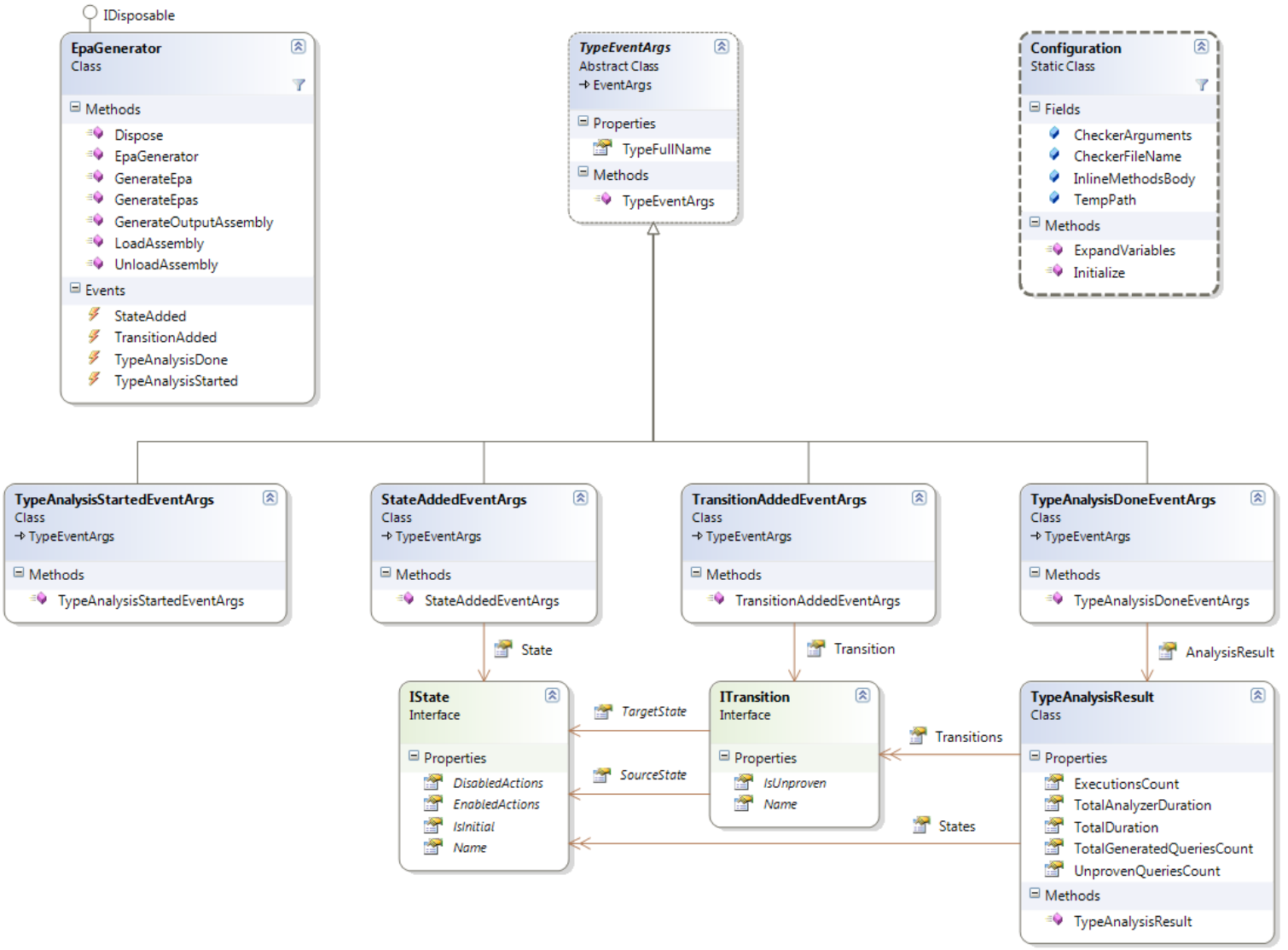

Figura 5.3: Diagrama de clases correspondiente a la API de Contractor.Net.

Capítulo 6

Resultados

En este capítulo comentaremos sobre los aspectos involucrados en la validación de nuestra herramienta. Para ello presentamos algunos ejemplos nuevos que nos servirán para analizar la utilidad de nuestra propuesta así como también la escalabilidad de la misma. Principalmente nos interesa responder, mediante ejemplos concretos, las siguientes preguntas:

- Utilidad
  - ¿El nivel de abstracción propuesto es adecuado para validar el modelo?
  - ¿Permite descubrir errores de implementación en el código fuente?
  - ¿Ayuda a identificar problemas en los requerimientos?
- Performance
  - ¿Es nuestra implementación lo suficientemente robusta para manejar programas reales complejos?
  - ¿Code Contracts es una tecnología adecuada para ser utilizada en este tipo de herramientas?

Los programas seleccionados para ser analizados con el objetivo de responder dichas preguntas son `Mp3Player`, `Microwave`, `VendingMachine`, `Elevator`, `ATM` y por último `File`. Adicionalmente consideramos también los ejemplos `Door` y `Stack`, discutidos en los capítulos anteriores. Es importante aclarar que dichos programas no intentan dar solución a los problemas reales asociados a cada uno de ellos, sino que constituyen una versión reducida y simplificada de los mismos, con el único objetivo de ser claros y sencillos para facilitar su entendimiento y posterior análisis, evitando así el grado de complejidad que conlleva cada uno de ellos en la realidad. Por este mismo motivo elegimos ejemplos que modelan objetos de uso común, ya conocidos y utilizados diariamente por todos nosotros. De esta forma, se facilita su comprensión y podemos enfocarnos en lo que realmente nos interesa, sin tener que perder demasiado tiempo explicando los detalles del funcionamiento de cada uno de ellos. Veamos brevemente en qué consisten.

El `Mp3Player` modela las funcionalidades de un reproductor de música convencional. Por su parte, la clase `Microwave` intenta modelar, de manera simplificada, el funcionamiento de un horno microondas. De forma análoga, la clase `VendingMachine` abstrae la funcionalidad básica de una máquina expendedora de productos. `Elevator` encapsula, con varias limitaciones, las características principales de un ascensor de pasajeros. La clase `ATM` se encarga de modelar, de forma muy reducida. el funcionamiento

de un cajero automático. Por último, la clase `File` encapsula las operaciones de lectura y escritura de un archivo.

Todos los programas aquí presentados se encuentran disponibles para descargar desde la página web de CONTRACTOR.NET[1].

La siguiente tabla comparativa resume información relevante sobre los ejemplos recién mencionados. Para cada uno de ellos, la columna *Líneas* contiene la cantidad total de líneas de código, mientras que la columna *Métodos* contiene la cantidad total de métodos públicos. A su vez, la columna *Contratos* contiene la cantidad total de contratos, es decir, precondiciones más invariantes de clase. Es importante remarcar que los ejemplos no cuentan con postcondiciones, debido a que no son necesarias para la generación de las abstracciones. Por último, la columna *Invariante de clase* indica si el ejemplo tiene definido algún invariante de tipo asociado.

Nombre	Líneas	Métodos	Contratos	Invariante de clase
`Door`	68	6	9	✓
`Stack`	40	2	4	✓
`Mp3Player`	67	5	11	✓
`Microwave`	64	5	8	✓
`VendingMachine`	53	4	7	✓
`Elevator`	67	5	14	✓
`ATM`	66	7	9	✓
`File`	51	4	6	-

En las próximas secciones nos concentraremos en cada uno de los ejemplos por separado para realizar un breve análisis de los mismos. Omitiremos las secciones correspondientes a los ejemplos `Door` y `Stack` debido a que ya fueron analizados en detalle en los capítulos anteriores.

6.1. Mp3 Player

Esta clase cuenta con una lista de canciones que permite reproducir mediante el método `Play`, pausar con `Pause` o bien detener mediante `Stop`. A su vez, da la opción de avanzar a la siguiente canción, o retroceder a la anterior, siempre que sea posible, utilizando los métodos `Next` y `Previous` respectivamente.

La figura 6.1 muestra la abstracción generada utilizando CONTRACTOR.NET. Al analizarla, lo primero que nos llamó la atención fue la gran cantidad de estados y transiciones presentes en la misma. Como se trata de un ejemplo pequeño supusimos que el EPA resultante sería sencillo, sin embargo, dicha suposición resultó errónea debido a que, si bien es cierto que la clase no contiene una gran cantidad de métodos

[1]Disponible en: `http://lafhis.dc.uba.ar/contractor.net`.

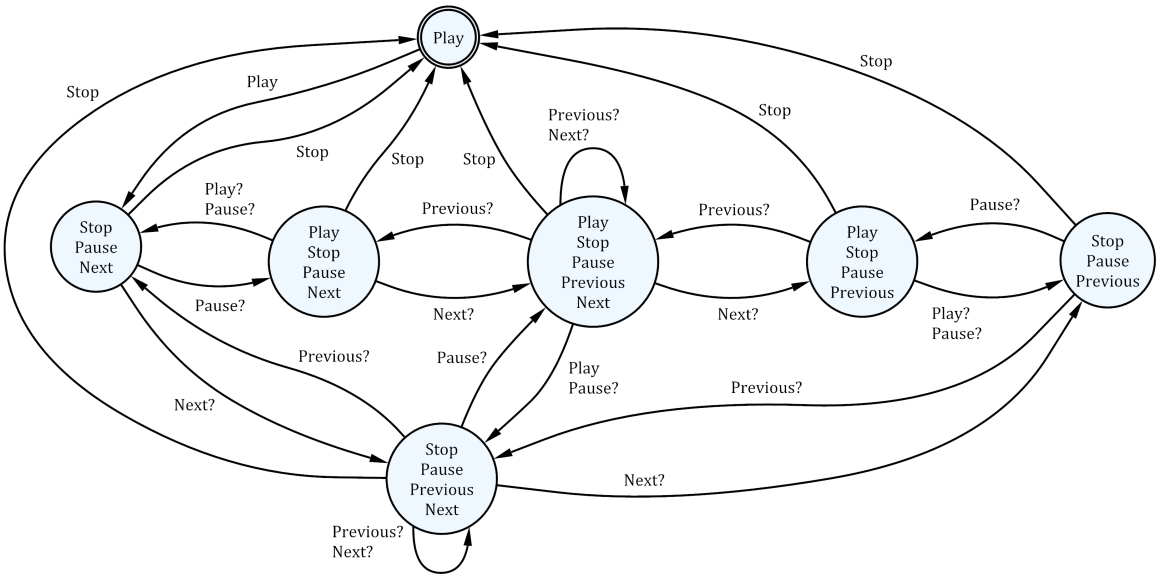

Figura 6.1: Abstracción correspondiente al controlador de un reproductor de música básico. El estado inicial está marcado con un doble círculo exterior.

públicos, los mismos están altamente relacionados entre sí. Por lo tanto, la complejidad del EPA obtenido se encuentra directamente condicionada por la interacción de sus métodos.

En este ejemplo se puede observar fácilmente que no produce el mismo resultado avanzar a la siguiente canción mientras se esta reproduciendo la actual, que avanzar a la siguiente mientras la canción actual se encuentra pausada. En el primer caso, la nueva canción comienza a reproducirse, mientras que en el segundo caso, el reproductor permanece en pausa. Esto se puede ver simulando las trazas `Play` $\rightarrow$ `Next` y `Play` $\rightarrow$ `Pause` $\rightarrow$ `Next`. Ambas llegan a distintos estados, los cuales están relacionados mediante el método `Pause`, lo que significa que se puede evolucionar de uno a otro luego de la ejecución de dicha acción. No es casualidad que sea justamente el método `Pause` el que los conecte entre sí, sino que por el contrario, se debe a que en realidad ambos estados representan casi la misma configuración, excepto que en uno el reproductor se encuentra pausado y en el otro no.

Un detalle curioso que se deduce de lo recién mencionado, y que también se puede observar a partir del gráfico, es el hecho de que tanto el método `Play` como `Pause` reanudan la reproducción de la canción actual cuando la misma se encuentra pausada. Es decir, el método `Pause` permite pausar y reproducir las canciones, en lugar de lanzar una excepción cuando el reproductor ya se encuentra en pausa. Esto se puede deducir notando que siempre que hay una transición `Pause` del estado ms al estado ns, también hay otra igual del estado ns al estado ms.

Por otro lado, similar a lo que sucede en el ejemplo del `Stack`, se pueden distinguir tres tipos de estados distintos, según cuál sea la canción actual: los que admiten configuraciones que tengan como actual la primer canción de la lista de reproducción, los que corresponden a configuraciones que tengan la última canción de la lista como actual, y finalmente, los que admiten configuraciones que tengan como actual cualquier otra canción intermedia. En este ejemplo hay dos estados de cada tipo debido a que la canción en cuestión puede estar siendo reproducida o encontrarse en pausa. También podemos observar que una vez finalizada la reproducción de la última canción, no se

continúa con la primera (es decir, la lista de reproducción no es cíclica).

Para finalizar el análisis, podemos mencionar que todos los estados tienen una transición `Stop` que lleva al estado inicial, por lo que podemos deducir fácilmente que es posible detener la reproducción en cualquier momento sin importar el estado del reproductor, y que además, la próxima canción a reproducir mediante `Play` será la primera de la lista.

6.2. Microwave

Esta clase encapsula las operaciones básicas de un horno microondas. Invocando el método `Start` se inicia la cocción, la cual, al finalizar el intervalo de tiempo pasado como parámetro, se detiene por medio del evento `Finish`. El usuario también tiene la opción de detener la cocción antes de finalizar el tiempo establecido, para lo cual el controlador cuenta con el método `Stop`. A su vez, por cuestiones de seguridad, la cocción también se debe detener cuando se abre la puerta y reanudarse al cerrarse la misma. Dichos eventos son capturados por los métodos `DoorOpen` and `DoorClosed` respectivamente.

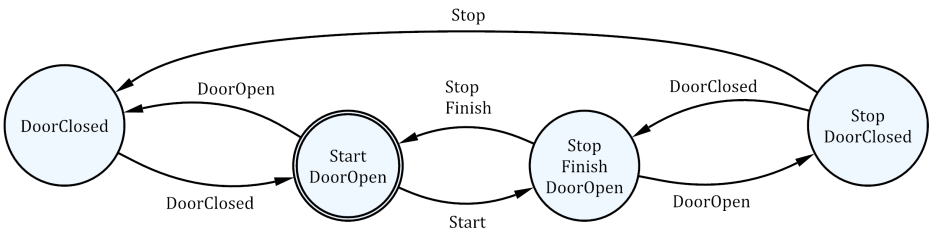

Figura 6.2: Abstracción correspondiente al controlador de un horno microondas básico. El estado inicial está marcado con un doble círculo exterior.

La figura 6.2 muestra la abstracción obtenida por Contractor.Net. En este ejemplo nos parece interesante analizar el siguiente escenario que constituye un típico caso de uso de un microondas. El usuario introduce la comida a calentar y como no está seguro cuánto tiempo es necesario dejarla, unos segundos antes de que la cocción termine, decide verificar si el tiempo transcurrido fue suficiente. Sin embargo, considera que todavía hace falta dejarla unos segundos más. El programa 6.1 implementa este escenario.

Luego se utiliza Code Contracts con el fin de verificar de forma estática el correcto funcionamiento del programa. Como es de esperar, debido a que los métodos de la clase `Microwave` no incluyen ninguna postcondición, el analizador estático no cuenta con la información necesaria para poder verificar el código cliente.

Sin embargo, dicha tarea se hace posible utilizando la versión enriquecida de la clase `Microwave`, la cual es el resultado de la instrumentación realizada por Contractor.Net. En este ejemplo en particular, podemos observar que en la línea 10 el cliente está intentando reanudar la cocción tras la verificación de la comida, provocando una violación de la precondición del método `Start`. El error surge debido a que el usuario no advirtió que la cocción se reanuda automáticamente luego de cerrar la puerta del horno microondas.

```
1  public void FoodNotReadyScenario() {
2      Microwave oven = new Microwave();
3
4      oven.DoorOpen();     // The user puts the food inside the oven
5      oven.DoorClosed();   // and then closes the door
6      oven.Start(90);      // Start cooking the food for 90 secs.
7      Thread.Sleep(60);    // Wait 60 secs.
8      oven.DoorOpen();     // The user verifys the food
9      oven.DoorClosed();   // and then decides that it is not ready
10     oven.Start(15);      // Resume cooking for another 15 secs.
11 }
```

Programa 6.1: Típico caso de uso de un horno microondas básico.

Este malentendido de la funcionalidad de la clase `Microwave` también podría haberse evitado inspeccionando cuidadosamente la abstracción de la figura 6.2. Al realizar un seguimiento paso a paso de la secuencia de llamadas a los métodos del controlador que realiza el código cliente presentado en el programa 6.1, llegamos a un estado donde el método `Start` no se encuentra habilitado mientras que el método `Stop` si lo está.

6.3. Vending Machine

Esta clase se encarga de modelar el funcionamiento básico de una máquina expendedora de productos. Para ello cuenta con los métodos `DisplayPrice` que muestra el precio de un determinado producto, `Buy` que se ejecuta cuando se realiza la compra de uno de los productos, devolviendo el cambio mediante el método `ReturnChange` en caso de ser necesario, y por último, `ReturnItem` que libera para su entrega el producto recién comprado. La figura 6.3 muestra la abstracción generada utilizando CONTRACTOR.NET.

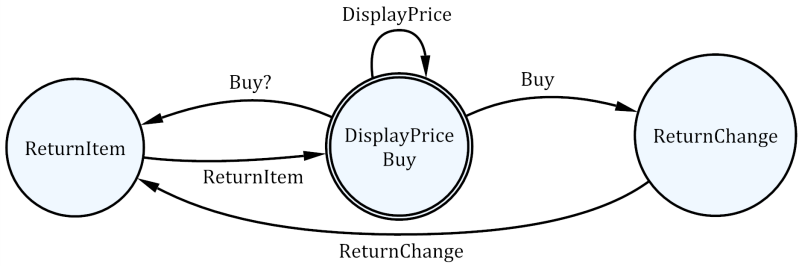

Figura 6.3: Abstracción correspondiente al controlador de una máquina expendedora de productos básica. El estado inicial está marcado con un doble círculo exterior.

6.4. Elevator

Esta clase abstrae la funcionalidad básica de un ascensor de pasajeros. El mismo puede ser requerido desde un piso particular mediante el método `Request`, ocasionando que el ascensor se ponga en marcha y dispare el evento `Arrive` cuando llegue a destino. Luego, se puede indicar el piso al que debe desplazarse utilizando el método `GoTo`.

Como medida de seguridad, el controlador cuenta con el requerimiento de verificar que no se sobrepase el límite máximo de peso que puede soportar el ascensor, para lo cual el método `Alarm` emite una alarma sonora de precaución, que es desactivada mediante el método `Safe` cuando el sobrepeso desaparece. La figura 6.4 muestra la abstracción obtenida por CONTRACTOR.NET.

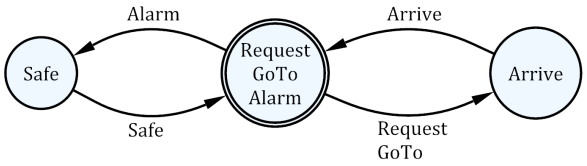

Figura 6.4: Abstracción correspondiente al controlador de un ascensor básico. El estado inicial está marcado con un doble círculo exterior.

6.5. ATM

Esta clase modela las operaciones principales de un cajero automático básico. Cada vez que se inserta una nueva tarjeta en la ranura del cajero, el evento `CardInserted` es invocado para mostrar la pantalla de ingreso de PIN. Luego, el usuario es autenticado mediante el método `Authenticate` del controlador, el cual le permite realizar distintas operaciones. Entre ellas se encuentra la posibilidad de cambiar el PIN mediante el método `ChangePin`, extraer o depositar dinero mediante los métodos `Extract` y `Deposit` respectivamente, y por último, el método `PrintTicket` permite imprimir un comprobante detallando el saldo y las operaciones realizadas. La sesión finaliza cuando la tarjeta es retirada del cajero por medio del método `RemoveCard`. La figura 6.5 muestra la abstracción generada utilizando CONTRACTOR.NET.

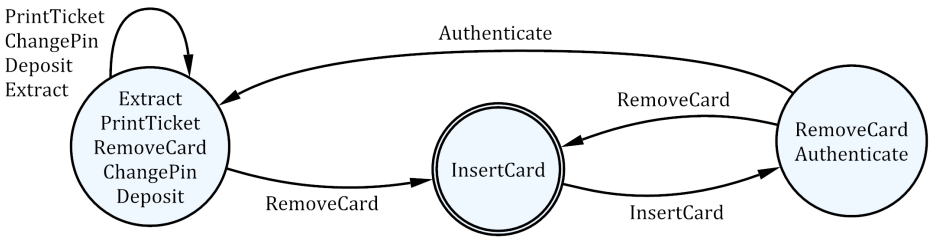

Figura 6.5: Abstracción correspondiente al controlador de un cajero automático básico. El estado inicial está marcado con un doble círculo exterior.

6.6. File

Esta clase encapsula el acceso a un archivo del *file system*. La misma permite abrir un archivo en modo lectura o escritura mediante el método `Open`, que recibe el modo deseado como parámetro. Luego, si el archivo fue abierto para lectura, se lo puede leer de forma secuencial invocando el método `Read`. De forma análoga, si el archivo fue abierto para escritura, se puede escribir información en el mismo mediante el método

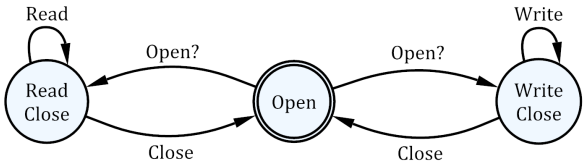

Figura 6.6: Abstracción correspondiente a una clase que permite leer y escribir archivos. El estado inicial está marcado con un doble círculo exterior.

`Write`. Una vez finalizado su uso, es necesario invocar el método `Close` para liberar el recurso tomado. La figura 6.6 muestra la abstracción obtenida por Contractor.Net.

6.7. Resultados

En esta sección presentaremos los resultados obtenidos para cada uno de los ejemplos elegidos y realizaremos un breve análisis de los mismos.

Para facilitar y agilizar la tarea, todas las mediciones realizadas fueron obtenidas utilizando la versión de consola de Contractor.Net, ideal para este tipo de análisis. El lector interesado puede encontrar un listado con la salida generada por esta herramienta en el apéndice A. Para ello se utilizó una computadora con procesador Intel i5 (doble núcleo con *hyper-threading*) y 4 GB de memoria RAM.

La siguiente tabla comparativa resume información relevante sobre la generación de las abstracciones correspondientes a los ejemplos presentados en las secciones anteriores. Para cada uno de ellos, la columna *Ejecuciones* contiene la cantidad total de invocaciones al verificador estático de Code Contracts. La columna *Precisión* indica el porcentaje de los métodos generados que pudieron ser verificados o refutados con certeza. La primer columna *Tiempo* contiene el tiempo total de ejecución del verificador estático de Code Contracts, considerando todas las corridas. La columna *Métodos generados* contiene la cantidad total de métodos generados automáticamente por Contractor.Net, como se explicó en el capítulo 5. Para finalizar, la última columna *Tiempo* contiene el tiempo total de ejecución de Contractor.Net, incluyendo el correspondiente a Code Contracts.

Lo primero que podemos notar a simple vista es que la gran parte del tiempo requerido por Contractor.Net para generar las abstracciones, corresponde al tiempo requerido por el analizador estático de Code Contracts para verificar las expresiones correspondientes a los métodos generados. Dicho resultado no es de sorprender debido a que nuestra implementación actual del algoritmo de construcción de los *typestates* es sincrónica, teniendo que esperar que el verificador estático termine antes de poder ejecutar el próximo análisis.

Por supuesto, mientras mayor sea la cantidad de invocaciones a Code Contracts por parte de Contractor.Net, mayor será el tiempo de duración total del mismo. Los casos `Door` y `Mp3Player` son buenos ejemplos. De forma similar, mientras más métodos sean generados por nuestra herramienta para ser verificados por el analizador estático de Code Contracts, mayor será el tiempo de duración total del mismo, y por lo tanto de Contractor.Net. Esta situación se evidencia en el ejemplo `ATM`, para el cual se generan 897 métodos para ser analizados entre 18 ejecuciones del verificador.

Como se puede observar en la tabla, la precisión del analizador estático de Code Contracts es bastante alta, pudiendo verificar y refutar casi todas las expresiones correspondientes a los métodos generados sin problemas, habiendo 3 de los 8 casos con el 100 % de certeza, siendo el `Stack` con el 77 % el de mayor incertidumbre.

Nombre	Code Contracts			Contractor.Net	
	Ejecuciones	Precisión	Tiempo	Métodos generados	Tiempo
`Door`	32	98 %	01′ 18″	208	01′ 23″
`Stack`	10	77 %	00′ 21″	27	00′ 22″
`Mp3Player`	50	95 %	03′ 08″	744	03′ 16″
`Microwave`	18	100 %	00′ 47″	99	00′ 49″
`VendingMachine`	10	99 %	00′ 22″	105	00′ 23″
`Elevator`	12	100 %	00′ 38″	128	00′ 39″
`ATM`	18	100 %	01′ 13″	897	01′ 17″
`File`	12	98 %	00′ 21″	129	00′ 22″

La siguiente tabla comparativa resume información relevante sobre las abstracciones correspondientes a los ejemplos presentados en las secciones anteriores. Para cada uno de ellos, la columna *Posibles* contiene la cantidad total de estados posibles, que es exponencial respecto de la cantidad de métodos públicos (2^M). La columna *Alcanzables* contiene la cantidad total de estados efectivamente alcanzables desde los iniciales, incluyéndolos. De forma análoga, la columna *Iniciales*, contiene justamente la cantidad total de estados iniciales. Por su parte, la columna *Posibles* contiene la cantidad total de transiciones posibles entre los estados alcanzables (S^2), mientras que la columna *Válidas* contiene la cantidad total de transiciones válidas dentro de las posibles, incluyendo las inciertas. Finalmente, la columna *Inciertas*, contiene justamente la cantidad total de transiciones que el verificador estático de Code Contracts no pudo verificar ni refutar. Dichas transiciones son distinguidas en los diagramas mediante el signo ?.

Es importante notar que los fragmentos alcanzables de las abstracciones generadas por Contractor.Net contienen significativamente menos estados que el espacio completo de 2^M estados posibles. Ejemplos claros son `ATM` que cuenta sólo con 3 estados alcanzables de los 128 posibles, y `Door` que cuenta sólo con 6 de los 64 posibles.

De forma similar, los fragmentos alcanzables de las abstracciones generadas por Contractor.Net contienen menos transiciones válidas que el total de S^2 transiciones posibles entre los estados alcanzables. Ejemplos claros son `Door` que cuenta sólo con 15 transiciones válidas de las 36 posibles, y `Microwave` que cuenta sólo con 8 de las 16 posibles.

Algo interesante para remarcar es el hecho de que todas las transiciones inciertas resultaron ser efectivamente válidas, lo que indica que es una buena idea incluirlas en las abstracciones. Por supuesto, cada una de ellas debe ser evaluada manualmente por el programador para validarla.

Nombre	Estados			Transiciones		
	Posibles	Alcanzables	Iniciales	Posibles	Válidas	Inciertas
`Door`	64	6	1	36	15	1
`Stack`	4	3	1	9	6	6
`Mp3Player`	32	7	1	49	28	20
`Microwave`	32	4	1	16	8	0
`VendingMachine`	16	3	1	9	5	1
`Elevator`	32	3	1	9	5	0
`ATM`	128	3	1	9	8	0
`File`	16	3	1	9	6	2

Relacionando la información de las tres tablas presentadas, podemos observar que los tiempos de ejecución de nuestra herramienta no sólo dependen de la cantidad de métodos públicos, sino que también influye el tamaño de las abstracciones generadas. Esto se puede ver fácilmente considerando el caso del `Mp3Player`, para el cual CONTRACTOR.NET tarda significativamente más tiempo en generar la abstracción comparado con el resto de los ejemplos (poco mas de 3 minutos), debido a la cantidad de estados abstractos alcanzables y transiciones válidas que también son superiores (7 y 28 respectivamente).

Otro aspecto que se puede observar inspeccionando las tablas, es la relación entre la precisión del analizador estático de CODE CONTRACTS y la cantidad de transiciones inciertas presentes en las abstracciones. Por ejemplo, si consideramos los casos `Microwave`, `Elevator` y `ATM` que no presentan transiciones inciertas, podemos ver que todos cuentan con el 100 % de certeza en la segunda tabla, lo cual es lógico. Un ejemplo más interesante para analizar es el del `Stack`, que tiene todas sus transiciones válidas como inciertas, y su correspondiente nivel de certidumbre es del 77 %. Es importante remarcar que si bien sus 6 transiciones válidas son inciertas, hay 3 de las 9 posibles que son inválidas.

Para finalizar, queremos mencionar que no hemos realizado un análisis exhaustivo respecto de la calidad de la verificación de código cliente instrumentado. Esto es debido principalmente a la complejidad que conlleva realizar dicha evaluación. Pensamos que la manera más interesante y adecuada para llevarla a cabo sería implementar un generador aleatorio de clientes a partir de las abstracciones generadas por CONTRACTOR.NET. Luego, dichos clientes tendrían que ser analizados utilizando CODE CONTRACTS dos veces: primero haciendo referencia a la versión original de la API; y en segundo lugar, haciendo referencia a la versión de la API instrumentada por nuestra herramienta. De esta forma, podríamos comparar los resultados obtenidos y extraer algunas métricas. Sin embargo, creemos que la implementación de un generador automático de clientes excede los objetivos y el alcance de nuestro trabajo.

Capítulo 7

Trabajo relacionado

Como se mencionó anteriormente, este proyecto esta basado en el trabajo realizado en [dCBGU11], donde a diferencia de nuestra implementación, las abstracciones son generadas utilizando *reachability queries* para ser analizadas por algún *software model checker* como Blast [BHJM07].

Desde el punto de vista de la construcción del *typestate*, nuestra propuesta está relacionada con los enfoques que sintetizan de forma estática interfaces seguras para ser utilizadas en clientes a partir de un programa dado [AČMN05, GP09, HJM05]. 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 [DF04, BR02] 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.

Es posible considerar nuestra propuesta como una forma de abstracción de predicados [Uri99]. 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 [LY92, GS97, GGSV02]. 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 [LMS07] 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 (*mining of temporal specifications*) [GS08, LMP08, DKM+10], 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.

Desde el punto de vista de la verificación de código cliente, los modelos de [HJM05] pueden ser muy conservadores y no considerar válidos usos legales de la API. En contraposición, nuestra propuesta consiste en una sobre-aproximación del espacio de estados abstractos, pudiendo llegar a aceptar alguna secuencia inválida en el cliente. Sin embargo consideramos que aún así, el programador cuenta con más herramientas para

detectar posibles errores. Además, es importante remarcar que nuestra implementación tiene un enfoque híbrido, ya que todos los errores que hayan logrado pasar desapercibidos al realizar la verificación estática, serán descubiertos luego en tiempo de ejecución. Este comportamiento híbrido viene dado por la utilización de Code Contracts, como se explicó en el capítulo 5.

Para finalizar, desde el punto de vista del enriquecimiento de la especificación utilizando información de *typestate*, nuestra propuesta está relacionada con las técnicas de monitoreo dinámico de *typestates* [BLH08]. Consideramos que el enfoque más parecido al nuestro es el presentado en [KBAK09]. El mismo, consiste en una extensión del lenguaje de modelado para Java (*Java Modeling Language* o simplemente JML) para incluir anotaciones de *typestate* explícitas. Luego, dichas anotaciones son traducidas automáticamente al JML tradicional siguiendo una estrategia similar a la nuestra. Existen también enfoques híbridos como el adoptado por Clara [Bod09] que realizan tanto verificación estática como dinámica, complementándose. En este caso, trata de verificar estáticamente todos los aspectos de monitoreo, dejando para ejecutar luego en *runtime* únicamente aquellos que no pudieron ser verificados. Sin embargo, esta propuesta requiere que el programador escriba manualmente los aspectos que se van a encargar de monitorear el *typestate*.

Capítulo 8

Conclusiones

En este trabajo presentamos CONTRACTOR.NET, una herramienta que permite construir, de manera estática y automática, modelos abstractos de comportamiento, en forma de *typestates*, a partir del código fuente de un programa.

El modelo es generado utilizando un nivel de abstracción denominado *enabledness-preserving*, el cual expresa de forma concisa, pero representativa, el comportamiento que una determinada clase tendrá en tiempo de ejecución. Con este objetivo, hemos implementado un algoritmo para construir dichos modelos, el cual utiliza internamente el analizador estático de CODE CONTRACTS como motor de decisión para verificar ciertas expresiones lógicas. Además, hemos mostrado cómo se pueden utilizar las abstracciones generadas para validar y facilitar el entendimiento del código fuente subyacente por parte del programador, posibilitando la identificación de problemas existentes en el mismo.

Por otro lado, mostramos cómo se pueden usar estas abstracciones, de forma complementaria, para enriquecer automáticamente la especificación por contratos original de una clase, permitiendo tanto la verificación estática como dinámica de código cliente utilizando CODE CONTRACTS.

Creemos importante resaltar que hemos podido analizar y comprender de forma fácil y rápida el comportamiento de cada una de las clases presentadas en el capítulo anterior, sin la necesidad de mostrar una sola línea de código de sus correspondientes implementaciones. Más aún, tampoco ha sido necesario ejecutarlas para conocer sus propiedades y características. En muchos casos, bastó simplemente con inspeccionar la abstracción generada por CONTRACTOR.NET para obtener una idea clara y precisa del dominio que el programador intentó modelar, o mejor aún, del dominio que el código realmente está modelando.

Es justamente por esta razón que consideramos que este tipo de abstracciones pueden ser útiles como documentación adicional de una determinada API, 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.

8.1. Trabajo a futuro

Son muchas las mejoras y extensiones que se pueden realizar a la implementación actual de nuestra propuesta. Entre ellas, creemos que la más importante para abordar a corto plazo es la construcción de un algoritmo paralelo para la generación de abstracciones. Como se explicó en el capítulo 5, dicho algoritmo invoca al analizador estático de Code Contracts en varias ocasiones para evaluar las expresiones lógicas que determinan los estados alcanzables y sus transiciones. Sin embargo, actualmente dichas invocaciones se realizan de forma secuencial y sincrónica, lo cual se ve reflejado en el tiempo de ejecución final de nuestra herramienta. En el capítulo 6, pudimos observar que la mayor parte de este tiempo corresponde al análisis realizado por el verificador estático de Code Contracts. Por lo tanto, consideramos que la construcción de un algoritmo paralelo puede mejorar de forma significativa la escalabilidad de Contractor.Net.

Otro tema importante para analizar tiene que ver con las precondiciones de los métodos generados durante la etapa de construcción de la abstracción. Como se explicó en el capítulo 5, dados dos estados abstractos ms y ns, y un método público m, para saber si $ms \xrightarrow{m} ns$ es una transición válida, se genera un método a ser verificado por Code Contracts cuya precondición es el invariante del estado ms. Recordemos que el invariante de un estado abstracto es un predicado que hace referencia, entre otras cosas, a las precondiciones de todos los métodos públicos, ya sea que se encuentren habilitados o no en dicho estado. El problema surge cuando las precondiciones de estos métodos hacen referencia a sus parámetros. En tal caso, la precondición del método generado también hará referencia a los mismos, lo cual es un problema porque no los conoce. Una posible solución podría ser incluir todos los parámetros en cada método generado. Otra posibilidad podría ser eliminar las cláusulas de las precondiciones que se refieran a los parámetros, asumiendo que existen valores para los mismos que las verifican, como se explica en [dCBGU11].

Respecto de la usabilidad y funcionalidad que provee nuestra implementación, creemos que se sería interesante mejorar la interfaz gráfica de la extensión para el Visual Studio, incorporando a la información de cada estado su correspondiente invariante. También creemos que sería útil permitirle al usuario cambiar los nombres de los estados para que sean más descriptivos.

Por otro lado, nos gustaría evaluar la posibilidad de realizar la instrumentación de las clases directamente al nivel del código fuente, como alternativa a la instrumentación actual, que trabaja al nivel del código intermedio. De esta forma el programador podría modificar los nuevos contratos generados para enriquecer aún más la especificación de la clase.

Para finalizar, el aspecto más importante que queremos abordar a largo plazo es analizar el desempeño de nuestra propuesta en casos de estudio reales, de mayor tamaño y complejidad. Pensamos que las bibliotecas que implementan protocolos pueden ser buenos candidatos para estudiar en mayor profundidad la utilidad y escalabilidad de Contractor.Net.

Apéndice A

Salida de la versión de consola

El siguiente listado muestra la salida generada por la versión para línea de comandos de Contractor.Net correspondiente a los ejemplos presentados anteriormente en el capítulo 6.

```
Contractor.NET Version 1.4.0.0
Copyright (C) LaFHIS - UBA. All rights reserved.

Starting analysis for type API_Examples.ATM
Analysis for type API_Examples.ATM done
        Code Contracts analysis total duration: 00:01:12.8031643
        Code Contracts analysis precision:      100%
        Code Contracts executions:              18
        Total duration:                         00:01:16.6583846
        Generated queries:                      897 (0 unproven)
        States:                                 3 (1 initial)
        Transitions:                            8 (0 unproven)

Starting analysis for type API_Examples.Elevator
Analysis for type API_Examples.Elevator done
        Code Contracts analysis total duration: 00:00:37.5581482
        Code Contracts analysis precision:      100%
        Code Contracts executions:              12
        Total duration:                         00:00:39.4172545
        Generated queries:                      128 (0 unproven)
        States:                                 3 (1 initial)
        Transitions:                            5 (0 unproven)

Starting analysis for type API_Examples.Microwave
Analysis for type API_Examples.Microwave done
        Code Contracts analysis total duration: 00:00:46.5806640
        Code Contracts analysis precision:      100%
        Code Contracts executions:              18
        Total duration:                         00:00:49.1168093
        Generated queries:                      99 (0 unproven)
        States:                                 4 (1 initial)
        Transitions:                            8 (0 unproven)

Starting analysis for type API_Examples.Stack
Analysis for type API_Examples.Stack done
        Code Contracts analysis total duration: 00:00:21.0442034
        Code Contracts analysis precision:      77%
        Code Contracts executions:              10
        Total duration:                         00:00:22.4522842
        Generated queries:                      27 (6 unproven)
        States:                                 3 (1 initial)
        Transitions:                            6 (6 unproven)

Starting analysis for type API_Examples.VendingMachine
Analysis for type API_Examples.VendingMachine done
        Code Contracts analysis total duration: 00:00:21.5682336
        Code Contracts analysis precision:      99%
```

```
        Code Contracts executions:              10
        Total duration:                         00:00:22.9043101
        Generated queries:                      105 (1 unproven)
        States:                                 3 (1 initial)
        Transitions:                            5 (1 unproven)

Starting analysis for type API_Examples.Door
Analysis for type API_Examples.Door done
        Code Contracts analysis total duration: 00:01:18.4414866
        Code Contracts analysis precision:      98%
        Code Contracts executions:              32
        Total duration:                         00:01:23.3017646
        Generated queries:                      208 (3 unproven)
        States:                                 6 (1 initial)
        Transitions:                            15 (1 unproven)

Starting analysis for type API_Examples.File
Analysis for type API_Examples.File done
        Code Contracts analysis total duration: 00:00:20.7411865
        Code Contracts analysis precision:      98%
        Code Contracts executions:              12
        Total duration:                         00:00:22.3412779
        Generated queries:                      129 (2 unproven)
        States:                                 3 (1 initial)
        Transitions:                            6 (2 unproven)

Starting analysis for type API_Examples.Mp3Player
Analysis for type API_Examples.Mp3Player done
        Code Contracts analysis total duration: 00:03:08.4787802
        Code Contracts analysis precision:      95%
        Code Contracts executions:              50
        Total duration:                         00:03:15.7631970
        Generated queries:                      744 (36 unproven)
        States:                                 7 (1 initial)
        Transitions:                            28 (20 unproven)

Done!
```

Bibliografía

[ABF+09] M. Andersen, M. Barnett, M. Fähndrich, B. Grunkemeyer, K. King, F. Logozzo, V. Patel, and D. Zuniga. Code contracts, 2009.

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

[BHJM07] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *STTT*, 9:505–525, 2007.

[BLH08] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *FSE '08*, pages 36–47, 2008.

[Bod09] E. Bodden. Clara: a framework for implementing hybrid typestate analyses. Technical report, Darmstadt University of Technology, 2009.

[BR02] T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02*, pages 1–3, 2002.

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

[dCBGU10] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Contractor for code validation. Technical report, DC. UBA, 2010.

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

[DF04] R. DeLine and M. Fahndrich. Typestates for objects. *ECOOP'04 (LNCS)*, pages 465–490, 2004.

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

[FBL10] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *SAC'10*, pages 2103–2110, 2010.

[FL09] M. Fähndrich and F. Logozzo. Clousot: a language agnostic abstract interpretation-based static analyzer for .net, 2009.

[GGSV02] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 112–122. ACM, 2002.

[GP09] D. Giannakopoulou and C.S. Păsăreanu. Interface generation and compositional verification in JavaPathfinder. In *FASE '09*, pages 94–108, 2009.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *Computer aided verification*, pages 72–83. Springer, 1997.

[GS08] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE '08*, pages 51–60, 2008.

[HJM05] T.A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *FSE '05*, pages 31–40, 2005.

[KBAK09] T. Kim, K. Bierhoff, J. Aldrich, and S. Kang. Typestate protocol specification in JML. In *SAVCBS '09*, pages 11–18. ACM, 2009.

[LMP08] D. Lorenzoli, L. Mariani, and M. Pezzê. Automatic generation of software behavioral models. In *ICSE '08*, pages 501–510, 2008.

[LMS07] L. Liu, B. Meyer, and B. Schoeller. Using contracts and boolean queries to improve the quality of automatic test generation. *Tests and Proofs*, pages 114–130, 2007.

[LY92] D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 264–274. ACM, 1992.

[Mey88] B. Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.

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