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

ContractorJ: validando el comportamiento de clases de Java

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Patricio Palladino

Director: Diego Garbervetsky

Buenos Aires, 2017

CONTRACTORJ: VALIDANDO EL COMPORTAMIENTO DE CLASES DE JAVA

El diseño de un módulo de software tiene como uno de sus objetivos disminuir el acoplamiento temporal del mismo, el cual consiste en relaciones de orden entre las invocaciones de los métodos o funciones del artefacto diseñado. Dichas restricciones dificultan el razonamiento sobre este, lo que hace su implementación y uso más propensos a errores.

Sin embargo, existen situaciones donde la naturaleza del problema o requerimientos no funcionales de la solución derivan en módulos de software con cierto acoplamiento temporal. Esta situación se ve agravada por la falta, total o parcial, de especificación o documentación al respecto.

Para atacar este problema se desarrolló el concepto de *Enabledness-Preserving Abstractions*, o *EPAs*. Una *EPA* es un modelo de comportamiento similar a un *typestate*, que refleja las dependencias temporales internas de una pieza de software. Estas tiene como objetivo ayudar al programador a validar su modelo mental sobre cómo debe comportarse un módulo que está implementando, y pueden ser utilizadas posteriormente como documentación del mismo.

En este trabajo presentamos una extensión de *EPAs* que contempla la salida excepcional de los distintos métodos, llamada *Permissive Enabledness-Preserving Abstraction* o *PEPA*, junto a una herramienta que genera de forma automática dichas abstracciones a partir de clases de *Java*.

CONTRACTORJ: VALIDATING JAVA CALSSES' BEHAVIOUR

One of the goals when designing a software module is to avoid or decrease its' methods temporal coupling, which refers to restrictions on their valid orders of invocation. Such restrictions make reasoning about them more difficult, and turn their implementation and use more error-prone.

There are situations where the problem being solved, or non-functional requierements of the software being built, lead to software modules with some temporal coupling. To make things worse, usually there's incomplete or non-existent documentation about this.

Different techniques have been developed to help programmers deal with this situation. On of those are *Enabledness-Preserving Abstractions*, or *EPAs*. An *EPA* is a behaviour model similar to a *typestate*, that preserves the ordering in which methods can be called. These abstractions can be helpful as documentation, and can be used to contrasts a programmers' mental model of a software artifact against its implementation.

This work addresses some limitations of such abstractions, presenting an extension of its model, called *Permissive Enabledness-Preserving Abtractions* or *PEPAs*. This extension is able to analyze a wider case of examples, giving cleaner results when the module's contract are poorly specified and errors may arise from calling one of its methods.

Índice general

1. INTRODUCCIÓN

Las módulos de software con requerimientos no triviales respecto al orden en que deben invocarse sus métodos abundan en la práctica, en especial en forma de implementaciones de protocolos o APIs. Estos requerimientos complican el razonamiento, la codificación y el uso de los mismos. Se suman a estos problemas la falta total o parcial de especificación al respecto, lo cual inhibe la verificación y validación tanto de ellos como del código que los utiliza.

En respuesta a esta situación se han desarrollado técnicas que extraen modelos formales a partir piezas de código con el objetivo de verificar su correcto uso o validar su comportamiento. Una de estás técnicas es la generación de *EPAs* [Cas+11; Cas+13].

Una *EPA* es una máquina de estados finita que comprime todas las secuencias posibles de invocaciones a métodos o funciones de una pieza de software. Estas son construidas a un nivel de abstracción cuyo objetivo es preservar el estado de habilitación de conjuntos de operaciones y las transiciones entre los mismos. Dicho de otra manera, es un modelo que cocienta el potencial espacio de estados de ejecución de un módulo o clase de acuerdo a qué métodos están habilitados. Se muestra en la figura 1.1 la *EPA* de una clase que representa un archivo. Se observan dos estados no-iniciales en la misma, el primero que sólo permite abrir el archivo, y el segundo que tiene habilitadas el resto de las operaciones.

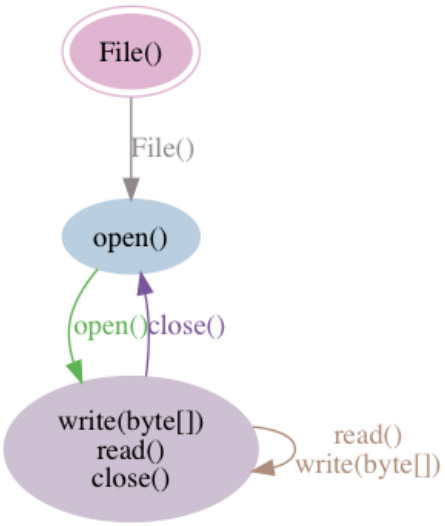

Fig. 1.1: EPA de un modelo de un archivo

Estas abstracciones suelen ser lo suficientemente compactas como para ser comprendidas por un observador, y resultan útiles para validar el modelo mental que tiene este sobre el artefacto analizado. En particular, son de gran utilidad para el implementador del módulo en cuestión, quien puede corroborar que existen exclusivamente las transiciones entre estados que él desea.

Existen herramientas para generar *EPAs* de forma automática a partir de código *C*[Cas+11; Cas+13] y *C#* [ZG12; LG16] que cuente con invariantes de representación y las precondiciones de sus operaciones, expresadas de forma explícita en el mismo lenguaje de programación.

Si bien estas herramientas funcionan, las plataformas en que están desarrolladas limitan la aplicación de las mismas. Contar con una implementación de un generador de *EPAs* para esta plataforma ampliaría el horizonte de aplicaciones de las mismas, ya que podrían integrarse con desarrollos de terceros.

Por otro lado, los generadores de *EPAs* actuales, así como el modelo formal de *Enabledness-Preserving Abstractions* suponen el correcto funcionamiento de cada operación en caso de cumplirse sus precondiciones. Una *EPA* no brinda información explícita sobre los casos donde el módulo analizado arroja excepciones. Tampoco contempla la posibilidad de que un defecto en el código o la especificación rompan el invariante, lo cual limita su utilidad como herramienta de *debugging*.

1.1. Objetivo

Este trabajo cuenta con tres objetivos. El primero es implementar un generador de *EPAs* en Java, con el afán de acercar este desarrollo a otros proyectos de investigación y tecnologías en ingeniería formal de software.

Por otro lado, nos interesa extender el modelo de *EPAs* explicitando la posibilidad de que una operación rompa el invariante o arroje una excepción, incluso si se encuentra habilitada por la aproximación a su precondición expresada por el implementador del módulo analizado. Llamamos a este nuevo modelo *Permissive Enabledness-Preserving Abstraction* o *PEPA*.

Por último, se desea mostrar un prototipo de una posible aplicación de este desarrollo al combinarlo con proyectos de terceros. Para esto se creó un programa en el cual se infieren los contratos de un módulo a analizar y se genera su *PEPA* a partir de código sin anotar.

1.2. Trabajo relacionado

Nuestro trabajo está basado en [Cas+13] y [LG16]. Donde las principales diferencias con estos radica en la elección de *Java* como lenguaje de análisis y plataforma de desarrollo.

Asimismo, nuestra técnica está relacionada con enfoques que sintetizan *typestates* [DF01] o interfaces [Alu+05; GP09; HJM05] a partir de un programa. Cualquier secuencia de métodos que no sea aceptada por nuestra abstracción, no será permitida por un programa que la codifique. Sin embargo, en este tipo de propuestas el foco está puesto en la verificación modular [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.

Nuestra propuesta también está relacionada con las técnicas de minería de especificaciones temporales [GS08; LMP08; Dal+10], las cuales producen a partir de trazas un autómata de estados finitos que describe cómo son utilizadas un conjunto de operaciones. El enfoque está puesto en la generación de casos de prueba y la verificación del código cliente, por lo que, de nuevo, no pueden ser inspeccionados de manera sencilla. Más aún, estas técnicas suelen ser dinámicas y dependen fuertemente de la calidad y cantidad de las trazas utilizadas para el análisis, lo cual no ocurre con la generación de *PEPAs*. Sin embargo, en el capítulo 7 se hace uso de técnicas similares para inferir los contratos de un artefacto de código y así poder generar su *Permissive Enabledness-Preserving Abtraction*.

1.3. Estructura de la tesis

En el capítulo 2 se presenta un ejemplo ilustrativo de cómo se genera una *EPA*, junto a las limitaciones que ésta abstracción tiene.

En el capítulo 3 se presenta el modelo formal de *EPAs* y un algoritmo de generación de ellas. A su vez se hace una breve introducción a *Corral*, la herramienta de verificación utilizada en nuestra implementación del algoritmo.

Para superar las limitaciones del modelo de *EPAs* se presenta en el capítulo 4 una extensión del mismo llamada *PEPAs*. Ésta relaja el concepto de precondición, de modo que se contemple la posibilidad de que sean incompletas o erróneas.

Se implementó un generador de *PEPAs* para *Java*, *ContractorJ*, el cual es descripto en el capítulo 5. Se analizan ejemplos representativos y los resultados obtenidos en la generación de *PEPAs* con *ContractorJ* en el capítulo 6.

El capítulo 7 pretender ser un ejemplo de las ventajas que tiene traer la generación de *EPAs* o *PEPAs* a la plataforma *Java*. En este se describe cómo haciendo uso de otros dos proyectos de investigación, *Randoop* y *Daikon*, pueden generarse *PEPAs* a partir de código sin contratos explícitos.

Finalmente, en el capítulo 8 se presentan las conclusiones de este trabajo, y en el último se proponen distintas alternativas de trabajo a futuro.

2. MOTIVACIÓN

El objetivo de esta sección es presentar un escenario simple y de fácil entendimiento para mostrar como el uso de *EPAs* ayuda a la comprensión de un artefacto de software, y las limitaciones de las mismas.

2.1. Validando el modelo subyacente de una pieza de software

Supongamos que se está desarrollando el controlador de un brazo robótico encargado de llenar, cargar y despachar cajas en una fábrica.

Uno de los primeros pasos del desarrollo del software es modelar las entidades que nos interesan, junto a las operaciones que pueden llevarse a cabo con éstas. En el caso del controlador del brazo robótico, se debe poder tomar una nueva caja vacía, cargarla, vaciarla, abrirla, cerrarla, y despacharla. Es importante notar que existen restricciones sobre la funcionalidad que debe presentar el dispositivo, por ejemplo, no se puede cerrar una caja vacía, ni despachar una abierta.

En este trabajo nos enfocamos en modelos de software desarrollado en *Java*, por lo que nuestro controlador será implementado como una clase de este lenguaje. Por lo tanto, contará con métodos para cada una de sus operaciones:

- **tomarCaja():** Toma una nueva caja, abierta y vacía, y lo coloca en el área de trabajo donde será cargada antes de ser despachada.
- **cargar(int pesoACargar):** Carga una pieza de mercadería en la caja, indicando el peso de la misma.
- **vaciar():** Remueve todo el contenido de la caja actual.
- **cerrar():** Cierra la caja con la que se está trabajando.
- **abrir():** Abre la caja actual.
- **despachar():** Despacha la caja, removiéndola de la zona de trabajo y dejándola lista para ser enviada a destino.

En la figura 2.1 puede verse el código de una clase Java que modela nuestro controlador. Por cada uno de los métodos que operan sobre el brazo robótico, se encuentra también un método booleano que indica si puede o no ejecutarse dicha acción. Es decir, dado un método m, sólo puede llamarse si m_pre retorna **true**.

```
public class Controlador {

  public boolean hayCaja;
  public boolean cajaCerrada;
  public boolean cajaVacia;

  public Controlador() { hayCaja = false; }

  public void tomarCaja() {
    hayCaja = true; cajaCerrada = false; cajaVacia = true;
  }

  public void cargar(int peso) { cajaVacia = false; }
  public void vaciar() { cajaVacia = true; }
  public void cerrar() { cajaCerrada = true; }
  public void abrir() { cajaCerrada = false; }
  public void despachar() { hayCaja = false; }

  // Contratos:
  public boolean inv() { return !hayCaja || !(cajaVacia && cajaCerrada); }

  public boolean tomarCaja_pre() { return !hayCaja; }

  public boolean cargar_pre() { return hayCaja && !cajaCerrada; }

  public boolean vaciar_pre() {
      return hayCaja && !cajaCerrada && !cajaVacia;
  }

  public boolean cerrar_pre() {
      return hayCaja && !cajaCerrada && !cajaVacia;
  }

  public boolean abrir_pre() { return hayCaja && cajaCerrada; }

  public boolean despachar_pre() { return hayCaja && cajaCerrada; }
}
```

Fig. 2.1: Código del controlador del brazo robótico

Utilizando *ContractorJ* podemos generar una *EPA* de nuestra clase, la cual puede observarse en la figura 2.2.

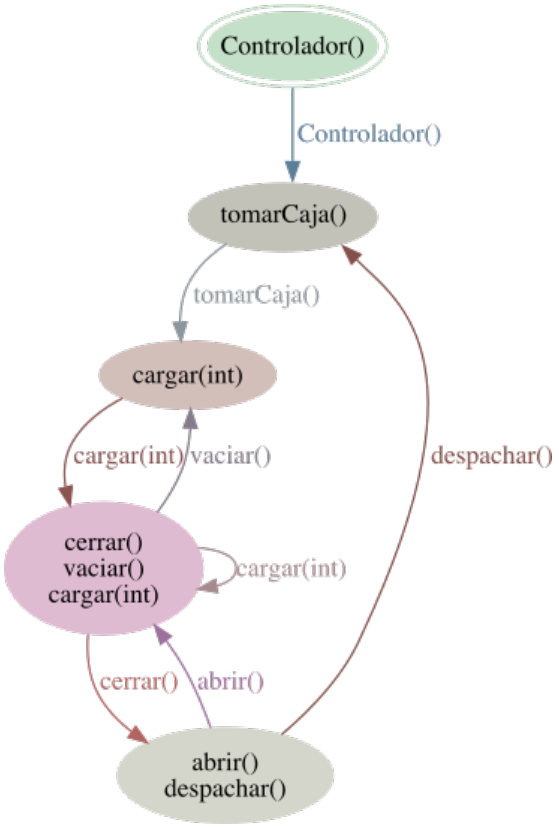

Fig. 2.2: EPA del controlador del brazo robótico

En ésta abstracción puede observarse, de manera resumida, el orden en que se pueden llamar a los métodos del controlador. Dada una traza de un programa, la *EPA* puede simular la misma como una serie de transiciones entre estados, ignorando las partes del programa que no le incumben, así como también los detalles de implementación de la clase analizada.

A modo de ejemplo mostramos en la figura 2.3 una posible ejecución, donde han sido atenuados los colores de los aspectos de la *EPA* que no son de interés. La ejecución comienza en un estado donde solo puede tomarse una nueva caja, y tras hacerlo la *EPA* pasa al estado donde únicamente puede cargarse, y luego, a pesar de haber otras operaciones habilitadas, es cerrada y despachada, volviendo al estado inicial.

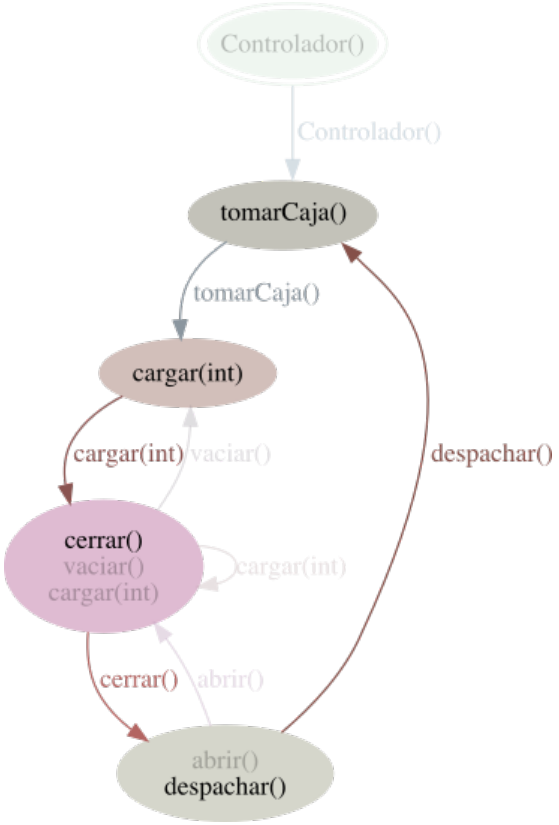

Fig. 2.3: Representación de una ejecución en la EPA

Es la posibilidad de realizar este seguimiento de una corrida de manera sencilla, como también la detección de transiciones o estados indeseados, lo que motiva el estudio de las *EPAs* y brinda una ayuda el desarrollador.

2.2. Limitaciones de las *EPAs*

Las *EPAs* cuentan con un número de limitaciones que pueden disminuir su utilidad. La primera de ellas es que admiten trazas inválidas, ya que se ignoran los aspectos particulares de cada configuración, agrupándolas únicamente según que métodos están habilitados.

En este trabajo nos enfocamos en otras dos limitaciones que se explican a continuación.

Como se vio en la figura 2.1, para generar una *EPA* se requiere que el programador exprese de manera explícita tanto el invariante de representación de su clase como las precondiciones de cada uno de los métodos. Sin embargo esto no es una tarea sencilla, ya que en la práctica no siempre se cuenta con el conocimiento suficiente sobre el problema como para poder hacerlo. Es esperable que el implementador desarrolle tanto el código como los contratos que este cumple de manera iterativa, por lo que se pasará por versiones incompletas y/o erróneas.

Esto conflictúa con el hecho de que el modelo de *EPAs* supone que estos contratos y la

implementación de la clase son correctos. Es decir, que no contempla la posibilidad de que la invocación de un método pueda romper el invariante o terminar de manera excepcional en caso de cumplirse sus precondiciones.

Supongamos que durante el desarrollo del controlador del brazo robótico el programador comete un error al codificar la precondición del método **cerrar()**, de modo que la expresa como se muestra en la figura 2.4. En este caso el brazo podrá cerrar una caja vacía, lo cual va en contra del invariante.

```
public boolean cerrar_pre() {
  return hayCaja && !cajaCerrada;
}
```

Fig. 2.4: Código de una precondición defectuosa

Utilizando esta versión defectuosa del código corrimos *ContractorJ* y obtuvimos la *EPA* que se ve en la figura 2.5. En esta se puede observar el método **cerrar()** habilitado inmediatamente después de tomar una nueva caja, pero sin embargo no se tiene información de qué ocurre al invocarlo, y no es evidente que hacerlo rompería el invariante. Notar que si el método contara también con transiciones validas de **cerrar()** a partir de tal estado, este error no sería visible de forma alguna.

Este resultado en particular es producto de implementación que utilizamos, pero podría variar para otras, ya que la situación no está contemplada por el modelo de las *EPAs*.

La otra limitación del modelo que afronta este trabajo es la falta de conocimiento que las *EPAs* tienen respecto a la terminación excepcional o errónea de una invocación a un método. Una

Definición 2.2.1. requiere que el programador exprese las precondiciones del método de manera correcta, lo cual no siempre ocurrirá, por lo que queremos tener en cuenta posibles errores.

Sin embargo, las precondiciones no siempre son expresadas de forma correcta en la práctica, por lo que tanto el usuario como el implementador de un módulo pueden verse beneficiados de la inclusión de posibles salidas excepcionales en un modelo de comportamiento del mismo. En particular algo que no suele encontrarse documentado es en qué estado queda una pieza de software luego de una invocación no exitosa de un procedimiento y cuales de sus métodos pueden ser invocados, lo cual motiva la extensión de las *EPAs* para incluir esto.

A modo de ejemplo, se introdujo un defecto en el código del controlador ya visto, como se puede observar en la figura 2.6. En esta versión hay un peso máximo de carga que puede ser soportado. Sin embargo, esto no está expresado en las precondiciones del método, por lo que es posible invocarlo de manera correcta y que se lance una excepción.

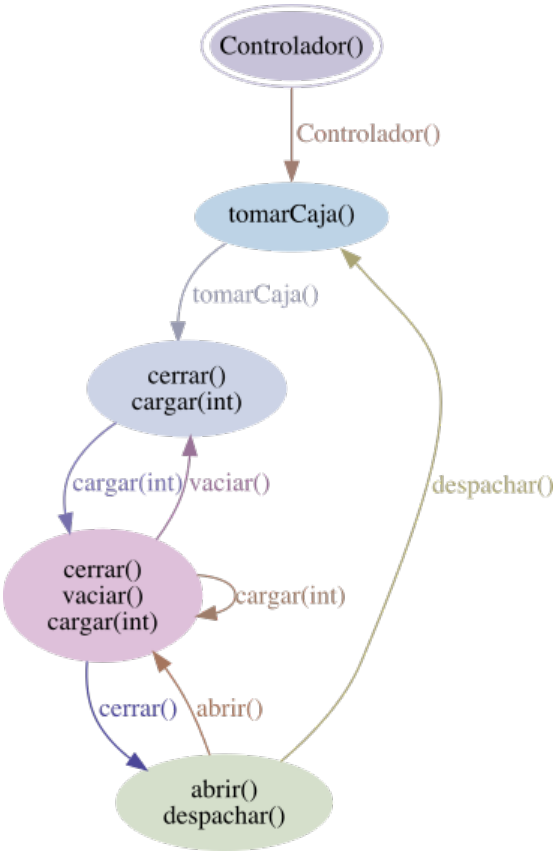

Fig. 2.5: EPA del controlador defectuoso del brazo robótico

```
public void cargar(int peso) {
  cajaVacia = false;

  if (peso > 1000) {
    throw new RuntimeException("Carga demasiado pesada");
  }
}
```

Fig. 2.6: Código de un método que lanza una excepción

Se corrió nuevamente *ContractorJ* con ésta versión del módulo, y se obtuvo la misma *EPA* que en la versión sin modificar, por lo que no ayudó a que se detecte este error. Nuevamente es necesario aclarar que este resultado es circunstancial, ya que al no contemplarse en el modelo de las *EPAs*, cada implementación es libre de actuar a criterio.

A lo largo de este trabajo se revisará el modelo de comportamiento con el que venimos trabajando, dando lugar a las *Permissive Enabledness-Preserving Abtractions*, y se describirá una implementación de un generador de estas.

3. PRELIMINARES

3.1. Enabledness-Preserving Abstractions

Se presenta en esta sección la definición formal de *EPAs*, y su algoritmo de construcción, los cuales serán extendidos más adelante para dar lugar a una abstracción más permisiva.

Ésta sección tiene como objetivo introducir los conceptos involucrados de forma breve. Se recomienda al lector consultar [Cas+11] para encontrar una explicación más profunda.

3.1.1. Modelo formal

Como primer paso para poder analizar formalmente artefactos de código, abstrayéndose de los detalles de implementación de cada plataforma de software, definimos la semántica de los mismos como transformaciones sobre un conjunto $\mathbb{C}$ de configuraciones del sistema. Por configuración se entiende al estado interno del módulo bajo consideración en cada momento de la ejecución de un programa que lo utilice, así como también a otros posibles elementos de la memoria del sistema que se relacionen con este.

Definimos entonces a la semántica de una pieza de código como un *Action System*.

Definición 3.1.1 (Action System)**.** Un *Action System* es una estructura $AS = \langle Act, F, R, inv, init \rangle$ donde:

- $Act = \{a_1, ..., a_n\}$ es un conjunto finito de etiquetas, que llamamos acciones. Se corresponden con las operaciones disponibles en el módulo analizado.
- F es un conjunto de funciones indexadas por los miembros de Act. Por cada etiqueta a, la función $F_a : \mathbb{C} \times \mathbb{Z} \to (\mathbb{C} \cup \{\bot\})$ toma una configuración y un parámetro y retorna una nueva configuración, o $\bot$ en caso que la invocación que representa no termine.

  Se restringe, sin pérdida de generalidad, el modelo a funciones con sólo un parámetro de tipo entero.
- R es un conjunto de precondiciones indexadas por las etiquetas de Act. Por cada etiqueta a, $R_a : \mathbb{C} \times \mathbb{Z} \to \{\textbf{true}, \textbf{false}\}$ indica si el método al que a referencia se encuentra habilitado para ser llamado en una configuración y con un parámetro dado.
- $inv : \mathbb{C} \to \{\textbf{true}, \textbf{false}\}$ es el invariante de representación del módulo.
- $init : \mathbb{C} \to \{\textbf{true}, \textbf{false}\}$ indica si la configuración es inicial.

Volviendo al ejemplo del controlador del brazo robótico, su *Action System* tendría los siguientes elementos:

- $Act = \{tomarCaja,\ cargar,\ vaciar,\ cerrar,\ abrir,\ despachar\}$.
- $F = \{F_{tomarCaja},\ F_{cargar},\ F_{vaciar},\ F_{cerrar},\ F_{abrir},\ F_{despachar}\}$.
- $R = \{R_{tomarCaja},\ R_{cargar},\ R_{vaciar},\ R_{cerrar},\ R_{abrir},\ R_{despachar}\}$

$$R_{tomarCaja} = \neg hayCaja$$

$$R_{cargar} = hayCaja \wedge \neg cajaCerrada$$

$$R_{vaciar} = hayCaja \wedge \neg cajaCerrada \wedge \neg cajaVacia$$

$$R_{cerrar} = hayCaja \wedge \neg cajaCerrada \wedge \neg cajaVacia$$

$$R_{abrir} = hayCaja \wedge cajaCerrada$$

$$R_{despachar} = hayCaja \wedge cajaCerrada$$

- $inv = hayCaja \Rightarrow \neg(cajaVacia \wedge cajaCerrada)$
- $init = \neg hayCaja$

Luego, definimos la semántica de un *Action System* como un sistema de transición etiquetado (*Labelled Transition System* o *LTS*) finito y determinístico.

Definición 3.1.2 (Semantica de un *Action System*)**.** Dado un *Action System* $AS = \langle Act, F, R, inv, init\rangle$ su semántica está provista por un *LTS* $L = \langle \mathbb{A}, \mathbb{S}, \mathbb{S}_0, \Delta\rangle$, que satisface:

- $\mathbb{A} = Act \times \mathbb{Z}$
- $\mathbb{S} = \{c \in \mathbb{C} \mid inv(c) = \textbf{true}\}$
- $\mathbb{S}_0 = \{c \in \mathbb{S} \mid init(c) = \textbf{true}\}$
- $\forall c \in \mathbb{S}, a \in Act, p \in \mathbb{Z}. R_a(c,p) = \textbf{true} \wedge F_a(c,p) = c' \wedge inv(c') \Rightarrow \Delta(c,(a,p)) = c'$.
- Δ no está definido para cualquier otro caso.

Notar que los estados de este *LTS* son las configuraciones donde el invariante es válido. Más adelante se relajará esta restricción a la hora de definir el modelo de *PEPAs*.

La semántica anterior tiene una cantidad infinita de estados, por lo que no es adecuada para el objetivo de validar contra el modelo mental del usuario. Para salvar esto definimos una relación de equivalencia en $\mathbb{C}$ y formulamos las *EPAs* haciendo uso de esta.

Definición 3.1.3 (Enabledness-equivalencia)**.** Dado un *Action System* AS y $c_1, c_2 \in \mathbb{C}$, decimos que son *Enabledness-equivalentes*, y lo notamos $c_1 \equiv_e c_2$, sii para cada etiqueta a de AS $\exists p \in \mathbb{Z}. R_a(c_1,p) = \textbf{true} \iff \exists p' \in \mathbb{Z}. R_a(c_2,p') = \textbf{true}$.

Por ejemplo, en el caso del controlador del brazo robótico, dadas las configuraciones

$$g_1 = \langle hayCaja = \textbf{false}, cajaCerrada = \textbf{false}, cajaVacia = \textbf{true} \rangle$$

$$g_2 = \langle hayCaja = \textbf{false}, cajaCerrada = \textbf{true}, cajaVacia = \textbf{false} \rangle$$

podemos afirmar que son equivalentes, es decir $g_1 \equiv_e g_2$, pues ambas tienen sólo $tomarCaja$ habilitado.

Finalmente, definimos la abstracción que nos interesa del siguiente modo.

Definición 3.1.4 (Enabledness-Preserving Abstraction)**.** Dado un *Action System* $AS = \langle Act, F, R, inv, init \rangle$ y su *LTS* $L = \langle \mathbb{A}, \mathbb{S}, \mathbb{S}_0, \Delta \rangle$, decimos que $M = \langle \Sigma, S, S_0, \delta \rangle$ es una *Enabledness-Preserving Abstractions* o *EPA* de AS sii existe una función total $\alpha : \mathbb{S} \to S$ tal que

- $\alpha(\mathbb{S}_0) \subseteq S_0$
- $\forall c \in \mathbb{S}, a \in Act, p \in \mathbb{Z}.R_a(c,p) = \textbf{true} \Rightarrow \alpha(\Delta(c,(a,p))) \in \delta(\alpha(c), a)$.
- $\forall c_1, c_2 \in \mathbb{S}.c_1 \equiv_e c_2 \iff \alpha(c_1) = \alpha(c_2)$

Cabe aclarar que se hace abuso de notación en ésta definición, utilizando α sobre conjuntos para referirse también a la extensión natural de esta.

Se puede encontrar en [Cas+13] una caracterización de las *EPAs*, y su demostración de correctitud, que da origen al algoritmo de generación de las mismas que se explica a continuación.

3.1.2. Construcción de *EPAs*

Antes de mostrar el algoritmo de generación de *EPAs* vamos a definir un predicado que nos será de utilidad.

Definición 3.1.5 (Predicado de un conjunto de acciones)**.** Dado un subconjunto A de las etiquetas de un *Action System*, definimos $pred_A : \mathbb{C} \to \{\textbf{true}, \textbf{false}\}$ como

$$pred_A(c) \overset{def}{\Leftrightarrow} inv(c) \wedge \bigwedge_{a \in A} \exists p.R_a(c,p) \wedge \bigwedge_{a \notin A} \nexists p.R_a(c,p)$$

Haciendo uso de esta definición podemos expresar la construcción de *EPAs* cómo en el algoritmo 1. Nuevamente referimos al lector a [Cas+13] para un estudio en más detalle del mismo.

Algorithm 1 Construcción de una *EPA*

Entrada Un *Action System* $AS = \langle Act, F, R, inv, init \rangle$
Salida La *EPA* $M = \langle \Sigma, S, S_0, \delta \rangle$

1: $\Sigma = Act; S = \emptyset;$
2: $\delta(A, a) = \emptyset, \forall a \in Act, A \subseteq Act;$
3: $A^- = \{a \in Act \mid \forall c \in \mathbb{C}.init(c) \Rightarrow \nexists p \in \mathbb{Z}.R_a(c, p)\};$
4: $A^+ = \{a \in Act \mid \forall c \in \mathbb{C}.init(c) \Rightarrow \exists p \in \mathbb{Z}.R_a(c, p)\};$
5: $S_0^c = \{A \subseteq Act \mid A^+ \subseteq A, A^- \cap A = \emptyset\};$
6: $S_0 = \{A \in S_0^c \mid \exists c \in \mathbb{C}.pred_A(c) \wedge init(c)\};$
7: $W =$ *cola con los elementos de* S_0;
8: **mientras** *haya un conjunto* A *en el tope de* W **hacer**
9: $\quad W - [A];$
10: $\quad S \cup \{A\};$
11: $\quad$ **por cada** $a \in A$ **hacer**
12: $\quad\quad B^- = \{b \in Act \mid \forall c \in \mathbb{C}, p \in \mathbb{Z}.pred_A(c) \wedge R_a(c, p) \Rightarrow \nexists p' \in \mathbb{Z}.R_b(F_a(c, p), p')\};$
13: $\quad\quad B^+ = \{b \in Act \mid \forall c \in \mathbb{C}, p \in \mathbb{Z}.pred_A(c) \wedge R_a(c, p) \Rightarrow \exists p' \in \mathbb{Z}.R_b(F_a(c, p), p')\};$
14: $\quad\quad S^c = \{B \subseteq Act \mid B^+ \subseteq B, B^- \cap B = \emptyset\};$
15: $\quad\quad$ **por cada** $B \in S^c$ **hacer**
16: $\quad\quad\quad$ **si** $\exists c \in \mathbb{C}, p \in \mathbb{Z}.pred_A(c) \wedge R_a(c, p) \wedge pred_B(F_a(c, p))$ **entonces**
17: $\quad\quad\quad\quad \delta(A, a) = \delta(A, a) \cup \{B\};$
18: $\quad\quad\quad\quad$ **si** $B \notin S \wedge B \notin W$ **entonces**
19: $\quad\quad\quad\quad\quad W = W \cup [B];$
20: $\quad\quad\quad\quad$ **fin si**
21: $\quad\quad\quad$ **fin si**
22: $\quad\quad$ **fin por cada**
23: $\quad$ **fin por cada**
24: **fin mientras**

Es importante notar que este algoritmo se basa en la demostración de predicados de primer orden, en particular en las lineas 3, 4, 12, 13 y 16. Sin embargo no plantea cómo demostrar las mismas, ni que hacer cuando esto no es posible, lo cual queda a elección del implementador.

Este trabajo se basa en [LG16], y utiliza *Corral* como motor para resolver estas fórmulas.

3.2. Corral: Reachability modulo theories solver

El problema de *alcanzabilidad*, o *Reachability* en inglés, tiene origen en el estudio de máquinas de estado finito, y consiste en responder la siguiente pregunta: dado un grafo, un nodo *origen* y uno *destino*, existe un camino de *origen* a *destino*? Debido a que una gran clase de programas puede representarse con esta abstracción, ha atraído la atención de los investigadores dedicados a la verificación de software.

Con los años el modelo original ha sido extendido para aumentar la expresividad del mismo con el objetivo de analizar clases más amplias de programas. Una de estas extensio-

nes es la hecha [LQL12], la cual introduce un lenguaje de representación intermedia para la verificación. Este tiene el objetivo de disminuir la brecha semántica entre los lenguajes de programación utilizados en la industria, como *C*, *C#* y *Java*, y el modelo utilizado para resolver efectivamente el problema de la alcanzabilidad. Se llama *Reachability Modulo Theories* a la versión de este problema para los modelos que ésta extensión permite.

Corral es un proyecto de *Microsoft Research* que resuelve *queries* de *reachability* en modelos escritos en *Boogie* [Lei08], un lenguaje de representación intermedio orientado a la verificación. Para utilizarlo se traduce un programa a este lenguaje, y luego se plantea la alcanzabilidad como una función en *Boogie* con uno o más *asserts*. Este motor buscará un contraejemplo que viole estas aserciones.

Como se muestra en la figura 3.1 *Corral* hace uso del *solver* de *Satisfability Modulo Theories Z3*, convirtiéndolos de *Reachability*, o *queries*, a *Verification Conditions*, o *VC*s, que este resuelve.

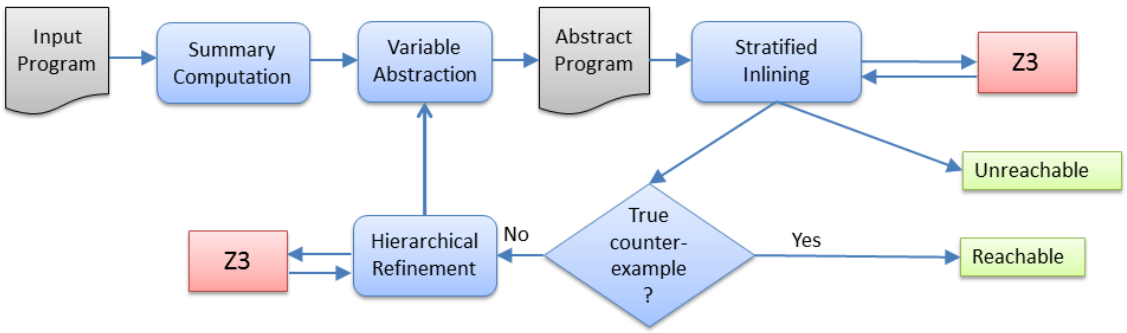

Fig. 3.1: Arquitectura de Corral

La ejecución de *Corral* se centra en un ciclo de dos niveles de refinamiento de soluciones guiado por contraejemplos, llamado *CEGAR loop* por las siglas en inglés de $counterexample - guidedabstractionrefinement$.

El primer nivel del ciclo se encarga de abstraer al solver de las variables globales del programa. En la práctica no suele ser necesario tenerlas en cuenta para verificar una parte del mismo, e incluir las mismas es muy costoso en la *VC*. Por lo tanto inicialmente se ignoran todas las variables globales y se pasa al siguiente nivel.

Este nivel, llamado *Stratified Inlining* es el encargado de los llamados a funciones en el programa a analizar. Como hacer *inlining* estático de todos los llamados es muy costoso, este módulo utiliza *sub-* y *sobreaproximaciones* de los mismos para decidir donde hacer o no *inlining* en cada iteración. Si se decide no hacer *inlining* se utiliza la construcción *assume false* de *Boogie* que recorta toda posible traza de modo tal que solo se analice hasta ese punto. En caso contrario se utiliza una abstracción de de la función.

Haciendo uso de estos dos niveles, la resolución de una *query* P comienza con esta y un conjunto de todos los *call-sites* de la misma, C. En todo momento, se mantiene un programa P parcialmente *inlined*, junto con un conjunto de call-sites C en P que todavía

no han sido *inlined*.

Cada iteración se compone de dos etapas. En la primera, cada *call-site* abierto en P es reemplazado por su sub-aproximación para así obtener un programa cerrado P', que es chequeado usando *Z3*. Lo que se intenta en este caso es determinar la satisfacibilidad de las aserciones en el programa. Si es posible alguna, el algoritmo concluye que hay un *bug*.

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

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

Es importante aclarar que en una ejecución de *Corral* el usuario acota el espacio de búsqueda limitando la cantidad de veces que se hace *inlining* en un llamado recursivo. Esto lleva a que la respuesta arrojada al intentar resolver una *query* sea *True Bug*, *No Bug*, o *Maybe Bug* cuando se alcanza la cota.

Una implementación del algoritmo 1 debe tener en cuenta la posibilidad de que *Corral* retorne *Maybe Bug* para ser correcta. En los casos en que una *query* se esté usando para acotar un conjunto, no incluiremos al elemento analizado en la cota si obtenemos un *Maybe Bug* ya que no sería seguro hacerlo. Cuando se esta analizando si una transición de estados se encuentra en la *EPA* y obtenemos este resultado sí la agregaremos, pues una *EPA* es una sobreaproximación del comportamiento real del módulo, y no sería seguro no incluirla, pues podrían quedar afuera de la *EPA* transiciones factibles.

4. PERMISSIVE ENABLEDNESS-PRESERVING ABSTRACTIONS

En el capítulo 2 se mostraron dos limitaciones de las *EPAs*. La primera es que no contemplan la posibilidad de que se rompa el invariante del módulo que abstraen. La segunda es que carecen de información alguna sobre la salida excepcional o errónea de un método.

Este modelo está basado en el diseño por contratos [Mey88], el cual requiere que el programador exprese de manera formal, precisa y verificable las interfaces de los componentes de software. Bajo estas premisas una invocación de un método que cumpla su contrato no podría romper el invariante. Sin embargo, expresar las precondiciones suele ser difícil en la practica. El programador realiza un esfuerzo por codificar las precondiciones, pero no siempre lo logra. Llamamos al resultado de esto *intended preconditions*, para distinguirlo de las precondiciones reales.

Con la idea de que incluir está discrepancia entre contratos e implementación hará de nuestros modelos más útiles en la práctica, extendemos la definición de los mismos, dando lugar a las *Permissive Enabledness-Preserving Abtractions*.

En esta sección se explican las dos extensiones que se realizan al modelo formal de *EPAs*, y se revisa el algoritmo de construcción de las mismas.

4.1. Modelo formal

El primer paso para hacer más flexible nuestro modelo formal es incluir las configuraciones donde no es válido el invariante del módulo. En particular, la semántica de un *Action System* debe revisarse de modo tal que no todos sus estados cumplan el invariante.

Es necesario detenerse a analizar que ocurre durante la ejecución de un programa cuando una pieza del mismo llega a un estado donde no se cumple su invariante. Consideramos que es de interés del implementador detectar este defecto y corregirlo tempranamente. Encontrar transiciones que partan desde configuraciones donde no se cumple el invariante excede el alcance de este trabajo.

Se tomó entonces la decisión de incluir un único estado **ERROR** que no cumple con el invariante, del cual no parte ninguna transición, y sólo es de interés detectar su presencia y qué caminos llevan a él.

La segunda extensión permite relajar el concepto de precondición. Si bien el programador debería intentar expresar los contratos de manera correcta, no siempre será el caso. Puede existir una diferencia entre lo expresado, la *intended precondition*, y la precondición real de la implementación, lo cual lleva a que una invocación de un método pueda terminar en un fallo.

Esto se ve en nuestro modelo como una extensión a las funciones que componen un

Action System. En el caso de las *EPAs* las mismas retornan una nueva configuración, o $\perp$ en caso de que el cómputo no termine. En nuestro modelo de *PEPAs* se retorna $\perp$, o una tupla donde el primer elemento es la nueva configuración y el segundo es un valor booleano que indica si se produjo un error o no.

La elección de esta forma de modelar los errores es arbitraria, y se ve influenciada por limitaciones a la hora de implementar lo expuesto en esta sección. Se explican las mismas en el capítulo 5. En particular, los lenguajes de programación permiten retornar distintos tipos de errores al ejecutar un mismo método, lo cual escapa nuestra formalización. Sin embargo, se verá más adelante que indicar la presencia de errores, sin discriminarlos por tipo, a pesar de no ser lo ideal, es de utilidad para un usuario.

La siguiente formalización es una extensión natural de lo visto en la sección 3.1, con los cambios mencionados. Definimos un *Permissive Action System*, le damos su semántica utilizando un *LTS*, y luego se definen las *Permissive Enabledness-Preserving Abtractions* en función de ésta.

Definición 4.1.1 (Permissive Action System)**.** Un *Permissive Action System* es una estructura $PAS = \langle Act, F, R, inv, init \rangle$ donde todo coincide con un *Action System*, a excepción de F.

En un *Permissive Action System* F es también un conjunto de funciones indexado por las etiquetas del sistema. Pero $F_a : \mathbb{C} \times \mathbb{Z} \to ((\mathbb{C} \times \{\textbf{true}, \textbf{false}\}) \cup \{\perp\})$ indica si se produjo o no un error en caso de que el cómputo converja.

De manera similar, la semántica de un *Permissive Action System* es simplemente una extensión a la de un *Action System*, con información sobre si se produjo o no un error en las transiciones, y un estado **ERROR** para representar la ruptura del invariante.

Definición 4.1.2 (Semantica de un *Permissive Action System*)**.** Dado un *Permissive Action System* $PAS = \langle Act, F, R, inv, init \rangle$ su semántica está provista por un *LTS* $L = \langle \mathbb{A}, \mathbb{S}, \mathbb{S}_0, \Delta \rangle$, que satisface:

- $\mathbb{A} = Act \times \mathbb{Z} \times \{\textbf{true}, \textbf{false}\}$
- $\mathbb{S} = \{c \in \mathbb{C} \mid inv(c) = \textbf{true}\} \cup \{\textbf{ERROR}\}$
- $\mathbb{S}_0 = \{c \in \mathbb{S} \mid c \neq \textbf{ERROR} \wedge init(c) = \textbf{true}\}$
- $\forall c \in \mathbb{S}, a \in Act, p \in \mathbb{Z}. c \neq \textbf{ERROR} \wedge R_a(c, p) = \textbf{true} \wedge F_a(c, p) = (c', err) \Rightarrow (inv(c') \wedge \Delta(c, (a, p, err)) = c') \vee (\neg inv(c') \wedge \Delta(c, (a, p, err)) = \textbf{ERROR})$.
- Δ no está definido para cualquier otro caso.

Al igual que en el capítulo 3, esta máquina de estados es infinita, por lo que la abstraemos en función a las clases de equivalencia de la relación definida a continuación.

Definición 4.1.3 (Permissive-Enabledness-equivalencia)**.** Dado un *Permissive Action System PAS*, su semántica, y c_1, c_2 estados de esta, decimos que son *Permissive-Enabledness-equivalentes*, y lo notamos $c_1 \equiv_{pe} c_2$, sii ($c_1 = c_2 = \mathbf{ERROR}$) ó

$$c_1 \neq \mathbf{ERROR} \wedge c_2 \neq \mathbf{ERROR} \wedge \forall a.(\exists p.R_a(c_1, p) = \mathbf{true} \iff \exists p'.R_a(c_2, p') = \mathbf{true})$$

Notar que a diferencia de $\equiv_e$, la nueva relación es sobre los estados del *LTS* que da semántica al *Permissive Action System*, y no sobre las configuraciones. Esto se debe a que su objetivo es particionar los estados de la interpretación semántica. Mientras que los estados de la interpretación de un *Action System* son un subconjunto de las configuraciones, en ésta versión incluyen también a **ERROR**, por lo que es necesario corregir la defínción.

Finalmente, extendemos también la definición de *Enabledness-Preserving Abstractions*.

Definición 4.1.4 (Permissive Enabledness-Preserving Abstraction)**.** Dado un *Permissive Action System* $PAS = \langle Act, F, R, inv, init \rangle$ y su *LTS* $L = \langle \mathbb{A}, \mathbb{S}, \mathbb{S}_0, \Delta \rangle$, decimos que $M = \langle \Sigma, S, S_0, \delta \rangle$ es una *Permissive Enabledness-Preserving Abstraction* o *PEPA* de PAS sii existe una función total $\alpha : \mathbb{S} \to S$ tal que

- $\alpha(\mathbb{S}_0) \subseteq S_0$
- $\forall s \in \mathbb{S} \;.\; \alpha(s) = \mathbf{ERROR} \Leftrightarrow s = \mathbf{ERROR}$
- $\forall c \in \mathbb{S}, a \in Act, p \in \mathbb{Z} \;.\; c \neq \mathbf{ERROR} \wedge R_a(c, p) = \mathbf{true} \wedge F_a(c, p) = (c', err) \Rightarrow \alpha(\Delta(c, (a, p, err))) \in \delta(\alpha(c), (a, err))$
- $\forall c_1, c_2 \in \mathbb{S} \;.\; c_1 \equiv_{pe} c_2 \iff \alpha(c_1) = \alpha(c_2)$

Se hace abuso de notación de α utilizándola también para su extensión a conjuntos.

4.2. Construcción de *PEPAs*

Presentamos en esta sección una revisión del algoritmo 1 para la construcción de *Permissive Enabledness-Preserving Abtractions*. El mismo es una extensión natural de su versión de *EPAs*.

Se utiliza en el algoritmo 2 un predicado equivalente al de la sección 3.1.2, cuya única diferencia es que las acciones pertenecen ahora a un *Permissive Action System*. Haciendo uso de este, se define el algoritmo para generar una *PEPA*.

Algorithm 2 Construcción de una *PEPA*

Entrada Un *Permissive Action System* $PAS = \langle Act, F, R, inv, init \rangle$
Salida La *PEPA* $M = \langle \Sigma, S, S_0, \delta \rangle$

1: $\Sigma = Act \times \{\textbf{true}, \textbf{false}\}; S = \emptyset;$
2: $\delta(A, (a, err)) = \emptyset, \forall a \in Act, A \subseteq Act, err \in \{\textbf{true}, \textbf{false}\};$
3: $\delta(\textbf{ERROR}, (a, err)) = \emptyset, \forall a \in Act, err \in \{\textbf{true}, \textbf{false}\};$
4: $A^- = \{a \in Act \mid \forall c \in \mathbb{C}.inv(c) \wedge init(c) \Rightarrow \nexists p \in \mathbb{Z}.R_a(c,p)\}$;
5: $A^+ = \{a \in Act \mid \forall c \in \mathbb{C}.inv(c) \wedge init(c) \Rightarrow \exists p \in \mathbb{Z}.R_a(c,p)\}$;
6: $S_0^c = \{A \subseteq Act \mid A^+ \subseteq A, A^- \cap A = \emptyset\}$;
7: $S_0 = \{A \in S_0^c \mid \exists c \in \mathbb{C}.pred_A(c) \wedge init(c)\}$;
8: $W =$ *cola con los elementos de* S_0;
9: **mientras** *haya un conjunto* A *en el tope de* W **hacer**
10: $W - [A]$;
11: $S \cup \{A\}$;
12: **por cada** $a \in A$ **hacer**
13: **si** $\exists c \in \mathbb{C}, p \in \mathbb{Z}.pred_A(c) \wedge R_a(c,p) \wedge F_a(c,p) = (c', \textbf{true}) \wedge \neg inv(c')$ **entonces**
14: $\delta(A, (a, \textbf{true})) = \delta(A, (a, \textbf{true})) \cup \textbf{ERROR}$
15: **fin si**
16: **si** $\exists c \in \mathbb{C}, p \in \mathbb{Z}.pred_A(c) \wedge R_a(c,p) \wedge F_a(c,p) = (c', \textbf{false}) \wedge \neg inv(c')$ **entonces**
17: $\delta(A, (a, \textbf{false})) = \delta(A, (a, \textbf{false})) \cup \textbf{ERROR}$
18: **fin si**
19: $B^- = \{b \in Act \mid \forall c, p.pred_A(c) \wedge R_a(c,p) \wedge F_a(c,p) = (c', err) \Rightarrow \nexists p'.R_b(c', p')\}$;
20: $B^+ = \{b \in Act \mid \forall c, p.pred_A(c) \wedge R_a(c,p) \wedge F_a(c,p) = (c', err) \Rightarrow \exists p'.R_b(c', p')\}$;
21: $S^c = \{B \subseteq Act \mid B^+ \subseteq B, B^- \cap B = \emptyset\}$;
22: **por cada** $B \in S^c$ **hacer**
23: **si** $\exists c, p.pred_A(c) \wedge R_a(c,p) \wedge F_a(c,p) = (c', \textbf{true}) \wedge pred_B(c')$ **entonces**
24: $\delta(A, (a, \textbf{true})) = \delta(A, (a, \textbf{true})) \cup \{B\}$;
25: **si** $B \notin S \wedge B \notin W$ **entonces**
26: $W = W \cup [B]$;
27: **fin si**
28: **fin si**
29: **si** $\exists c, p.pred_A(c) \wedge R_a(c,p) \wedge F_a(c,p) = (c', \textbf{false}) \wedge pred_B(c')$ **entonces**
30: $\delta(A, (a, \textbf{false})) = \delta(A, (a, \textbf{false})) \cup \{B\}$;
31: **si** $B \notin S \wedge B \notin W$ **entonces**
32: $W = W \cup [B]$;
33: **fin si**
34: **fin si**
35: **fin por cada**
36: **fin por cada**
37: **fin mientras**

Notar que B^+ y B^- son utilizados como cotas para los posibles conjuntos destino al ejecutar A tanto cuando ocurre como cuando no ocurre un error. Estas cotas podrían separarse para cada caso, lo cual podría mejorar el tiempo de corrida del algoritmo. Sin embargo, este trabajo no se enfoca en estos aspectos, y se optó por la versión más sencilla.

El algoritmo 2 genera una *PEPA* a partir de un *Permissive Action System*, y tiene una complejidad temporal exponencial en el número de acciones.

No se presentan demostraciones de correctitud ni de la complejidad del mismo, pero la primera de ellas puede basarse en la demostración del algoritmo de construcción de *EPAs* que se presentan en [Cas+13]. Notar que salvo el estado distinguido **ERROR**, los estados de un *Action System* y un *Permissive Action System* coinciden, por lo que sólo debemos analizar a éste de forma distinta. A su vez, no hay transiciones salientes de éste, lo que simplifica esta tarea. Luego, la otra diferencia entre *EPA* y *PEPA* es que por cada método hay dos posibles transiciones. Si se codifica a esto como dos acciones distintas, podría reutilizarse el resto de las ideas de la demostración del algoritmo *EPAs*. A su vez, se utiliza el mismo criterio cuando una query.

5. IMPLEMENTACIÓN

En esta sección se describe la implementación de *ContractorJ*, un generador de *PEPAs* para *Java* basado en el algoritmo antes visto.

ContractorJ está inspirado en la última versión de *Contractor.Net* [LG16] en el uso de *Corral* como model checker.

5.1. *ContractorJ*: arquitectura

El algoritmo 2 no impone un método de resolución para sus fórmulas de primer orden. *ContractorJ* utiliza *Corral*, reduciendo dichas fórmulas a problemas o *queries* de *Reachability*. Se explica con mayor detalle cómo se realiza esto más adelante, presentando ahora una visión general de la herramienta.

En la figura 5.1 se puede observar un diagrama de la arquitectura de *ContractorJ*. El mismo podría ser dividido en tres partes.

Para poder utilizar *Corral* debemos contar con un modelo de nuestro módulo expresado en *Boogie*, el lenguaje de verificación intermedio utilizado por esta herramienta. Por lo tanto, la primer tarea que se realiza *ContractorJ* es una única traducción de *bytecode* de *Java* a *Boogie*, sobre la cual se tiene el control total, como se explicará en la próxima sección.

Luego, está el *core* de la generación de *PEPAs*, el cual explora el espacio de potenciales estados, descubriendo los mismos y sus transiciones mediante la resolución de distintas *queries*.

Por último, se encuentra la ejecución de *Corral*, el cual se utiliza como un proceso independiente y resuelve las consultas del *core* de la herramienta.

Esta arquitectura se diferencia de la de *Contractor.Net* en que este realiza una serie de traducciones de *C#* a *Boogie*. Esto se debe que al no tener control sobre la traducción, las queries se realizan en el lenguaje de origen. Si bien este trabajo no pone el foco en los aspectos de performance, es interesante destacar que si bien *ContractorJ* genera modelos más complejos, y debe resolver un número superior de *queries*, el tiempo de ejecución del mismo es consistentemente inferior respecto al de *Contractor.Net*. Se cree que esto se debe a esta diferencia, ya que en la versión de *Java* generar una *query* se limita a manipulación de texto, mientras que en la de *.Net* se ejecuta un compilador y un traductor.

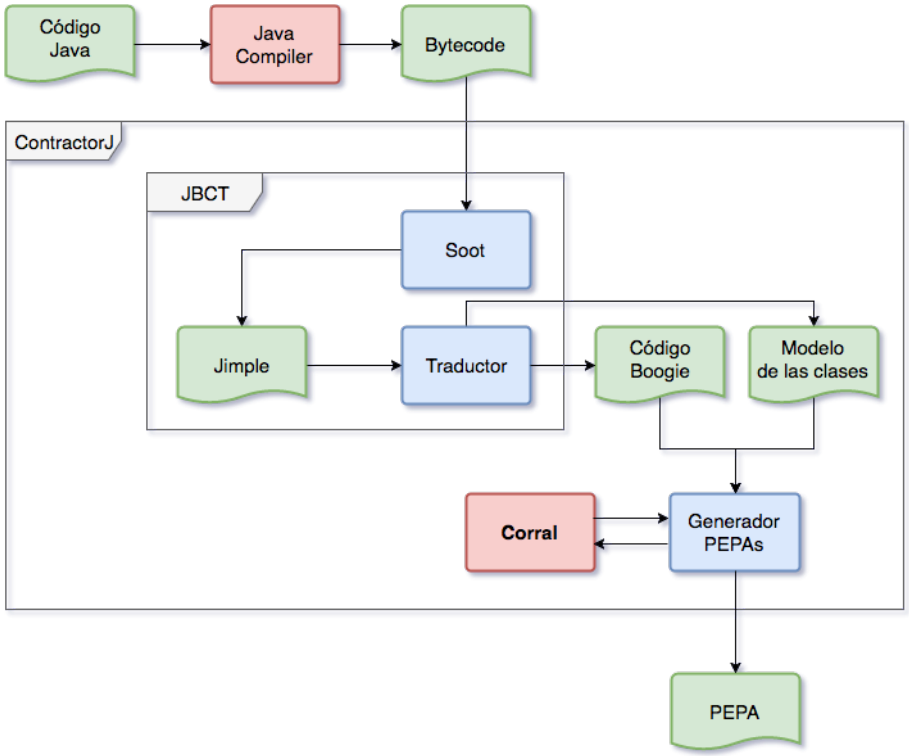

Fig. 5.1: Arquitectura de *ContractorJ*

5.2. JBCT: traducción de bytecode a Boogie2

Como se mencionó en la sección anterior, el primer paso en la ejecución de *ContractorJ* es la traducción del *bytecode* a *Boogie*. Se realizó una búsqueda exhaustiva de herramientas capaces de realizar esto, y se encontraron dos alternativas.

La primera de ellas es *Jar2Bpl*, parte del proyecto *Bixie* [MRS15], antes conocido como *Joogie* [ARS13]. Esta herramienta está desarrollada de forma que pueda ser utilizada como una librería por otros proyectos. Sin embargo, el repositorio oficial de la misma [Sch15] aclara de forma explícita que la traducción que genera no es compatible con *Corral*. El mismo cuenta también con información de por qué esto es así. Se analizó la posibilidad de modificar la herramienta de manera tal que se solucione esta limitación, pero fue descartada ya que la información disponible es poco clara, y utiliza diversas funcionalidades no documentadas del lenguaje *Boogie*, lo cual implicaba un riesgo para este trabajo.

La otra herramienta capaz de realizar la traducción encontrada es *VerCors* [BH14] de la universidad de Twente. El mismo utiliza *Boogie* como uno de los posibles motores internos para verificar propiedades sobre estructuras de datos concurrentes. Debido a la arquitectura del proyecto, no se encontró una forma sencilla de extraer el módulo de traducción para ser reutilizado, por lo que se descartó esta opción.

Se tomó la decisión de hacer un traductor propio, llamado *Java Bytecode Translator* o *JBCT*. Este está basado en *Soot* [Val+99] al igual que *Jar2Bpl*, el cual se utilizó a modo de ejemplo e inspiración.

Soot nació como un *framework* para estudiar posibles optimizaciones de programas *Java*, pero cuyo alcance ha crecido con el tiempo. Hoy cuenta con un conjunto de herramientas para analizar, instrumentar, optimizar y visualizar software para esta plataforma. Para la traducción se hizo uso de *Jimple* [VH98]. Esta herramienta procesa el *bytecode* de *Java*, transformándolo de un lenguaje *stack-based* sin tipos con un amplio *set* de instrucciones, a uno sin *stack*, tipado y con menos instrucciones. A su vez, brinda utilidades para trabajar con esta representación más sencilla del programa, de forma que implementar un traductor no sea demasiado arduo.

Sin embargo, siendo que el traductor es simplemente una herramienta para los objetivos de este trabajo, se plantearon los siguientes cinco criterios a la hora de encarar su desarrollo:

- Hacer el traductor más simple posible, prefiriendo velocidad de desarrollo sobre otros atributos de calidad de software.
- Traducir solamente el subconjunto de *Jimple* que resulte de interés para los casos de estudio de este trabajo. Sin embargo, el traductor debe producir un error explícito y claro en caso de encontrarse con código que no es capaz de traducir. De este modo *JBCT* se pudo desarrollar de manera incremental, agregando nuevas funcionalidades cuando fueron requeridas.
- Se ignora todo aspecto de concurrencia y *multithreading* de *Java*. Las queries que se quieren resolver son *single-threaded*, por lo que este enfoque funciona.
- El modelo del programa que se genera debe ser muy similar al que genera el traductor oficial de *C#* a *Boogie*, llamado *BCT*[Qad]. De este modo se puede reutilizar el conocimiento de otros miembros del equipo de investigación, que trabajan en proyectos relacionados a *EPAs* en *.Net*, y utilizan dicho traductor.
- No se modela el sistema de tipos, por lo que no hay soporte de subtipado ni invocaciones virtuales. Esto no es un problema mayor para los ejemplos que nos interesan, ya que en general pueden aplanarse las jerarquías de clases de los programas que analizamos, manteniendo el comportamiento de los mismos. Sin embargo, como se mencionó en la sección 4.1, esto limita la expresividad sobre los errores que puede arrojar la ejecución de un método, ya que hace indistinguibles sus tipos.

Con estos criterios en mente, se implementó una versión del traductor lo suficientemente completa como para realizar pruebas de interés. Puede encontrarse en el código de fuente adjunto a este trabajo.

5.3. Contratos en Java

Una diferencia importante entre *Java* y *C#*, es que en el primero no existe soporte nativo de programación por contratos. Mientras que *Contractor.Net* utilizan la librería de *Code Contracts*, que permite expresar precondiciones en cada método, *ContractorJ* utiliza

su propia convención para suplir la falta de un equivalente.

En este trabajo se reutilizó lo planteado en la versión original de *Contractor*[Cas+13], de manera que las precondiciones de un método están implementadas también como procedimientos de la misma clase. Dado un método m, *ContractorJ* considera que sus precondiciones están expresadas como dos métodos con nombre m_pre que retornan un valor booleano. Uno de estos no recibirá parámetros, y predicará sobre el estado interno del objeto. El otro tendrá los mismos parámetros que m, y predicará sobre estos.

El motivo esta separación en dos métodos está relacionado a que las fórmulas de primer orden del algoritmo 2 se traducen a problemas de *reachability*, los cuales sólo pueden verificar *safety properties*. Algunas de las fórmulas que deben ser resueltas no pueden expresarse de esta forma. Por ejemplo en la linea 4 del algoritmo se debe resolver $\forall c \in \mathbb{C}.inv(c) \wedge init(c) \Rightarrow \nexists p \in \mathbb{Z}.R_a(c,p)$, y una prueba de la validez de la misma tendría la forma de una función que retorna un p válido para cada configuración c, lo cual no puede traducirse a un problema de *Reachability*.

En [Cas13] se realiza un análisis en profundidad de este problema. En éste trabajo optamos por sacrificar parcialmente la expresividad de las precondiciones, exigiendo que sean separadas. Dado que la precondición que predica sobre los parámetros no puede acceder al estado interno, suponemos que el usuario del módulo analizado siempre podrá elegir un p que la cumpla. Por lo tanto, utilizamos las mismas únicamente para asegurarnos que estamos llamando de manera correcta a un método, pero no para el cómputo de cúales se encuentran habilitados en cada estado, evitando el problema.

A modo de ejemplo, mostramos a continuación una posible implementación de las precondiciones de el método **cargar(int)** de la figura 2.6. Notar que la primer precondición corrobora que haya una caja abierta disponible, mientras la otra especifica que el peso máximo por carga es de 1000g.

```
public boolean cargar_pre() {
  return hayCaja && !cajaCerrada;
}

public boolean cargar(int peso) {
    return peso <= 1000;
}
```

Fig. 5.2: Precondiciones del método *cargar*

5.4. Queries en *ContractorJ*

Para implementar el algoritmo de generación de *PEPAs* es necesario resolver una serie de fórmulas booleanas de primer orden. En esta sección se describe como las mismas son reducidas a *queries* de *reachability* para ser resueltas por *Corral*.

Podemos clasificar a las *queries* que necesitamos resolver del siguiente modo:

1. **Queries para calcular el estado inicial:** se encuentran fuera del ciclo principal del algoritmo, y sirven para computar cúales son los primeros métodos en estar habilitados. *ContractorJ* se limita a analizar clases de *Java*, y asume que el estado inicial posee únicamente los constructores de la clase bajo análisis, de modo que no se implementan estas *queries*.

2. **Queries de violación de invariante:** son fórmulas que intentan encontrar una posible violación de un invariante a través de una invocación a un método.

   2.1. Con salida excepcional del método (linea 13).

   2.2. Con salida exitosa del método (linea 16).

3. **Queries de transición:** fórmulas que demuestran la existencia de una transición entre dos estados al llamar a un método.

   3.1. Con salida excepcional del método (linea 23).

   3.2. Con salida exitosa del método (linea 29).

4. **Cotas de un posible estado destino:** El algoritmo de construcción es de naturaleza exploratoria, computando los estados de la *PEPA* a medida que se intentan encontrar transiciones hacia ellos. Como la cantidad de estados es exponencial, se usan estas dos queries para acotar la cantidad de estados destino posibles de un llamado a un método.

   4.1. Acción necesariamente habilitada (linea 19).

   4.2. Acción necesariamente deshabilitada (linea 20).

ContractorJ realiza una traducción de la clase a analizar y genera *queries* en lenguaje *Boogie* como se explica a continuación.

5.4.1. Manejo de excepciones

Para generar una *PEPA* es necesario ser capaz de construir *queries* que distingan cuando la invocación a un método arroja o no una excepción.

En este trabajo se preserva la manera de representar las excepciones que se utiliza en *Bytecode Translator* para *.Net*. La misma consiste en tener una variable global $\$Exception$, de manera que el arrojar una excepción sea simplemente asignar una nueva referencia a esta y retornar prematuramente. A su vez, inmediatamente después de un *call-site* debe compararse si $\$Excepcion$ es *null*, y en caso de no serlo retornar, de modo tal que se simule el *unwinding* del *stack*.

Una de las ventajas de este modo de implementar las excepciones es la sencillez con la cual se puede armar una *query* que predique sobre si se lanzó o no una excepción, ya que es simplemente asumir un valor de $\$Exception$.

Si queremos que en una *query* el método m no lance una excepción basta con insertar $assume\ \$Exception = null$; inmediatamente después de su *call-site*, lo cual hará que *Corral* descarte las posibles trazas donde esto no valga. De manera análoga,
$assume\ \$Exception != null$; se usa para suponer una excepción.

Una limitación que es necesaria destacar, es que debido a que *JBCT* no preserva información de tipado alguna, no es posible implementar *handlers* de excepciones. Por lo que en el código traducido las excepciones siempre terminan con la ejecución del programa, sin ejecutar el cuerpo de ningún *catch* presente en el módulo.

A pesar de esto, *ContractorJ* puede analizar un gran número de casos de estudio. Esto se debe a que en la práctica existen patrones de manejo de excepciones que simplemente se limitan a encapsular el error capturado o imprimir un mensaje de error, sin afectar la lógica del programa. En [NHT16] se analiza el uso de los mismos en el manejo de excepciones chequeadas.

5.4.2. Construcción de las queries

Veremos ahora cómo se construye cada *query*. Nos limitamos a mostrar a los casos donde no se lanza una excepción, ya que el caso análogo es muy similar.

Debido a que las precondiciones y el invariante en *ContractorJ* se implementan como métodos de la propia clase, una *query* está construida por llamados a métodos ya traducidos, asunciones y aserciones sobre los valores que estos retornan.

En caso de utilizar una asunción, *Corral* limitará su espacio de búsqueda a los casos donde la misma es válida, por lo que éstas pueden utilizarse para concentrarnos en casos donde la *query* comienza en un estado en particular, o cuando se lanza o no una excepción.

Definimos en *ContractorJ* dos tipos posibles de estado. Por un lado tenemos el estado inicial, compuesto únicamente por constructores, que se encuentran todos habilitados. Por el otro, se tiene a un subconjunto de los métodos de la clase habilitados y al resto deshabilitados.

El algoritmo 2 utiliza el predicado $pred_A$ para corroborar si un estado es válido para una configuración. En nuestra implementación alteramos levemente esta noción con el fin de tratar a los constructores como posibles transiciones de estados.

Definimos la guarda de un estado como una extensión de $pred_A$. En un estado constituido por constructores será **true**, ya que suponemos que siempre puede llamárselos. A su vez, no incluimos el invariante, ya que no hay hasta el momento un objeto que mantenga estado.

En el resto de los casos, se realiza una serie de llamados a las precondiciones de estado de todos los métodos de la clase y al invariante. Luego, la guarda se define como una fórmula booleana construida de manera que refleje $pred_A$ utilizando el valor de retorno de

cada llamado en lugar de una configuración.

Contando con esta noción, construimos las queries de cada tipo como un procedimiento en lenguaje *Boogie* con una o más aserciones. Veamos cómo son las mismas:

Queries de violación de invariante: Este tipo de *queries* es el más sencillo. El algoritmo 3 muestra un ejemplo de la misma en pseudocódigo.

Algorithm 3 Query de violación de invariante

Entrada s : Estado, this : Referencia, p : Parametro, m : Método

```
1: calls de precondiciones de estado, e inv si s no es un estado de constructores
2: assume guarda(s)
3:
4: call preParams := m_pre_params(p);
5: call m(this, p);
6: assume Exception = null;                    ▷ != cuando se lanza excepción
7:
8: call inv := inv(this);
9: assert inv;
```

Las primeras dos líneas representan una serie de llamados a función y la asunción de la guarda del estado y recortan el espacio de búsqueda a las configuraciones donde vale $pred_s$.

Luego se asegura que los parámetros recibidos por la query sean también parámetros válidos para m, para luego llamar a este. Notar que en el caso de un constructor, la traducción es también un método que inicializa un objeto, y no la alocación del mismo.

En este ejemplo sólo mostramos el caso donde la invocación no lanza una excepción, por lo que asumimos que la variable $\$Exception$ es *null*. Se cuenta también con una query complementaria donde sólo cambia esa linea.

Por último, volvemos a llamar al método del invariante, e incluimos una aserción sobre el retorno del mismo. En caso de que *Corral* logre encontrar un contraejemplo de la misma, retornará *True Bug* y agregaremos una transición del estado s al estado **ERROR**. En caso de que se retorne *Maybe Bug* la transición podría también estar presente y que simplemente se haya llegado a la cota, por lo que la incluimos, indicando que puede ser una transición espúrea.

Queries de transición Estas queries son similares a las anteriores, y también detectan la presencia de una transición en la *PEPA* cuando *Corral* encuentra un error.

Algorithm 4 Query de transición

Entrada s : Estado, t : Estado, this : Referencia, p : Parametro, m : Método

```
1: calls de precondiciones de estado, e inv si s no es un estado de constructores
2: assume guarda(s);
3:
4: call preParams := m_pre_params(p);
5: call m(this, p);
6: assume Exception = null;                    ▷ != cuando se lanza excepción
7:
8: call inv := inv(this);
9: assume inv;
10:
11: calls de precondiciones de estado
12: assert !guarda(t);
```

A diferencia del anterior, tenemos un estado origen s, y uno destino t en este caso, como se ve en el algoritmo 4. A su vez, en este caso asumimos que el invariante es válido después de llamar al método. Notar que si no lo fuera el estado destino debería ser **ERROR**, y se encontraría la transición al mismo con una *query* del tipo anterior.

Nos interesa ver que se llegó al estado t, por lo que se llama una segunda vez a los métodos de precondiciones de estado, y se realiza una aserción sobre la negación de la guarda de t. De esta forma, si *Corral* encuentra una traza donde sí se llegue al estado deseado nos retornara *True Bug* y agregamos la transición. Del mismo modo que antes, la agregamos también cuando el resultado es *Maybe Bug*.

Notar que se están generando transiciones del tipo $(m, \textbf{false})$, ya que las transiciones de una *PEPA* son tuplas de $(accion, error)$, y estamos analizando los casos donde no hubo excepción.

Cotas de un posible estado destino A diferencia de las *queries* anteriores, este tipo no se utiliza para hallar transiciones de la *PEPA*, sino para reducir el espacio de búsqueda del algoritmo, acotando el conjunto de posibles estado destino luego de llamar a un método. En el algoritmo 2 son las queries utilizadas para construir B^- y B^+.

Algorithm 5 Query usada para acotar

Entrada s : Estado, this : Referencia, p : Parametro, m : Método, m2 : Método

```
1: calls de precondiciones de estado, e inv si s no es un estado de constructores
2: assume guarda(s);
3:
4: call preParams := m_pre_params(p);
5: call m(this, p);
6: Exception := null;                ▷ Se ignora si hay o no excepción, asignando null
7:
8: call inv := inv(this);
9: assert inv;
10:
11: calls preM2 := m2_pre_estado(this);
12: assert preM2;
```

Todo estado destino va a estar compuesto por un conjunto de métodos habilitados y el resto deshabilitados. Si se tienen k métodos distintos habrá entonces 2^k posibles estados destino. Para evitar esta explosión combinatoria, se utilizan dos tipos de *queries*, las cuales comprueban si luego de llamar a un método m, otro, $m2$, está siempre habilitado o siempre deshabilitado. En el algoritmo 5 vemos un ejemplo del primer tipo.

A diferencia de los casos anteriores, este tipo de consultas intenta demostrar la ausencia de una traza que viole las aserciones, por lo que incluimos a m en B^+ cuando *Corral* retorna *No Bug*. En caso de que el resultado sea *Maybe Bug* no podemos asegurar la presencia de $m2$ en todos los casos, por lo que no lo incluimos.

Notar que podrían tenerse cotas separadas para los casos donde m lanza o no una excepción. En *ContractorJ* se opta por una implementación más sencilla, ignorando una posible salida excepcional. No es claro que separar en casos de una mejora significativa en la *performance* del algoritmo, por lo que se decidió no hacerlo.

6. RESULTADOS

En este capítulo se muestran algunos casos de estudio y los resultados obtenidos con *ContractorJ*. Se realiza primero una comparación cualitativa de los resultados obtenidos con trabajos anteriores en el área. Luego se muestra cómo se comporta la herramienta en distintos casos donde la extensión del modelo de *EPAs* que presentamos permite obtener mejores resultados.

Este trabajo no pretende hacer un análisis de *Corral* como motor de resolución de consultas en la generación de *EPAs*/*PEPAs*. Se recomienda al lector consultar [LG16] para un estudio sobre el mismo, donde se analiza su comportamiento y limitaciones.

Por último, no se realizan comparaciones de tiempo de ejecución. Esto se debe a que *ContractorJ* genera modelos distintos a las versiones para *C* y *C#* de *Contractor*, por lo que tal comparación carece de sentido. Sin embargo, vale la pena notar que durante la preparación de este documento se observó una notable diferencia entre la *performance* de *ContractorJ* y *Contractor.Net*, siendo el primero consistentemente más rápido, a pesar de realizar mayor cantidad de consultas. Creemos que esto se debe a las diferencias arquitectónicas entre estos, vistas en la sección 5.1, y a un mejor manejo de la concurrencia.

6.1. Comparación con otros trabajos

Debido a la existencia de trabajo previo [Dal+10] contra el cual poder contrastar sus resultados, las primeras publicaciones sobre *Enabledness-Preserving Abstractions* [Cas+11; Cas+13] analizan distintas clases de la librería estándar de *Java*, reimplementando las mismas en el lenguaje que estaban analizando.

En esta sección se analizan los resultados obtenidos por *ContractorJ* para los mismos artefactos de software. A diferencia de los trabajos ya mencionados, analizamos la versión original de éstas clases, salvo modificaciones menores.

Para esto, se descargó el código de fuente del proyecto *OpenJDK*, una implementación libre de la *Java Virtual Machine*, en su versión *6b27*. Luego, se extrajeron las clases a analizar, junto a toda otra clase necesaria para que éstas compilen. Se eliminó la funcionalidad que no estaba presente en los trabajos anteriores, con el fin de generar abstracciones de tamaño similar, y que puedan ser comparadas. Finalmente, se aplanó manualmente su jerarquía de tipos dejando solamente clases concretas sin herencia, de forma que puedan ser traducidas por *JBCT*.

Una vez obtenidas clases que *ContractorJ* es capaz de analizar, se agregaron contratos a la misma. Para esto se insertaron precondiciones de modo tal que invocar a los métodos no arroje excepciones mientras sea posible asegurarlo. También se consultó [Cas+13], comparando las precondiciones obtenidas con las allí mostradas, y modificándolas según corresponda.

6.1.1. ListIterator

Quizás el ejemplo más estudiado sobre *EPAs* es el de *ListIterator* de la librería estándar de *Java*. Esta clase permite recorrer una lista en su orden natural, y agregar y quitar elementos a la misma. Gracias a la gran cantidad de trabajo que se ha realizado sobre el mismo, se cuenta con una *EPA* generada de manera manual, la cual se considera correcta.

Haciendo uso de *ContractorJ* generamos la *PEPA* de la figura 6.1 y comparamos las mismas. Las transiciones con un símbolo de descarga delante de su nombre simbolizan una invocación al método mencionado que resultó en una excepción. Notar que si eliminamos las mismas, y el estado **ERROR** si lo hubiere, se obtiene una *EPA*.

Comparamos la *EPA* ideal con la que ya contábamos, con la obtenida mediante la reducción de la *PEPA* generada, y ambas coinciden.

Por último, analizamos las transiciones que arrojan excepciones. Inspeccionando la salida de *Corral* al encontrar las mismas, y contrastándolo con el código y los contratos del mismo, se concluyó que las mismas se deben a un mecanismo de protección de modificaciones concurrentes de la lista con el que cuenta el iterador.

Cabe destacar que siendo esta una excepción generada por un posible comportamiento concurrente, modificar los contratos para que la eviten puede no ser siempre efectivo. Si no se toman precauciones al respecto, en el tiempo en que un *thread* corrobora la precondición para llamar a un método m, y este es ejecutado, un segundo *thread* podría hacer que ésta deje de valer. Por lo tanto, se decidió no alterar las precondiciones de los métodos para evitar estas excepciones, dejando la *PEPA* como documentación sobre estos.

Por último, la *PEPA* también nos brinda información del estado en que queda un objeto luego de arrojar una excepción. En la figura 6.1 se observa que las transiciones con errores no cambian nunca de estado. Esto coincide con lo esperado cuando la clase está programada de manera defensiva, pero al no haber garantías al respecto, contar con la abstracción que lo aclare puede ser un gran beneficio para el usuario de la clase.

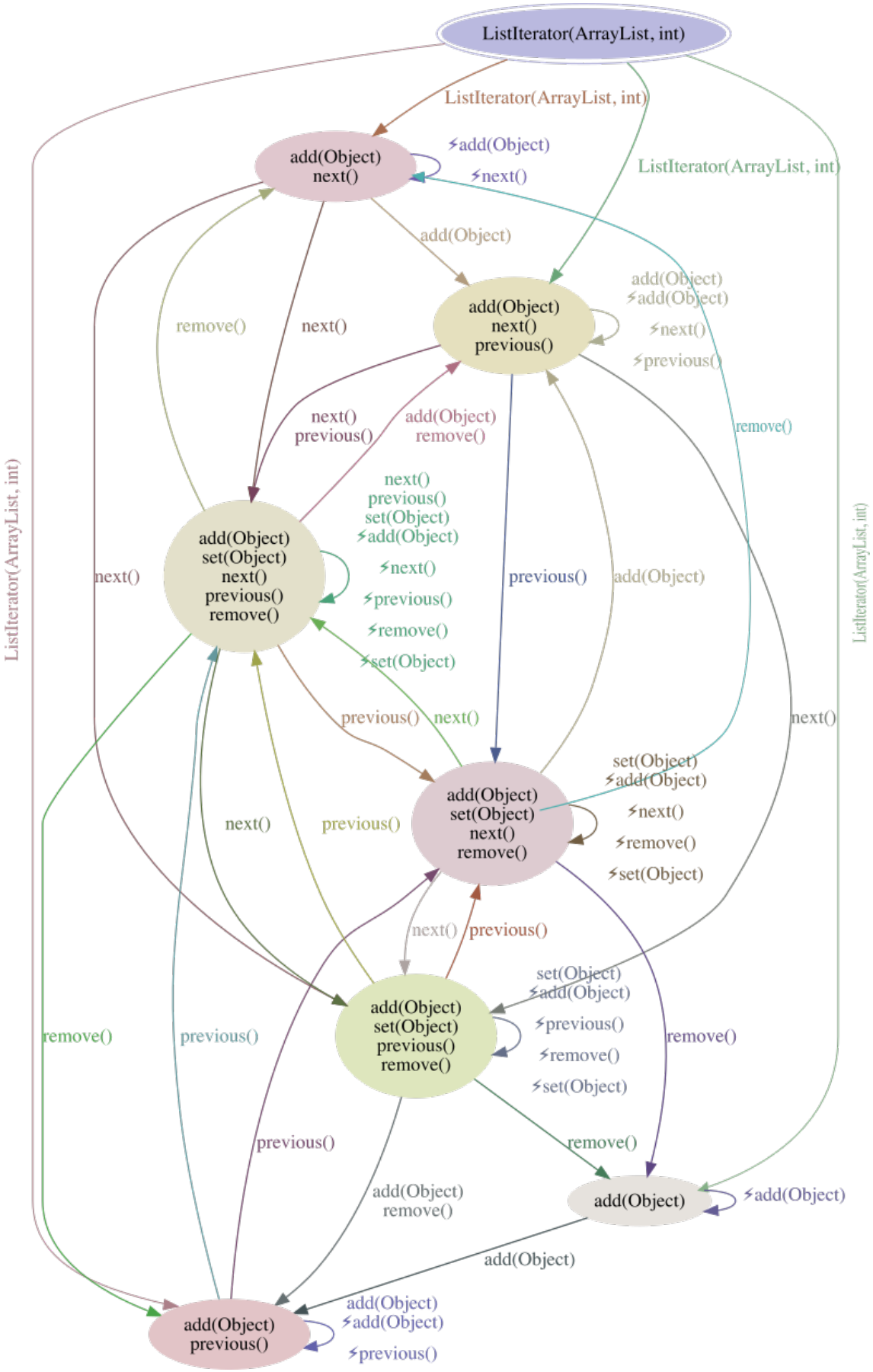

Fig. 6.1: PEPA de ListIterator

6.1.2. Signature

El segundo ejemplo que mostramos es el de la clase *Signature* de *Java*. La misma es una clase abstracta que cuenta con funcionalidad básica de firmas digitales y es extendida por distintas implementaciones de la misma.

Debido a la complejidad de los algoritmos criptográficos involucrados en las distintas subclases de *Signature* se decidió utilizar ésta y no una de sus subclases. Para ello, se remplazaron sus métodos abstractos por *stubs* de los mismos, transformándola en una clase concreta.

De igual modo que en el ejemplo anterior, se agregaron contratos que eviten que se produzcan errores al ejecutar los distintos métodos. A diferencia del iterador de la lista, esta clase no cuenta con aspectos concurrentes, por lo que los contratos si evitan toda posible excepción.

La figura 6.2 muestra los resultados obtenidos. Notar que al carecer de un estado de error y de excepciones, esta coincide con una *EPA*. En particular coincide con la *EPA* obtenida en [Cas+13] haciendo uso de la versión en *C* de *Contractor*.

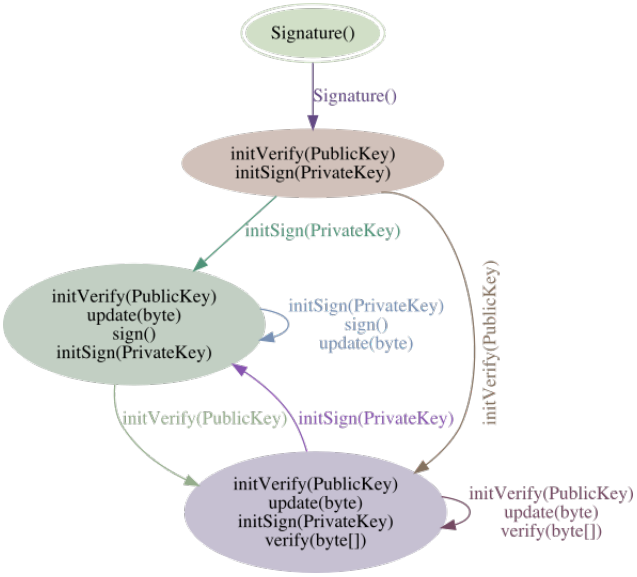

Fig. 6.2: PEPA de Signature

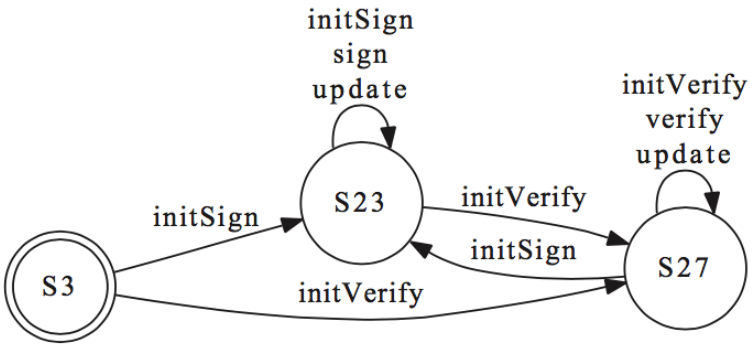

Fig. 6.3: EPA de Signature de [Cas+13]

En la figura 6.4 se muestra un modelo realizado a mano de ésta misma clase, presentado en [Dal+10]. Se puede observar que el mismo es muy similar a nuestra *PEPA*, excepto por un estado extra llamado *ex*. Este es utilizado para señalar una posible terminación excepcional al llamar a un método, lo cual no está presente en nuestro caso por cómo se elaboraron los contratos y los *stubs* de la clase. Notar que no provee información de en qué estado queda el objeto, a diferencia de las *PEPAs* que si lo hacen, por lo que es de menor utilidad para el programador.

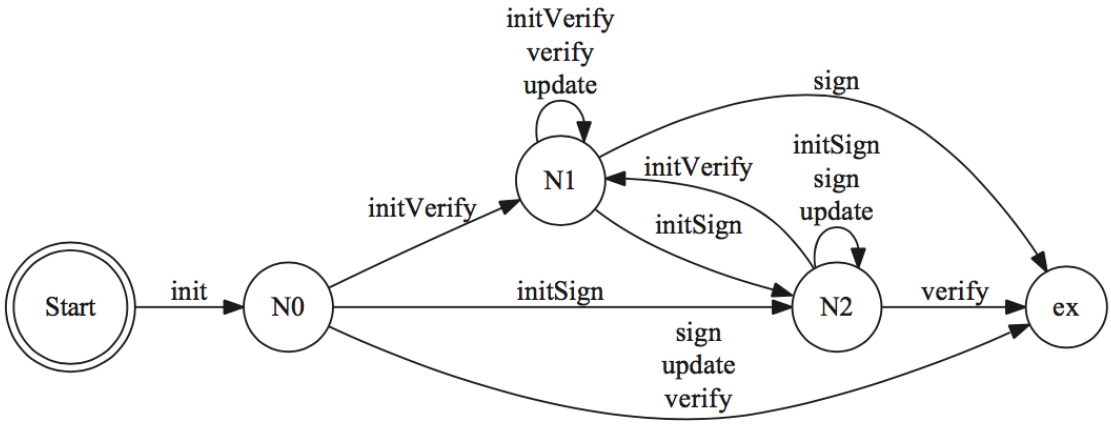

Fig. 6.4: Modelo realizado a mano de la clase Signature en [Dal+10]

6.2. Defectos detectados utilizando *PEPAs*

Como se vió en la motivación, una *EPA* a veces puede no ayudar a encontrar defectos en la implementación de una clase o sus contratos, o dar resultados poco claros. Vemos en esta sección como una *PEPA* brinda información sobre los mismos.

6.2.1. Rotura de invariante

Una de las ventajas que tienen las abstracciones presentadas en este trabajo es que contemplan las configuraciones donde el invariante es inválido. El motivo para hacer esto, es poder representar una transición a una de estas de manera explícita.

En el capítulo 2 se presentó una versión defectuosa del controlador del brazo robótico (figura 2.4) y se vio como una *EPA* puede brindar información poco clara sobre esto.

En la figura 6.5 vemos el resultado de correr *ContractorJ* para genera una *PEPA* de la misma clase. En este caso se observa un estado distinguido llamado **ERROR**, y una transición hacia el mismo, mostrando de manera explícita que se rompe el invariante. A su vez, se elimina la posibilidad que estaba presente en las *EPAs* de no mostrar de forma alguna esta situación.

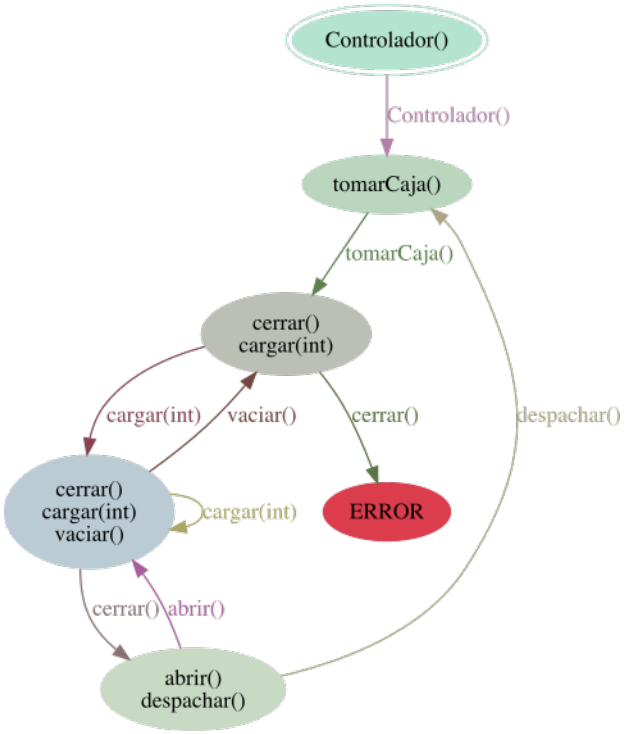

Fig. 6.5: PEPA del controlador del brazo robótico que rompe su invariante

6.2.2. Lanzamiento de excepciones

La segunda extensión que realizamos a las *EPAs* es agregar información sobre el lanzamiento de excepciones al invocar a un método. Nuevamente analizaremos usando *ContractorJ* uno de los ejemplos de la sección de motivación, en este caso el código de la figura 2.6.

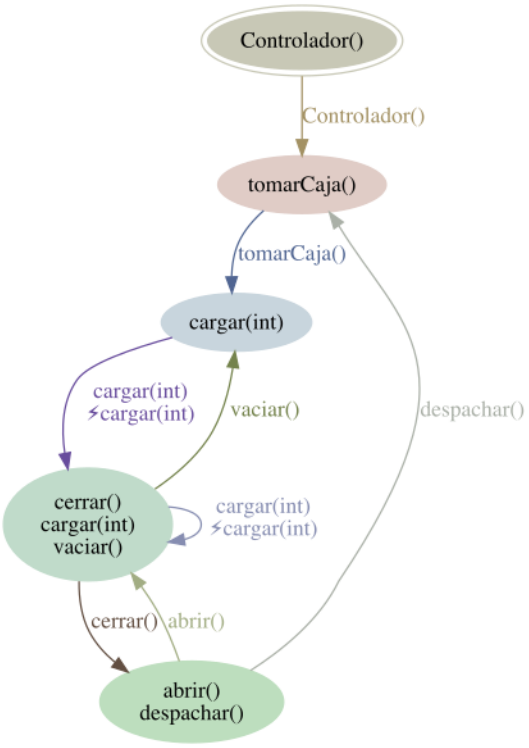

Fig. 6.6: PEPA del controlador del brazo robótico que lanza excepción

Se puede ver en la figura 6.6 que en todo estado, dependiendo de los parámetros que se utilicen, llamar a *cargar* puede llevar a que se arroje una excepción. Esto indica que la precondición es débil. También es interesante notar que en un caso el hecho de que *cargar* arroje una excepción cambia de estado a la *PEPA*, lo cual parece ser un error. Si se consulta la figura 2.6 puede verse que efectivamente la excepción es lanzada en un lugar incorrecto, alterando el estado interno de la clase mientras reporta que la operación no pudo concretarse.

6.2.3. Rotura de invariante debido a una excepción

Como último ejemplo, se pretende mostrar que una *PEPA* permite ver de manera sencilla si una excepción deja al objeto que la lanza en un estado inválido o si se puede seguir operando con este.

Tomamos la versión original del controlador y modificamos únicamente el método **cargar(int)** de modo que en caso de que se exceda el peso máximo el brazo cierre la caja en un intento de salvar la situación, y luego arroje una excepción. Se puede observar esta modificación en la figura 6.7.

```
public void cargar(int peso) {

  if (peso > 1000) {
    cajaCerrada = true;
    throw new RuntimeException("Carga demasiado pesada");
  }

  cajaVacia = false;
}
```

Fig. 6.7: Código de un método que lanza una excepción

Podemos observar la *PEPA* generada para esta versión en la figura 6.8. Como se esperaba, cuenta con transiciones donde cargar lanza una excepción. Lo que es interesante destacar es que la *PEPA* permite ver de manera clara que esto puede llevar a dos situaciones distintas. La primera es cuando se intenta cargar un objeto muy pesado en una caja vacía. En este caso el brazo robótico cerrará la caja, lo cual nos lleva a un estado de error, y el controlador no puede seguir siendo usado de manera segura. Por otro lado, si el problema ocurre con una caja que ya posee alguna carga, esta se cerrará y se lanzará una excepción, pero podemos seguir trabajando.

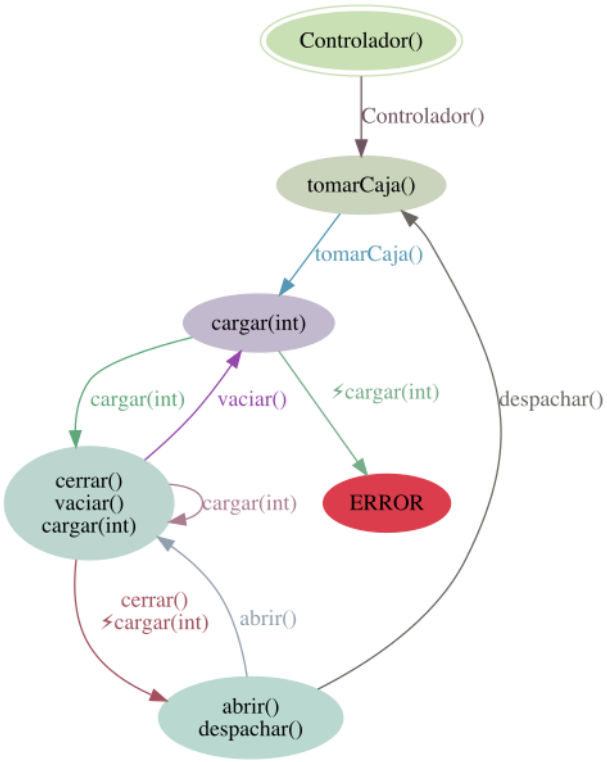

Fig. 6.8: PEPA del controlador del brazo robótico defectuoso

7. GENERADOR INTEGRAL DE *PEPAs*

Como se mencionó en capítulos anteriores, expresar de manera precisa las precondiciones de un método no es una tarea sencilla ni habitual en la práctica. Sin embargo, es necesario contar con los contratos de una clase para poder generar su *PEPA*.

Uno de los objetivos de este trabajo es mostrar el potencial que trae poder generar *EPAs/PEPAs* en *Java*, permitiendo integrar éstas a desarrollos de terceros. Con este fin, se diseñó e implementó un prototipo de una herramienta capaz de construir una *PEPA* a partir de una clase sin contratos explícitos.

7.1. Extracción de invariantes y precondiciones

Como se mencionó, debemos contar con el invariante de la clase y las precondiciones de sus métodos para poder generar la abstracción deseada, pero no contamos con estos expresados de forma explícita.

Es importante notar que tanto el invariante como las precondiciones, son propiedades de la clase, que siguen valiendo a pesar de no estar bien formuladas. Si se observa una cantidad significativa de trazas de distintas ejecuciones de un programa que haga uso de la clase, podrían inferirse dichas propiedades.

Es bajo este concepto que se creó *Daikon*[Ern+07], un detector de posibles invariantes. Esta herramienta ejecuta un programa, analizando el estado durante toda la ejecución, y computa invariantes válidos en diversos puntos de este. En particular, podemos usarlo para obtener el invariante de una clase, y las precondiciones de sus métodos. Es importante notar que los resultados obtenidos son subaproximaciones de las propiedades reales, y su calidad dependerá de el programa ejecutado.

La segunda herramienta utilizada en este prototipo es *Randoop*[PE07], un generador automático de casos de test. Ésta genera casos de tests de manera aleatoria, hasta alcanzar una cierta cantidad de estos o un máximo de tiempo indicado. Se utiliza a la misma para generar una suite de tests de regresión de la clase a analizar, los cuales tienen como objetivo documentar el comportamiento actual de la clase.

La figura 7.1 muestra la arquitectura de *Annotator*, que combina ambos proyectos.

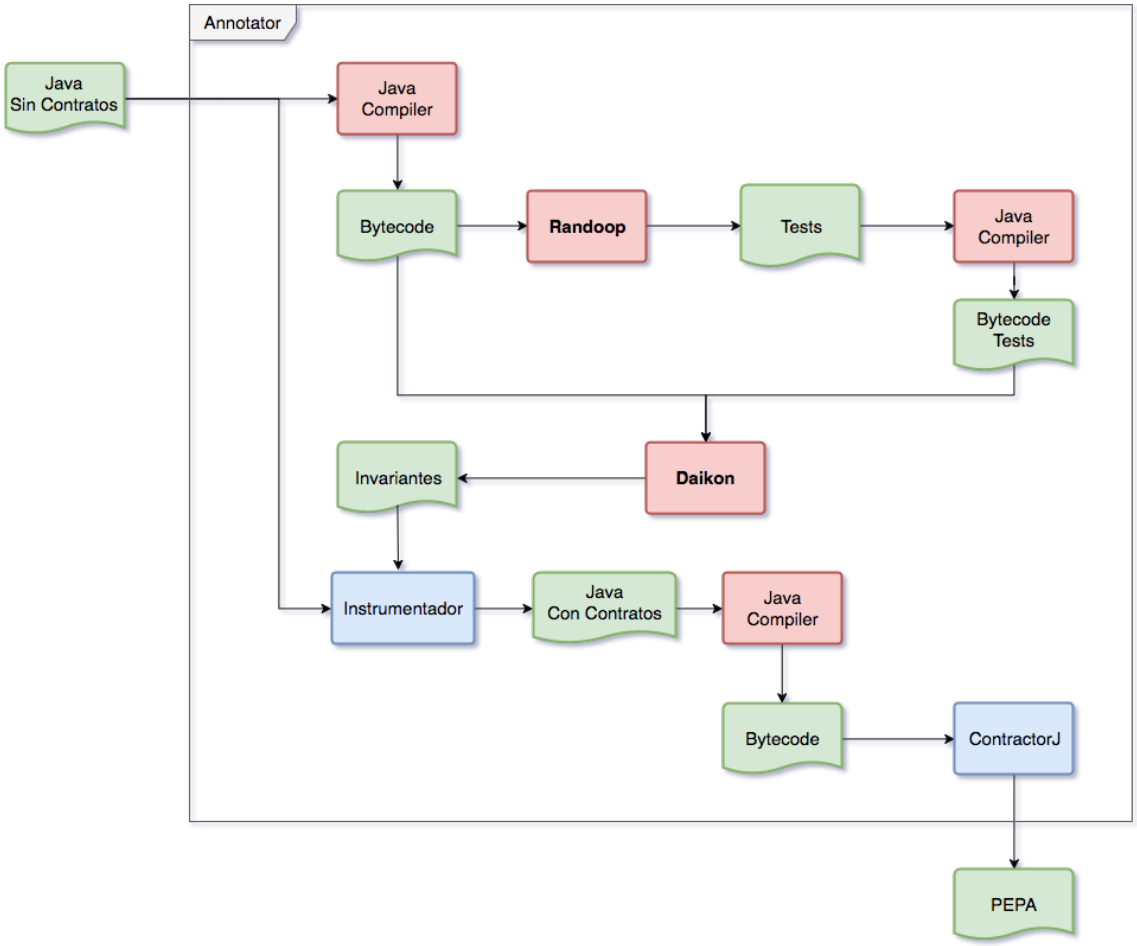

Fig. 7.1: Arquitectura de Annotator

La herramienta recibe como entrada el código de fuente de una clase *Java*, la cual es inicialmente compilada. Luego se utiliza *Randoop*, invocándolo como un proceso externo, que analiza el *bytecode* de la clase y procede a generar tests de regresión de manera aleatoria, dando lugar a una serie de clases con los mismos. Estas son compiladas, y utilizadas para invocar a *Daikon* de forma tal que se infieran invariantes a partir de la ejecución de ellas. Luego se *parsean* los resultados obtenidos, descartando aquellos que no pueden ser traducidos a *Java* de manera sencilla, o aquellos que traerían complicaciones al *model checker* utilizado por *ContractorJ*, como por ejemplo bucles o checkeos en runtime de tipos. Luego *Annotator* reescribe la clase original, agregándole contratos explícitos. Finalmente, se genera la *PEPA* a partir de ésta nueva versión de la clase.

7.2. Resultados

Se analizaron diversos casos de estudio para poner a prueba el funcionamiento de *Annotator*. Se exponen en esta sección algunos de ellos, junto a los resultados obtenidos.

7.2.1. FiniteStack

La figura 7.4 muestra una implementación de la estructura de datos clásica *Stack* o *Pila*, con un limite en la cantidad de elementos que puede contener a la vez. La misma no cuenta con información explícita sobre su invariante y precondiciones. Incluso se puede observar que los métodos **Push(T)** y **Pop()** no se resguardan de posibles accesos inválidos al arreglo **data**, lo cual podría considerarse un error.

Intentar generar una *PEPA* a partir del código sin contratos es equivalente a suponer al invariante y a todas las precondiciones siempre válidos. La figura 7.2 muestra el resultado que se obtiene realizando esto, el cual no es de utilidad alguna.

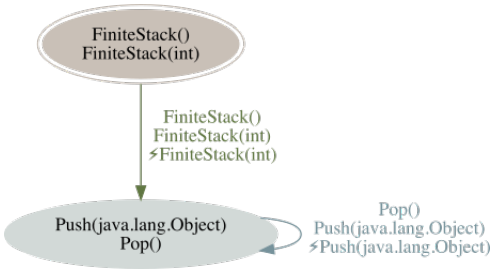

Fig. 7.2: PEPA de FiniteStack sin contratos

Se implementó también una versión con contratos explícitos de manera manual, y se generó la *PEPA* de ésta.

Luego, se corrió *Annotator* con esta clase como entrada, dando como resultado la *PEPA* de la figura 7.3, la cual coincide con la de la versión con contratos explícitos.

Cabe destacar que no sólo las abstracciones generadas son iguales, sino que también los contratos inferidos son lógicamente equivalentes a los escritos a mano.

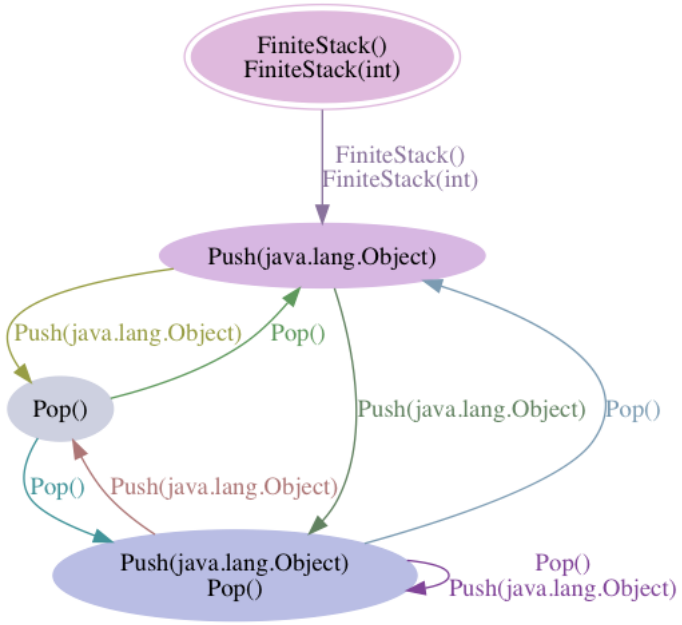

Fig. 7.3: PEPA de FiniteStack

```
public class FiniteStack<T> {

  private final Object[] data;
  private int size = 0;

  public FiniteStack() {
    this(5);
  }

  public FiniteStack(final int capacity) {
    if (capacity <= 0) {
      throw new IllegalArgumentException();
    }
    data = new Object[capacity];
  }

  public void Push(T item) {
    data[size++] = item;
  }

  public T Pop() {
    return (T) data[--size];
  }
}
```

Fig. 7.4: Código de FiniteStack

7.2.2. Controlador del brazo robótico

Se analizó también el controlador del brazo robótico del capítulo 2. Para ello se removieron las precondiciones y su invariante, y se agregó lógica en cada uno de los métodos,

de manera que se lance una excepción en caso de ser llamado de manera inválida.

La *PEPA* obtenida coincide con la de la versión con contratos explícitos, que al no tener excepciones ni errores, es idéntica a la *EPA* de la figura 2.2.

A diferencia del caso anterior, los contratos generados no coinciden en este caso. En particular el invariante inferido es simplemente **true**. Sin embargo, las precondiciones generadas son más estrictas que las escritas a mano, asegurando también que se cumpla el invariante, como en la figura 7.5.

```
public boolean cerrar_pre() {
    return (this.cajaCerrada == this.cajaVacia) &&
        (this.hayCaja == true) &&
        (this.cajaCerrada == false);
}
```

Fig. 7.5: Precondicion inferida para **cerrar()**

Notar que el modo de funcionamiento de *Daikon* dificulta la detección de disyunciones, y por lo tanto del invariante del controlador. El mismo cuenta con facilidades para ayudarlo con esta tarea, las cuales podrían usarse para la detección del invariante del controlador. Se decidió no exportar estos aspectos de *Daikon* en *Annotator* para facilitar su implementación y uso.

7.2.3. Signature

Se modificó de manera similar al capítulo 6 la clase *Signature* de la librería estándar de *Java*. En este caso se removieron los llamados a los métodos abstractos y la declaración de los mismos, dando lugar a una clase concreta.

A su vez, debido a que no se exportó en *Annotator* ningún método para indicar a *Randoop* qué instancias de una clase utilizar cómo parámetro a la hora de generar tests, se remplazaron los tipos de algunos argumentos por *String*.

La figura 7.6 muestra la *PEPA* generada en este caso. La misma es una sobreaproximación de la de su versión con contratos escritos a mano, y cuenta únicamente con dos transiciones más, las cuales corresponden a casos donde se lanza una excepción.

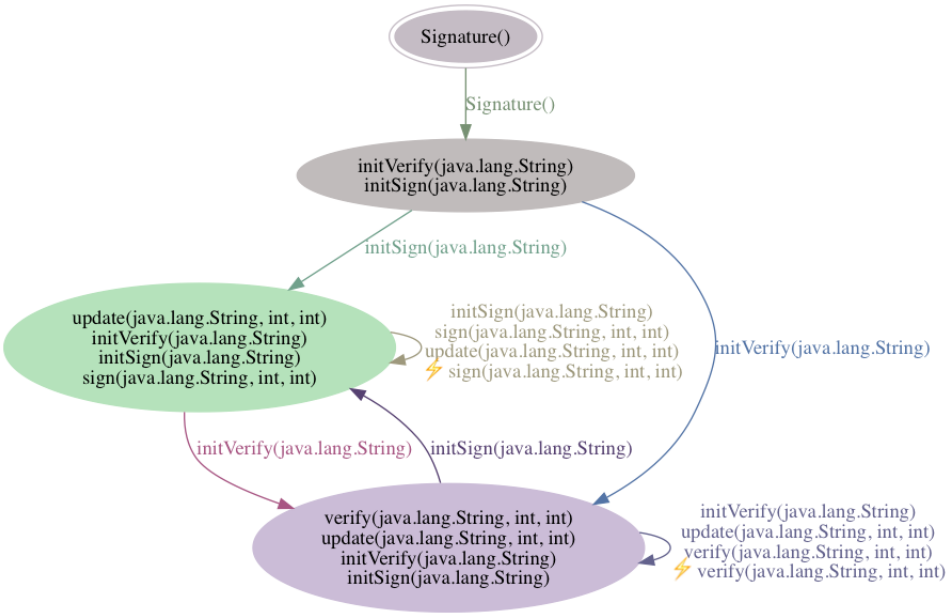

Fig. 7.6: PEPA de Signature sin contractos explícitos

La clase *Signature* cuenta con condiciones complejas sobre sus parámetros, como la que se muestra en la figura 7.7. Nuevamente *Daikon* no detecta de forma efectiva esta conjunción, por lo que aparecen transiciones que resultan en errores.

```
if ((signature == null)
    || (offset < 0)
    || (length < 0)
    || (offset + length > signature.length())) {
  throw new IllegalArgumentException("Bad arguments");
}
```

Fig. 7.7: Condición de Signature

7.2.4. ListIterator

Por ultimo vemos un ejemplo con contratos más complejos, donde *Annotator* no arroja resultados tan buenos como antes.

Se tomó la versión de *ListIterator* generada para la sección 6.1.1, se le removieron sus contratos, la protección contra modificaciones concurrentes, y se agregaron chequeos para validar los parámetros que recibe su constructor. Se agregaron también constructores redundantes para guiar a *Randoop* durante la generación de casos de test. Luego, se corrió nuestra herramienta.

La figura 7.8 muestra la *PEPA* obtenida. Debido a su tamaño se utilizó otra forma de organizar el grafo, que si bien no permite que ésta se interprete con facilidad, se pueden ver algunos aspectos erróneos en la misma.

El primero de ellos es que la cantidad de estados es distinta de la de *PEPA* de la figura 6.1. Recordemos que esta última coincide con la abstracción hecha a mano, que consideramos correcta, por lo que la generada por *Annotator* es errónea.

Por otro lado, se observa un estado que no posee ningún método habilitado. Notar que esto no es una violación del invariante, ni el resultado de una excepción, pues estos comportamientos se manifiestan de forma explícita en una *PEPA*. Por el contrario, este estado cumple con el invariante inferido, y las transiciones que llegan al mismo no simbolizan excepción alguna.

Analizando de forma más profunda el resultado se puede ver que el mismo no es ni una sub- ni sobreaproximación de la *PEPA* ideal. Creemos que esto se debe a la complejidad del invariante del módulo analizado, como a las limitaciones de la herramienta. Versiones futuras podrían exponer funcionalidades más avanzadas de *Randoop* y *Daikon*, para generar casos de test mas exhaustivos, y ayudar a encontrar invariantes más complejos, lo cual posiblemente genere mejores resultados.

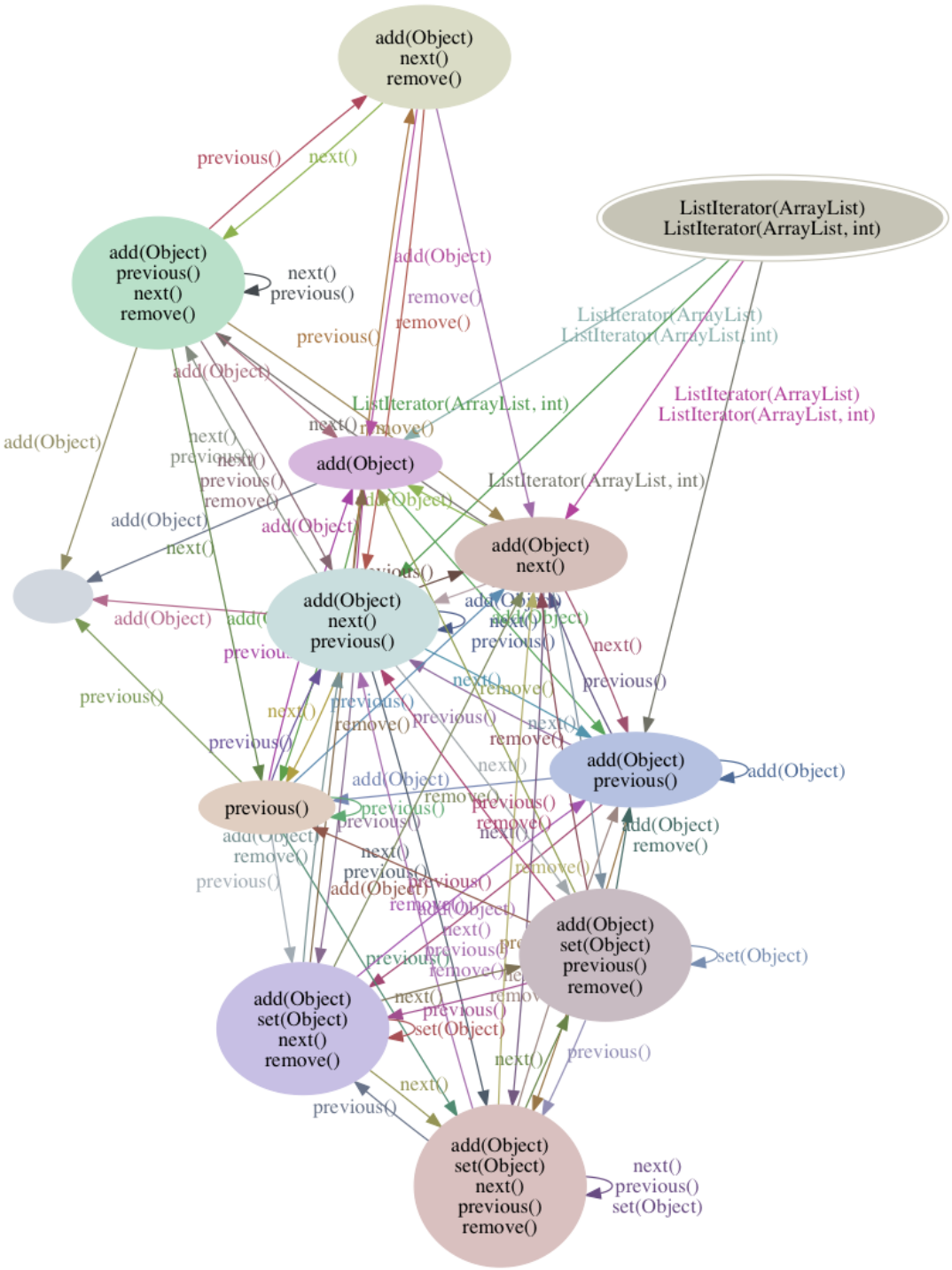

Fig. 7.8: PEPA de ListItertor sin contractos explícitos

8. CONCLUSIONES

En este trabajo presentamos *ContractorJ*, una herramienta que permite generar modelos de comportamiento compactos de clases de *Java*. Las abstracciones generadas se estudian en [Cas+11; Cas+13] y muestran utilidad para facilitar la depuración de errores y la generación automática de casos de test.

La herramienta está conceptualmente basada en la última versión de *Contractor.Net* [LG16], pero al ser enfocada en otra plataforma de desarrollo es una reimplementación total de la misma. Como era nuestro objetivo, los resultados obtenidos presentan una calidad equivalente o incluso superior a la versión ya existente.

Se mostraron a su vez dos limitaciones presentes en el modelo de *Enabledness-Preserving Abstractions* y por lo tanto en los generadores de estas. Se extendió dicho modelo de forma tal que sea más permisivo ante la presencia de errores en las especificaciones y la implementación de los distintos métodos, dando lugar a las *Permissive Enabledness-Preserving Abtractions*.

Se extendió *ContractorJ* de forma tal que genere nuestra versión extendida de las abstracciones, y se mostraron diferentes casos donde éstas son de mayor utilidad por sobre las *EPAs*.

Por último, se desarrolló un prototipo de una segunda herramienta. El mismo tiene como objetivo ser una breve presentación del universo de oportunidades que se abren al poder generar *EPAs*/*PEPAs* en *Java* y combinar esto con otros proyectos de investigación.

Annotator, dicho prototipo, hace uso de *Randoop* y *Daikon* para inferir el invariante y las precondiciones de los métodos de una clase sin contratos explícitos, y luego generar la *PEPA* de esta. Se mostraron resultados prometedores de ésta herramienta, donde las abstracciones generadas coincidían o eran muy similares a las obtenidas al expresar los contratos de forma explícita. Se vió también cómo la herramienta cuenta con limitaciones en casos donde los invariantes a detectar son complejos, y se propusieron mejoras para afrontar esto.

Consideramos que la generación de modelos de comportamiento, y en particular las *PEPAs*, son útiles tanto como documentación de una pieza de código, como para facilitar la detección de errores del mismo. A su vez, es de interés académico utilizar estas abstracciones en otras áreas, como se muestra en [CBU16], lo cual *ContractorJ* facilita.

9. TRABAJO FUTURO

Hay muchas mejoras y extensiones que pueden realizarse a este trabajo, tanto en los aspectos teóricos como ingenieriles del mismo.

ContractorJ puede analizar un gran cantidad de casos de estudio, pero se encuentra limitado por el alcance actual de *JBCT*. Continuar con el desarrollo del mismo, de modo que soporte un subconjunto más grande del *bytecode* de *Java*, ampliaría la cantidad de casos que pueden analizarse, y posiblemente haría que este sirva para otras aplicaciones.

Una caso particular de cómo podría extenderse el traductor desarrollado, es agregando un soporte parcial de subtipado. Este permitiría traducir de manera correcta el manejo de excepciones en *Java*. Una vez hecho esto, se podrá seguir trabajando en extender la expresividad de las abstracciones generadas, ya que una *PEPA* no distingue las transiciones con excepción según el tipo de la misma. Esto último se debe a que *JBCT* no lo permite.

Si bien fue desarrollado a modo de prototipo para exponer una idea, se obtuvieron buenos resultados con *Annotator*. Mucho se puede trabajar sobre el mismo, agregando nuevas funcionalidades, como exponer la posibilidad de guiar a *Randoop* y *Daikon* en sus tareas.

Por último, creemos que es interesante seguir buscando nuevas aplicaciones a las *PEPAs* y la extracción automática de modelos de comportamiento de software en general. Contar con *EPAs*/*PEPAs* en *Java* permite la colaboración con otros trabajos. Por ejemplo, en [CBU16] se muestra que el cubrimiento de *EPAs* es un buen criterio de *testing*. A su vez existen herramientas de generación automática de casos de *test* para *Java*, en particular *EvoSuite* permite generar una batería de *tests* optimizada para un criterio en particular, y podrían utilizarse las *EPAs* para esto.

BIBLIOGRAFíA

[Alu+05] Rajeev Alur y col. "Synthesis of Interface Specifications for Java Classes". En: *SIGPLAN Not.* 40.1 (ene. de 2005), págs. 98-109.

[ARS13] Stephan Arlt, Philipp Rümmer y Martin Schäf. "Joogie: From Java Through Jimple to Boogie". En: *Proceedings of the 2Nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*. SOAP '13. Seattle, Washington: ACM, 2013, págs. 3-8.

[BR02] Thomas Ball y Sriram K Rajamani. "The S LAM project: debugging system software via static analysis". En: *ACM SIGPLAN Notices*. Vol. 37. 1. ACM. 2002, págs. 1-3.

[Qad] Shaz Qadeer. *Respoitorio de BCT*. URL: `https://github.com/boogie-org/bytecodetranslator`.

[BH14] Stefan Blom y Marieke Huisman. "The VerCors Tool for Verification of Concurrent Programs". En: *Proceedings of the 19th International Symposium on FM 2014: Formal Methods - Volume 8442*. New York, NY, USA: Springer-Verlag New York, Inc., 2014, págs. 127-131.

[CBU16] Hernan Czemerinski, Victor Braberman y Sebastian Uchitel. "Behaviour abstraction adequacy criteria for API call protocol testing". En: *Software Testing, Verification and Reliability* 26.3 (2016). stvr.1593, págs. 211-244.

[Dal+10] Valentin Dallmeier y col. "Generating Test Cases for Specification Mining". En: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ISSTA '10. Trento, Italy: ACM, 2010, págs. 85-96.

[Cas+11] Guido de Caso y col. "Program Abstractions for Behaviour Validation". En: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA: ACM, 2011, págs. 381-390.

[Cas+13] Guido De Caso y col. "Enabledness-based Program Abstractions for Behavior Validation". En: *ACM Trans. Softw. Eng. Methodol.* 22.3 (jul. de 2013), 25:1-25:46.

[Cas13] Guido de Caso. *Modelos abstractos de comportamiento basados en habilitación*. 2013.

[DF01] Robert DeLine y Manuel Fähndrich. "Enforcing High-level Protocols in Low-level Software". En: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. PLDI '01. Snowbird, Utah, USA: ACM, 2001, págs. 59-69.

[DF04] Robert DeLine y Manuel Fähndrich. "Typestates for objects". En: *European Conference on Object-Oriented Programming*. Springer. 2004, págs. 465-490.

[Ern+07] Michael D. Ernst y col. "The Daikon system for dynamic detection of likely invariants". En: *Science of Computer Programming* 69.1–3 (dic. de 2007), págs. 35-45.

[GS08] Mark Gabel y Zhendong Su. "Symbolic mining of temporal specifications". En: *Proceedings of the 30th international conference on Software engineering*. ACM. 2008, págs. 51-60.

[GP09] Dimitra Giannakopoulou y Corina S. Păsăreanu. "Interface Generation and Compositional Verification in JavaPathfinder". En: *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*. FASE '09. York, UK: Springer-Verlag, 2009, págs. 94-108.

[HJM05] Thomas A. Henzinger, Ranjit Jhala y Rupak Majumdar. "Permissive Interfaces". En: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. Lisbon, Portugal: ACM, 2005, págs. 31-40.

[LQL12] Akash Lal, Shaz Qadeer y Shuvendu K. Lahiri. "A Solver for Reachability Modulo Theories". En: *Proceedings of the 24th International Conference on Computer Aided Verification*. CAV'12. Berkeley, CA: Springer-Verlag, 2012, págs. 427-443.

[Lei08] K Rustan M Leino. "This is boogie 2". En: *Manuscript KRML* 178.131 (2008).

[LG16] Leandro Lera Romero y Diego Garbervetsky. *ACorralando EPAs: acercando el modelo mental al computacional*. Abr. de 2016.

[LMP08] Davide Lorenzoli, Leonardo Mariani y Mauro Pezzè. "Automatic generation of software behavioral models". En: *Proceedings of the 30th international conference on Software engineering*. ACM. 2008, págs. 501-510.

[MRS15] Tim McCarthy, Philipp Rümmer y Martin Schäf. "Bixie: Finding and Understanding Inconsistent Code". En: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. ICSE '15. Florence, Italy: IEEE Press, 2015, págs. 645-648.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. 1st. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[NHT16] Suman Nakshatri, Maithri Hegde y Sahithi Thandra. "Analysis of Exception Handling Patterns in Java Projects: An Empirical Study". En: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. Austin, Texas: ACM, 2016, págs. 500-503.

[PE07] Carlos Pacheco y Michael D. Ernst. "Randoop: Feedback-directed Random Testing for Java". En: *OOPSLA 2007 Companion, Montreal, Canada*. ACM. Oct. de 2007.

[Sch15] Martin Schäf. *Respoitorio de Jar2Bpl*. Jun. de 2015. URL: `https://github.com/martinschaef/jar2bpl`.

[VH98] Raja Vallee-Rai y Laurie J. Hendren. *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. 1998.

[Val+99] Raja Vallée-Rai y col. "Soot - a Java Bytecode Optimization Framework". En: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, págs. 13-.

[ZG12] Edgardo Zoppi y Diego Garbervetsky. *Enriqueciendo code contracts con Typestates*. Dic. de 2012.